Systém pro automatizaci zpracování modelů testovaných řídících systémů

Bc. Michal Klhůfek

Diplomová práce
2020

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně

Fakulta aplikované informatiky

Ústav informatiky a umělé inteligence

Akademický rok: 2019/2020

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Michal Klhůfek**
Osobní číslo:	**A18262**
Studijní program:	**N3902 Inženýrská informatika**
Studijní obor:	**Informační technologie**
Forma studia:	**Kombinovaná**
Téma práce:	**Systém pro automatizaci zpracování modelů testovaných řídicích systémů**
Téma práce anglicky:	**A System for the Model-processing of Tested Control Systems**

Zásady pro vypracování

1. **Seznamte se s principy modelování SW/FW pro programovatelné hradlové pole.**
2. **Prostudujte možnosti a přístupy využívané při testování softwaru.**
3. **Seznamte se se strukturou souborů *.mdl a možnostmi jejich parsování.**
4. **Definujte požadavky na vstupní, výstupní soubory a funkcionalitu systému.**
5. **Navrhněte architekturu vyvíjeného systému.**
6. **Proveďte vývoj a implementaci systému dle sestaveného návrhu.**
7. **Verifikujte implementaci pomocí testovacích vzorových příkladů.**

Rozsah diplomové práce:
Rozsah příloh:
Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam doporučené literatury:

1. ŠŤASTNÝ, Jakub. FPGA prakticky: realizace číslicových systémů pro programovatelná hradlová pole. Praha: BEN – technická literatura, 2010. ISBN 978-80-7300-261-9.
2. PATTON, Ron. Testování softwaru. Praha: Computer Press, 2002. Programování. ISBN 80-7226-636-5.
3. ROUDENSKÝ, Petr a Anna HAVLÍČKOVÁ. Řízení kvality softwaru: průvodce testováním. Brno: Computer Press, 2013. ISBN 978-80-251-3816-8.
4. KARBAN, Pavel. Výpočty a simulace v programech Matlab a Simulink. Brno: Computer Press, 2006. ISBN 978-80-251-1448-3.
5. MYERS, Glenford J., Corey SANDLER a Tom BADGETT. The art of software testing. 3rd ed. Hoboken, N.J.: John Wiley, c2012. ISBN 1118031962.
6. LUTZ, Mark. Programming Python. 3rd ed. Sebastopol, CA: O'Reilly, 2006. ISBN 0596009259.

Vedoucí diplomové práce: **Ing. Peter Janků**
Ústav informatiky a umělé inteligence

Datum zadání diplomové práce: **28. listopadu 2019**
Termín odevzdání diplomové práce: **15. května 2020**

doc. Mgr. Milan Adámek, Ph.D.
děkan

prof. Mgr. Roman Jašek, Ph.D.
ředitel ústavu

Ve Zlíně dne 9. prosince 2019

Jméno, příjmení:

Název bakalářské/diplomové práce:

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové/bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová/bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové/bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji diplomovou/bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou/bakalářskou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování diplomové/bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové/bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové/bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na diplomové/bakalářské práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne

Michal Klhůfek, v. r.
podpis diplomanta

ABSTRAKT

Tato diplomová práce se zabývá problematikou testování logických programovatelných obvodů. Popisuje principy plánování a implementování zmíněných obvodů s tím, že hlavní důraz je zde kladen na automatické testování. V práci jsou popsány různé přístupy, podle kterých se provádí testování programovatelných obvodů se zaměřením na reálné využití v praxi. Práce také popisuje implementaci programu, který bude sloužit k automatickému testování zmíněných obvodů. Postupně jsou zde popsány jednotlivé části, ze kterých se aplikace skládá spolu se sadou testů, které ověřují její správnost. V závěru je zhodnocena práce jako celek.

Klíčová slova: logické programovatelné obvody, FPGA, Simulink, testování pomocí černé a bílé skříňky, automatizované testování, testovací vektory, testovací scénáře

ABSTRACT

This diploma thesis considers problematics of testing programmable logic circuits. It describes principles of planning and implementation of programmable logic circuits, with the main focus on automatic testing of these circuits. Thesis describes various approaches to programmable circuits testing with a focus on real use in practice. Along with this, thesis describes implementation of a program that will automatically test this programmable logic circuits. At the end, the implementation of the solution is evaluated, and the other parts of the application are described, together with a set of tests that verify its correctness. In conclusion it evaluated thesis as whole.

Keywords: programmable logic device, FPGA, Simulink, white box testing, black box testing, automated testing, test vectors, test cases

Tímto bych chtěl poděkovat svému vedoucímu panu Ing. Petru Janků, Ph.D. za cenné rady a odborné vedení při tvorbě této diplomové práce. Dále pak děkuji Ing. Radku Svechovskému za pomoc při řešení odborných problémů a za řadu konzultací, které jsme spolu měli. V neposlední řadě pak mé rodině a přítelkyni za podporu po celou dobu studia.

Prohlašuji, že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

OBSAH

ÚVOD

Logická programovatelná pole jsou v dnešní době nedílnou součástí takřka všech složitějších systémů, ve kterých je kladen důraz na možné rozšiřování funkčnosti systému bez potřeby nového hardwaru. Setkáváme se s nimi nejen v leteckém, automobilovém, výrobním, ale také například v robotickém průmyslu.

Trend jejich využívání v praxi bude neustále více aktuální, neboť jejich použití je pro řadu průmyslových odvětví stěžejní a postupným automatizováním okolního světa bude tato potřeba ještě větší. Programovatelné čipy mají velkou výhodu ve své znovupoužitelnosti například v úkolech, kde jejich stávající využití již není potřebné, nebo v případech, kdy je kvůli kritické chybě v simulačním schématu potřeba jejich dodatečné vylepšení.

Cílem této diplomové práce je vytvořit program sloužící k testování logických programovatelných schémat, který by mohl pomoci testerovi při jejich vyhodnocování a testování. Program bude vytvořen pro potřeby společnosti Honeywell a vychází z potřeby o zdokonalení stávajícího přístupu k testování zmíněných obvodů.

Práce je rozdělena na sedm kapitol. V 1. kapitole jsou uvedeny základní principy z oblasti modelování SW/FW pro programovatelná hradlová pole. Ve 2. kapitole jsou následně popsány možnosti a přístupy využívané při testování softwaru. Kapitola 3. si klade za cíl popsat strukturu souborů *.mdl a možnosti jejich parsování. Kapitola 4. poté definuje požadavky na vstupní, výstupní soubory a funkcionalitu systému, a je následována 5. kapitolou a 6. kapitolou, ve kterých probíhá návrh architektury vyvíjeného systému spolu s popisem vývoje a implementací systému dle sestaveného návrhu. V 7. kapitole jsou popsány metody testování a verifikování provedené aplikace za využití testovacích vzorových příkladů. V závěrečné kapitole jsou následně shrnuty a zhodnoceny cíle aplikace a jejich zdárné splnění. Spolu s tím jsou zde navrženy i další možná rozšíření a vylepšení fungování navržené aplikace.

I. TEORETICKÁ ČÁST

1 Logické obvody

Již od vzniku číslicové techniky byla potřeba sestrojení obvodů, které budou vykonávat základní logické operace. Díky tomu vznikly první logické obvody, které pracují pouze s diskrétními stavy reprezentovanými logickými hodnotami 0 a 1. Tyto obvody složí jako základ všech číslicových systému a jejich vzájemné kombinování umožnuje sestavovat složitější struktury. Postupem času nalezly své využití v realizaci řady logických funkcí. Obecně si pod pojmem logický obvod lze představit elektronický obvod řešící určitou logickou funkci.

Při vývoji logického obvodu existuje řada přístupů, jak lze daný obvod sestrojit. Tyto přístupy se dělí do tří základních skupin. První skupina obsahuje univerzální procesory či mikrokontroléry, které nabízejí vysokou míru integrace obvodu a velkou míru univerzálnosti použití. Druhá skupina obsahuje obvody označované zkratkou ASIC (Application Specific Integrated Circuits), což v překladu znamená na zakázku vytvořený obvod. Tyto obvody se vyznačují velmi vysokou rychlostí, ale za cenu toho, že jsou využitelné pouze pro daný konkrétní problém. Třetí skupinou jsou programovatelné logické obvody, které nabízejí možnost programování potřebného odvodu za zachování vysoké rychlosti. [1]

1.1 Historie programovatelných logických obvodů

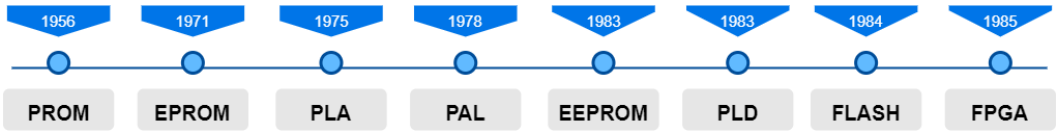
Nedlouho po vzniku prvních logických obvodů zde byla silná touha o vytvoření programovatelných logických obvodů. Za první programovatelný logický obvod lze považovat paměť typu PROM (Programmable Read-Only Memory) poprvé vyrobenou v roce 1956 vynálezcem Wen Tsing Chow v rámci armádního programu pro Spojené státy americké. Krátce po jejím uvedením na trh v roce 1969 byla její struktura zdokonalena a roku 1971 ji následovala pamět typu EPROM (Erasable Programmable Read-Only Memory), za jejíž vývojem stál dr. Dov Frohman. [1]

Dalším významným milníkem byl rok 1975, kdy byly na trh uvedeny první PLA (Programmable Logic Array). V témže roce se na trhu objevil čip IM5200. Jednalo se o první FPLA (Field Programmable Logic Array) čip na trhu a za jeho vývojem stál dr. Bill Sievers. V roce 1978 došlo k dalšímu vývoji a na trh byly předvedeny čipy PAL (Programmable Array Logic). Za jejím vývojem stáli dr. John Birkner, H.T. Chua a Andy Chan. Architektura čipů PAL vycházela z architektury PLA, ale dokázala nabídnout vyšší rychlost výpočtu a nižší náklady na výrobu čipu za cenu menší flexibility při použití. Společně s tím na trh vydali i jazyk PALASM (PAL Assembler), který samotnou práci s PAL čipy udělal ještě přívětivější. [1]

Vývoj nadále pokračoval i v devadesátých letech dvacátého století. Nejprve byla představena paměť typu EEPROM (electrically erasable programmable read-only memory) v roce 1983, která následovala paměť typu Flash v roce 1984. Za těmito pamětmi stál dr. Fujio Masuoka ze společnosti Toshiba. V roce 1983 došlo k vytvoření čipu pomocí architektury GAL (generic array logic), která již byla kompletně vymazatelná a přeprogramovatelná.[1]

V polovině devadesátých let dvacátého století se na trhu objevily tři odlišné přístupy jak pracovat s logickými programovatelnými poli, a to SPLD, CPLD (Simple/Complex Programmable Logic Device) a FPGA (field-programmable gate array). Jednotlivé přístupy se liší především ve své vnitřní struktuře, kde komplexnost SPLD či CPLD je více vhodná pro méně komplexní problémy z důvodu toho, že množství logických bloků je zde řádově nižší než jak je tomu u FPGA čipů, kde se můžeme setkat až s miliony logickými bloky. Z toho důvodu jsou SPLD a CPLD čipy levnější a vhodnější pro menší méně komplexní problémy.

Obvody typu SPLD a CPLD byly na trh představeny v roce 1983 firmou Altera. Tento přístup byl založený na několika úrovňové architektuře PAL čipů. Na druhé straně obvody typu FPGA byly na trh poprvé představeny v roce 1985 firmou Xilinx a za jejímž vývojem stál Ross Freeman. FPGA svým přístupem a univerzálností pro širokou škálu řešení dokázalo svou výhodnost v porovnání s architekturami SPLD a CPLD a díky tomu od roku 1995 začalo FPGA dominovat trhu s programovatelnými poli a jeho využití lze dnes nalézt téměř všude.[1] Časová osa zobrazující postupný vývoj programovatelných logických obvodů je znázorněna na obr.1.1.

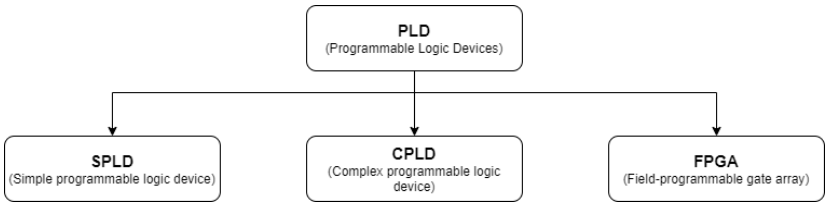


Obr. 1.1 historický vývoj programovatelných logických obvodů

1.2 Programovatelné pole

Pod pojmem PLD se souhrnně označují všechny programovatelné součástky, které mohou být elektricky mazány a znovu programovány k řešení složitých logických funkcí. Využití PLD čipů v průběhu vývoje umožnuje rychlé prototypování výsledného řešení v poměrně rychlém čase ve srovnání s čipy vytvořenými přímo na zakázku. Součástky typu PLD se následně dělí na tři dílčí podtypy. SPLD, CPLD a FPGA. [2] Typ SPLD se vyznačuje tím, že obsahuje jen jedno programovatelné pole. Typ CPLD se naopak vyznačuje tím, že obsahuje strukturu několika SPLD na jednom čipu. Poslední typ FPGA připomíná strukturu CPLD, ale obsahuje mnohonásobně vyšší počty logických hradel. Schéma znázornující rozdělení PLD čipů je zobrazeno na obr.1.2.

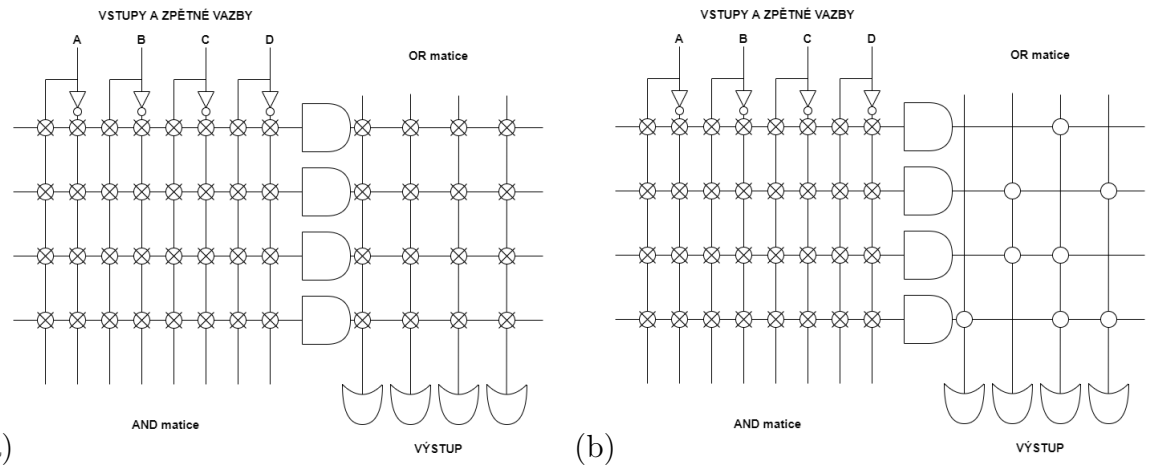
Z pohledu vnitřní struktury využívají obvody CPLD stejně jako paměti typu PROM kombinačních logických funkcí ve tvaru součtu součinů.[3] Obvody CPLD nabízejí programovatelné součinové pole a pevnou součtovou část, jak je tomu u čipů typu PAL, nebo variantu, kdy jsou programovatelné obě části, jak je tomu u čipů typu PLA. V současné době se obvody typu CPLD vyrábějí s rozsahem až do zhruba 15 000 ekvivalentních hradel. V porovnání s tím obsahují obvody FPGA jedno pole malých programovatelných buněk. Tyto buňky se k realizování patřičné operace musí propojit. Díky tomu je nalezení optimálního propojení mnohem složitější než u součástek typu CPLD. Výhodou FPGA ale je, že obsahuje řádově až miliony ekvivalentních hradel, které se dají k řešení daného problému využít.[2]



Obr. 1.2 schéma rozdělení programovatelných logických polí

1.2.1 Jednoduchá PLD (SPLD)

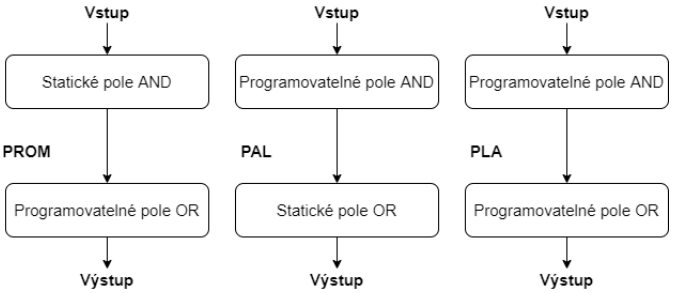
Struktura SPLD čipu se skládá ze skupiny hradel, které jsou tvořeny AND a OR logickými strukturami. Tyto struktury si můžeme představit jako pole AND logických bloků které vstupují do pole OR logických bloků. Architektura SPLD nabízí tři odlišné přístupy k samotné konstrukci výsledného logicky programovatelného čipu. Tyto přístupy se nazývají PLA, PAL a GAL. [2]



Obr. 1.3 schéma (a) popisuje čip typu PLA a schéma (b) popisuje čip typu PAL [5]

Při použití technologie PLA jsou k dispozici celkem čtyři vstupy a čtyři výstupy. Tento přístup ale nabízí pouze jednou programovatelný obvod, jeho následná rekonfi-

gurace již není možná. Technologie PAL vychází z technologie PLA, ale nabízí možnost rychlejšího vytváření potřebného logického obvodu a rychlejší zpracování dat v obvodu s odezvou okolo 5 ns. [3] Stejně jako u technologie PLA i tento přístup nabízí pouze jednou programovatelný obvod a jeho následná rekonfigurace není možná. Schéma pro čip PLA je zobrazeno na obr.1.3a a pro PAL na obr.1.3b. Poslední architektura, kterou jde u SPLD využít je technologie GAL, která vychází z paměti typu EEPROM a díky tomu je výsledný logický obvod znovu přeprogramovatelný. Vzájemné srovnání čipů typu PROM, PLA a PAL je zobrazeno na obr.1.4.



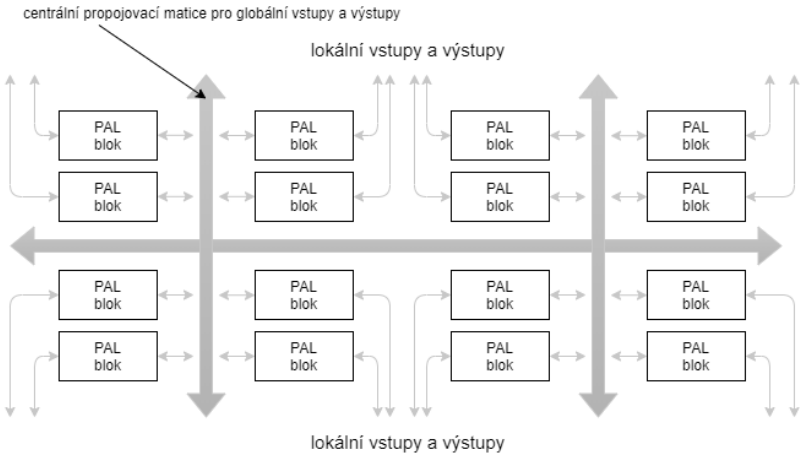
Obr. 1.4 vzájemné porovnání vnitřní logické struktury mezi PROM, PAL a PLA

1.2.2 Komlexní PLD (CPLD)

Čipy CPLD využívají ve své konstrukci řadu SPLD bloků vzájemně zapojených pomocí multiplexoru do matice přepínačů. Jejich konstrukce tak obsahuje velké množství logických a funkčních bloků, které jsou navzájem propojeny a které obsahují struktury typu PLA, PAL nebo GAL. Díky této zvýšené míře složitosti obvodu je umožněno programování více komplexních logických struktur. Průměrná doba odezvy na takovém čipu pak je kolem 8 ns. [3] CPLD jsou velmi vhodné pro kritické kontrolní aplikace, protože nabízejí i metody, jak předvídat chování obvodu v určitých časových úsecích. Mezi výrobci existuje řada různých typů, které se ale většinou liší jen ve struktuře programovatelné logiky a většina z nich vychází pouze z SPLD využívající PAL strukturu. Schéma znázornující CPLD čip je zobrazeno na obr.1.5.

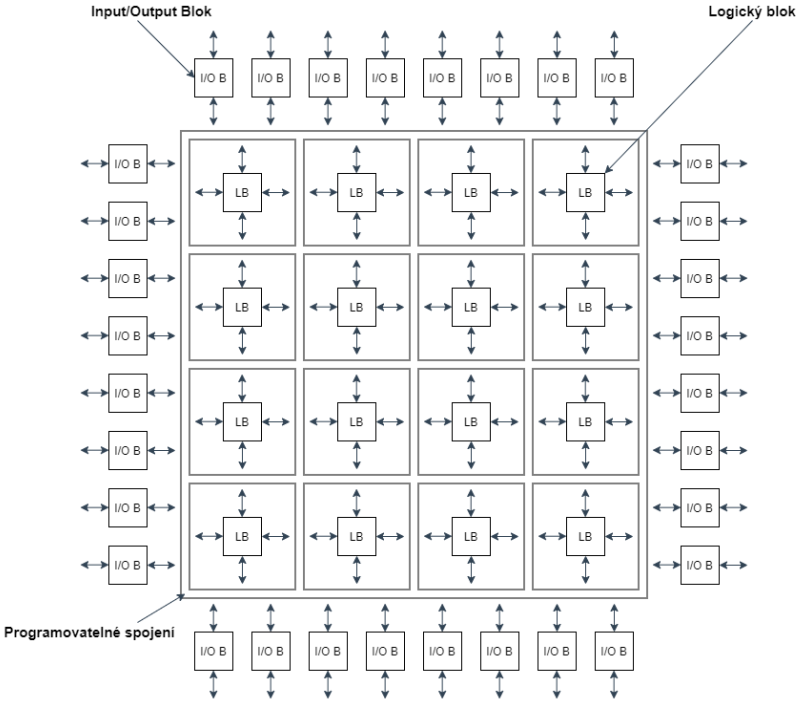
1.2.3 FPGA

Čipy FPGA jsou speciálním typem PLD obvodů. Jsou tvořeny logickými prvky v podobě hradel jako u čipů typu CPLD. Jednotlivá hradla jsou mezi sebou propojena jak ve vertikální, tak i horizontální podobě. Hradla nejsou tvořena dvojicí AND a OR logických prvků jako tomu bylo u PLA, ale jsou tvořena pomocí logických prvků typu NAND. Tato hradla jsou mezi sebou vzájemně propojeny. Těchto hradel může být na FPGA čipu až několik desítek až stovek tisíc. Díky tomu jsou FPGA čipy poměrně univerzální a jde pomocí nich naprogramovat takřka jakákoliv logická struktura.



Obr. 1.5 vnitřní schéma CPLD čipu [5]

Od mikrokontrolérů až po procesory, paměti, registry ALU jednotky či kontroly, a to vše na jednom programovatelném čipu.[2] Schéma znázornující rozdělení FPGA čip je zobrazeno na obr.1.6.



Obr. 1.6 vnitřní schéma FPGA čipu [5]

Mezi velké výhody FPGA patří jejich možnost znovu programovatelnosti. Nabízí možnost vytvářet stejné, ale i složitější logické struktury, podobně jako PLD čipy. Umožňují i paralelní zpracovávání dat, vzdálené úpravy systému pokud jsou potřeba, a navrhovat prototypy systémů ASIC, které představují zařízení šitá přímo na míru. Naopak mezi nevýhody FPGA jde zařadit jejich poměrně vysoká cena, a také neoptimalizovanost pro konkrétní problém kvůli jejich konceptu univerzálnosti na širokou paletu

problémů, na kterém jsou použity. Proto přespříliš komplikovaná struktura těchto čipů nemusí být vhodná pro všechny typy řešených problémů. [2] I tak lze FPGA považovat za velmi flexibilní čip, který nalezne své využití v mnohých průmyslových odvětvích.

1.3 Principy modelování SW/FW pro programovatelná hradlová pole

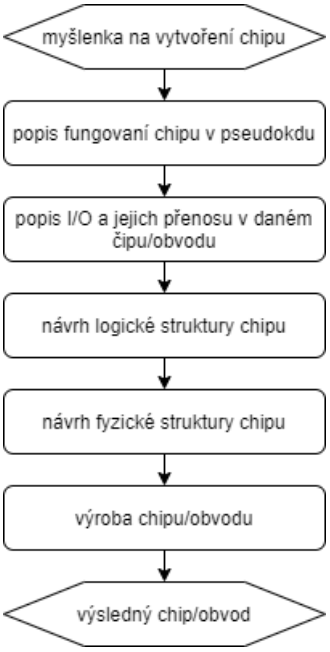
V počátcích vývoje výpočetní techniky obsahovaly jednotlivé logické čipy desítky až stovky tranzistorů, které zajišťovaly potřebné logické operace. V těchto počátcích bylo ještě možné dané čipy projektovat za použití papíru a tužky. Postupem času ale komplexnost čipů narostla a v současnosti je na čipu až biliony jednotlivých tranzistorů. Tento trend nárůstu počtu tranzistorů na čipu jde názorně vidět za pomoci Moorova zákona, který říká, že počet tranzistorů na čipu se každé dva roky zdvojnásobí. [7]

Díky tomuto dramatickému nárůstu složitosti čipů bylo nutné přijít i se složitějším a komplexnějším způsobem jejich návrhu. To vedlo ke vzniku speciálních nástrojů a programovacích jazyků, které tento problém řeší. Při samotném návrhu se tak vývojářům naskýtají možnosti nejen pro vytvoření požadovaného obvodu z logického hlediska, ale nabízí i snadnější přístup k optimalizaci logické struktury na čipu a tím zmenšení potřebné plochy. Tyto nástroje umožňují optimalizaci v podobě nižších nároků na elektrickou energii pro daný obvod či zvyšování rychlosti výsledného čipu. [10]

Aby bylo možné tyto obvody modelovat a programovat, bylo vhodné přijít se systémem standardizovaného vývoje. Tento systém se skládá z řady kroků, které je nutné dodržovat k vytvoření potřebného čipu. Celá struktura postupu začíná specifikací problému, který se bude řešit a končí výsledným návrhem hardwaru pro daný problém, který lze poté zaslat do výroby nebo nahrát do programovatelného čipu, například FPGA [7]. Doporučený postup při vývoji je zobrazen na obr.1.7.

1.4 Programovací jazyky pro popis HW a jejich přístupy k modelování HW

K programování a modelování logických obvodů se používá řada speciálních jazyků sloužících k modelování číslicového systému. Této skupině jazyků se souhrnným označením říká HDL (Hardware description languages). Tyto jazyky mají určité podobnosti, ale i řadu rozdílných vlastností v porovnání s ostatními programovacími jazyky, jako je například jazyk C. U klasického programovacího jazyka, například C, pracuje programátor s jazykem, který je kompromisem mezi vysokoúrovňovým jazykem a asemblerem, který se drží spíše na strojové úrovni. [3] Naopak u jazyka VHDL programátor pracuje s jazykem, který slouží čistě pro popis obvodů na čipu. Výrazným rozdílem mezi těmito jazyky může být také to, že u klasického programovacího jazyka lze vykonávat jeden proces v jednu chvíli, kdežto při využití VHDL je možné simulovat několik procesů



Obr. 1.7 schéma znázorňující doporučený postup při vývoji hardwaru [7]

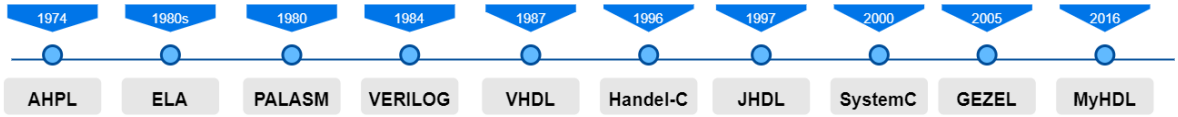
najednou, stejně jak by to probíhalo v reálném obvodu. Dalším významným rozdílem je také to, že k vytvoření programu v klasickém programovacím jazyce je potřeba znát jeho zákonitosti a řadu potřebných algoritmů k řešení problému. Na druhé straně u VHDL je nezbytná nejen znalost daného jazyka, ale také detailní znalosti o logických blocích. [3]

HDL jazyky slouží k popisu a modelování všech operací, které mají na daném čipu probíhat. Výsledný obvod je poté nahrán do speciálního programu zvaného Simulator, který umožňuje testovat a modelovat schéma obvodu přímo v počítači, aby se případná chyba dala odhalit ještě před výrobou čipu či před nahráním schématu do FPGA.

Existuje celá řada programovacích jazyků, které se využívají k programování HW. Nejrozšířenější jsou následující dva jazyky. Prvním z nich je VERILOG a druhým je VHDL. VERILOG se stal ve spojených státech amerických jedním z nejrozšířenějších jazyků pro popis HW. Tento jazyk byl poprvé představen v roce 1984 a za jeho vznikem stojí Prabhu Goel, Phil Moorby, Chi-Lai Huang a Douglas Warmke. Konkurentem VERILOGU je jazyk VHDL, který se stal nejrozšířenějším jazykem v Evropě pro programování HW. VHDL byl poprvé představen v roce 1987 pro americkou armádu. [3] Časová osa zobrazující postupný vývoj programovacích jazyků využívaných při vývoji hardwaru je zobrazena obr.1.8.

1.5 Využití Simulinku při modelování HW

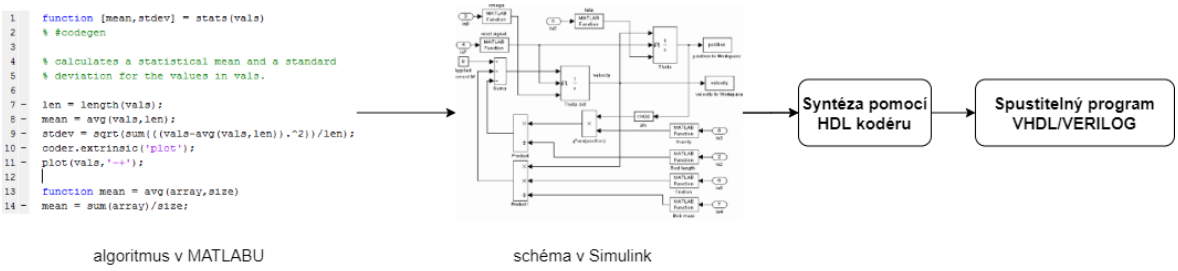
Kromě programovacích jazyků zmíněných v sekci 1.4 existuje ještě jeden způsob, kterým obvod naprogramovat. Tento způsob využívá programu Matlab a jeho vestavěného



Obr. 1.8 historický vývoj jazyků využívaných při vývoji harware

modelačního nástroje zvaného Simulink. Matlab je ideální nástroj ve kterém se dají zpracovávat velké shluky dat, aplikovat na ně potřebné logické operace a následně je graficky zobrazit. Kromě toho Matlab obsahuje i vestavěný Simulink, který představuje prostředí ve kterém lze potřebný obvod nasimulovat a otestovat, jak budou data systémem postupně zpracována. To umožňuje přidat hlubší implementační detaily do návrhu. Simulink poté umožní názorně zobrazit a odsimulovat paralelní architekturu celého navrhovaného systému. [6]

Aby bylo modelování obvodů za pomoci Simulinku ještě přívětivější obsahuje vestavěnou knihovnu, ve které jsou již obsaženy desítky logických bločků, které se dají využít pro návrh. Tento přístup byl také zvolen a je řešen v této diplomové práci. Kromě standardních knihovních funkcí budou touto prací zpracovávány i speciální logické bločky, které jsou sepsané v interní knihovně společnosti Honeywell a na kterých bude tato práce spuštěna a testována. Tyto logické bločky vycházejí ze standardních knihovních bločků a jsou následně doplněny o řadu komplexnějších a složitějších, které jsou v praxi využívány. Schéma znázornující vývoj za pomoci Simulinku je zobrazen na obr.1.9.



Obr. 1.9 schéma znázorňující postup při simulování obvodu za pomoci Simulinku[8]

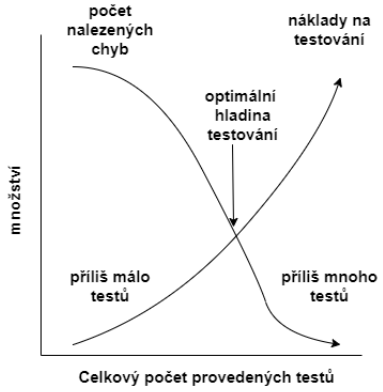
2 Testování Softwaru

Testování softwaru je proces, při kterém se kontroluje správnost a kvalita vyvíjeného softwarového produktu. Při tomto procesu se podrobně zkontroluje celý softwarový produkt a hledají se jeho chyby. Probíhá kontrola očekávaných výstupů sepsaných ve funkční dokumentaci produktu odpovídající reálným datům získaných z programu. Díky včasnému testování je možné případné chyby odhalit včas před uvedením produktu na trh. Pokud by se testování při vývoji vynechalo a chyby by se objevily až později při běžném provozu softwarového produktu, následné opravy v kódu by byly mnohem nákladnější, než kdyby se opravily ještě před uvedením na trh. Navíc zanechání chyby v softwaru může být nebezpečné nejen z bezpečnostního hlediska pro jeho budoucí uživatele, ale i z hlediska osobních údajů společnosti, která by program využívala. [10] Dá se tedy říct, že testování snižuje pravděpodobnost chybovosti produktu a čím dřív začne tím větší je šance, že se na případné problémy narazí včas.

Ke zvýšení kvality softwaru existuje řada vhodných metod jak při testování postupovat. Při využití těchto metod lze zjistit úroveň kvality výsledného produktu. To znamená, že ve výsledku se netestuje produkt jen vůči jeho projektové specifikaci, ale také způsobem, jakým ho bude cílový uživatel používat.

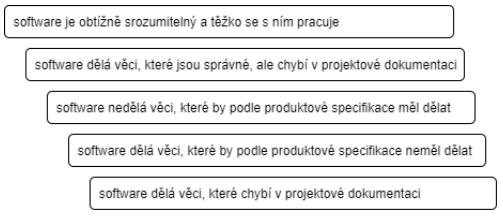
K dosažení vyšší kvality výsledného produktu lze využít široké řady norem, které poskytují patřičné rady a postupy. Například norma ISO/IEC 9126-1, která specifikuje šest základních požadavků pro určení kvality vyvíjeného softwarového produktu. Tyto požadavky jsou funkčnost, bezporuchovost, použitelnost, účinnost, udržovatelnost a přenositelnost. [11] Kromě těchto všeobecných doporučení se lze setkat i se směrnicemi cílenými na specifické průmyslové odvětví. Tato práce se zabývá testováním modelů pro FPGA využívaných v leteckém průmyslu a zde se využívají směrnice DO-178B, případně v novější verze DO-178C. Tyto směrnice popisují principy pro vývoj velmi kritického softwaru, ale také testování softwaru, který je třeba dodržet, aby šlo na konci vývoje řádně software i hardware certifikovat. [24]

Přes veškerou snahu o co nejvyšší kvalitu výsledného softwarového produktu se objevuje fakt, že kromě triviálních problémů není možné program otestovat zcela kompletně. S tímto rizikem je při vývoji softwarového produktu nutno počítat. I za předpokladu, že produkt bude testován v co největší míře, není tím možné zaručit, že se v něm již další chyby nenacházejí. Dokonce se při testování může stát, že se nalezne chyba, u které nejde s přesností vůči projektové specifikaci říct, zda se opravdu jedná o chybu či nikoliv. Z toho důvodu lze říct, že při testování je nutné stanovit určitou míru kompromisu kvality výsledného produktu. Jak přesně určit takovou míru kompromisu je poměrně složité. Pro lepší představu je problém znázorněn na obr.2.1.



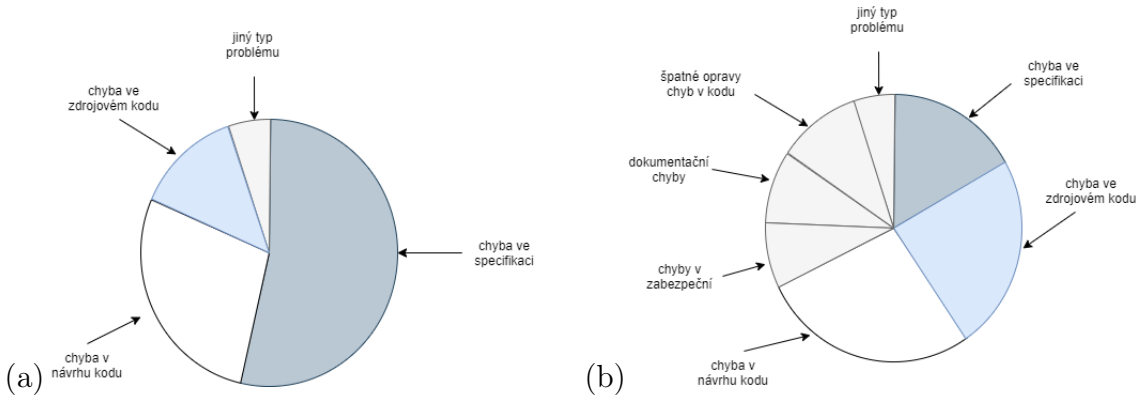
Obr. 2.1 schéma znázornující optimální množství testů [10]

Aby bylo možné s výsledným softwarovým produktem pracovat, je nezbytné důvěřovat jeho funkcím. To znamená, aby nedocházelo k selhání, které může být způsobeno rozdílnou implementací a specifikací produktu. Dále je nezbytné eliminovat defekty, které by mohl uživatel neopatrnou manipulací se systémem vyvolat. Posledním aspektem je zamezit výskytu chyb, které nebyly odhaleny předešlým testováním a které mohou způsobovat selhání celého systému. Při definování chyb vůči původní funkční dokumentaci lze problém popsat sadou pravidel poprvé sepsanou Ronem Pattonem v roce 2002 [10] a znázorněnou na obr.2.2.



Obr. 2.2 sada pravidel pro názornější popis chyb [10]

Při zkoumání řady softwarových projektů se zjistilo, že místa ve kterých vzniká nejvíce chyb se postupem času měnila. V roce 2002 přišel Ron Patton s grafem znázorněném na obr.2.3a, který uvádí, že největší podíl chyb je ve specifikaci a nejasné znění požadavků na softwarový produkt. To mělo několik důvodů, například nekompletní dokumentace či velmi častá potřeba po změnách ve specifikaci u řady projektů. Když v roce 2017 prováděl Jones Capers stejné měření zjistila, že za posledních patnáct let se situace na trhu zlepšila a nejčastější příčinou chyb v programech již není nepřesná specifikace problému, ale samotný kód programu se stal nejčastějším místem výskytu chyb, jak znázorňuje obr.2.3b. Tuto změnu výskytu chyb lze přisuzovat zavedení metodik při vývoji softwaru, dodržování patřičných norem, doporučení a využívání stále lepších nástrojů při návrhu systému. [12]



Obr. 2.3 (a) rok 2002 [10] (b) rok 2017 [12]

2.1 Přístupy k testování

Při testování je nutné určit, zda se bude provádět validace nebo verifikace. Oba pojmy slouží ke zjištění, jestli výsledný softwarový program dělá co je třeba a jestli odpovídá projektové specifikaci. Validace tedy určuje, zda výsledný produkt na konci svého vývoje odpovídá tomu, co od něj uživatel vyžadoval po stránce funkcionality. Na druhé straně verifikace slouží jako kontrola projektové specifikace pomocí sady dílčích testů. V závislosti na stylu vývoje softwarového produktu se můžeme s validací a verifikací setkat jak po celou dobu vývoje programu, tak v některých případech pouze v jeho finální fázi. [11]

2.1.1 Testovací metody

Před samotným testování softwarového produktu je nutné definovat množinu vstupů, očekávané výstupní hodnoty pro tyto vstupy, a množinu pravidel, které musí vstupy splňovat. Souhrnně se těmto informacím říká testovací případy. Tyto případy ověřují vždy pouze jeden konkrétní případ v dané aplikaci. Pokud by byly testovací případy přespříliš komplikované, může hrozit vznik chyby v samotném testu, který má chyby odhalovat. Jejich síla spočívá v jejich relativně malé velikosti. Při opětovném spuštění stejné testovací sady se získají pokaždé stejné výsledky. [10] Po každé změně ve specifikaci programu, která ovlivňuje daný testovací případ, je nutné jej aktualizovat.

2.1.2 Způsoby testování

Při testování softwarového produktu existuje řada kritérií, podle kterých se postupuje. V roce 2002 přišel Ron Patton s konceptem testování produktu formou černé nebo bílé skříňky. [10] Testování produktu formou černé skříňky představuje koncept funkčního testování. Při tomto způsobu tester nezná vnitřní strukturu programu ze strany zdro-

jového kódu a je nucen vycházet čistě z požadavků sepsaných ve funkční dokumentaci. Celý program si tedy lze představit jako černou skříňku, jejíž obsah není z venkovní strany patrný. Velkou výhodou tohoto přístupu je rychlost vytváření patřičných testů. Programátor nemusí znát detailní konstrukce jazyka, ve kterém je program vytvořen, stačí mu pouze znát výstupní hodnoty z programu, které porovnává s těmi, které jsou očekávané. Nevýhodou je, že neznalostí zdrojového kódu není možné zcela otestovat jeho komplexní správnost.

U testování pomocí metody bílé skříňky je tester obeznámen nejen se strukturou kódu výsledného produktu, ale také s projektovou dokumentací. Zde je mnohem větší šance na pokrytí a testování větší části kódu, jak ze strany neočekávaných vstupních hodnot, tak i ze strany nadbytečného zdrojového kódu, který nevychází z projektové dokumentace. Nevýhodou této metody v porovnání s černou skříňkou je v tom, že zde je nutná znalost detailní funkcionality jazyka, ve kterém je program vytvořen, aby byla zaručena adekvátní otestovanost. [9]

Postupem času byly tyto dva testovací přístupy rozšířeny ještě o koncept zvaný šedá skříňka. Jedná se o kompromis mezi černou a bílou skříňkou. [9] Díky tomu je tester obeznámen pouze s částečnou strukturou vnitřní implementace výsledného produktu. Využití může být například při testování webových aplikací, kde se tester sice dostane ke zdrojovému kódu dané aplikace, ale může se snadno podívat na zdrojový kód dané stránky napsaný v HTML. Schéma znázornující jednotlivé typy testování je zobrazeno na obr.2.4.

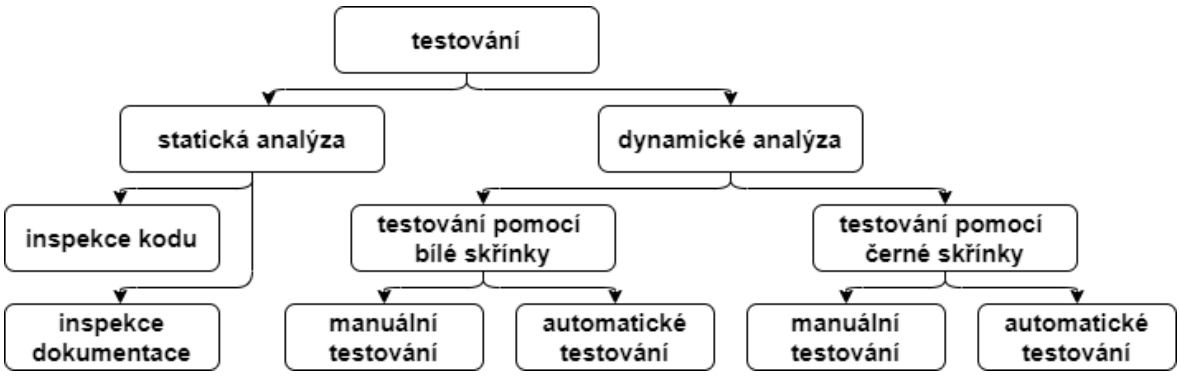
Obr. 2.4 schéma rozdílných typů testování

2.1.3 Statické a dynamické testování

Kromě předchozích zmíněných metod testování existují ještě dva důležité faktory, které ovlivňují přístup k ověřování softwarového produktu. Jde o statické a dynamické testování. Tyto přístupy se liší v tom, zda je potřeba mít k testování aplikaci zapnutou či nikoliv.

Statické testování je přístup, který nepotřebuje mít ještě hotovou celou spustitelnou aplikaci. Testování probíhá přímo na zdrojovém kódu a není třeba jej spouštět. Často se může jednat o testování dodržování správných standardů při vývoji, hledání syntaktických chyb, případně využívání správných specifikací, využívání vhodných datových typů jazyka nebo testování dokumentace popisující chování dané aplikace. Díky tomu, že ke kontrole není nutné mít hotový výsledný produkt, je možné toto testování provádět již od počátečních fází vývoje softwarového produktu. S možností testování kódu v počátečních fází lze docílit značné finanční úspory, pokud by byla případná chyba nalezena. [11]

Dynamické testování je bezesporu mnohem náročnější, než testování statické, a lze ho uplatnit až v pozdějších fázích vývoje, kdy je již spustitelná aplikace, a tím je mnohem více důkladnější. Uživatel s ním přímo komunikuje za běhu aplikace. Postupně mu vkládá či zasílá vstupní data a kontrolu patřičných výstupních hodnot. Pomocí této metody se postupně sledují vlastnosti systému a jeho chování nejen po stránce očekávaných výstupních hodnot, ale také po stránce využívání paměti. Schéma znázorňující rozdílné přístupy v testování je zobrazeno na obr.2.5.

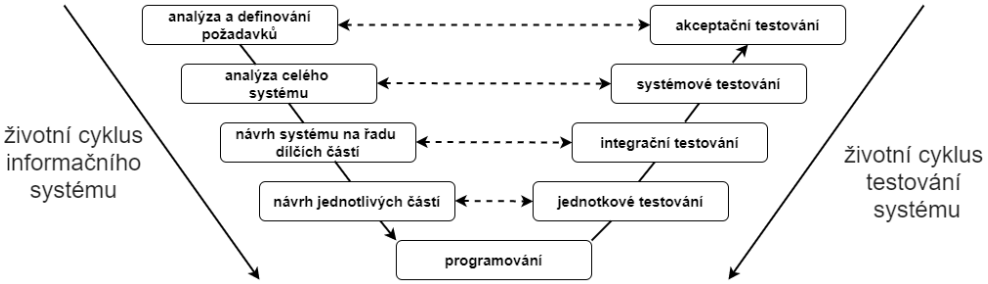


Obr. 2.5 rozdíly mezi statickým a dynamickým testováním [11]

2.1.4 Životní cyklus testování SW

Při určování kvality výsledného softwarového produktu je vhodné postupovat podle předem naplánované sady kroků testování. Jednou z metod, která tyto kroky popisuje je vodopádový model. Tento model nabízí nejen seznam kroků podle kterých lze daný produkt vytvořit, ale i seznam kroků, podle kterých lze postupně daný vývoj kontrolovat a testovat. Vodopádový model je znázorněn na obr.2.6.

V závislosti na úrovni vývoje lze vybrat jeden z následujících kroků. V prvním kroku dochází k analýze požadavků zákazníka, aby bylo jasné, co přesně se má vytvořit. Tomuto kroku odpovídá vytvoření sady akceptačních testů, které musí výsledný softwarový produkt splňovat, aby produkt odpovídal požadavkům zákazníka. Tyto testy



Obr. 2.6 schéma vodopádového modelu [11]

často provádí sám zákazník a může si tak přímo zkontrolovat správnost systému. Samotné testování se dá rozdělit na dvě části. V první části nazývané alpha testování se výsledný produkt testuje na straně vývojáře, kdy zákazník má plnou kontrolu, ale vývojář poskytuje případnou podporu. Následují část nazýváme beta testování. To již probíhá plně na straně zákazníka, který informuje vývojáře o případných nejasnostech či chybách.

Jakmile jsou požadavky patřičně specifikovány, dochází k analýze navrhovaného systému. Tento krok odpovídá vytváření sady systémových testů. Tyto testy si kladou za cíl otestoval celkovou aplikaci. Aby šlo tyto testy spustit, je nutné mít již aplikaci kompletně vytvořenou, a proto se tyto testy spouštějí až v pozdějších fázích vývoje. Snahou je co nejvíce do hloubky otestovat funkčnost celé aplikace ještě dřív, než se pošle zákazníkovi.

Následujícím krokem je návrh celkového systému pomocí dílčích komponent, případně modulů a jeho adekvátní testování v podobě integračních testů. Tyto testy si kladou za cíl ověřit, zda jednotlivé komponenty a moduly mezi sebou bezchybně komunikují, případně že jejich komunikace s okolím v podobě jiných programů, vstupních čidel či samotným operačním systémem probíhá zcela bez problému. Toto testování je hodně podrobné a probíhá na úrovni kontrolování zdrojového kódu.

Ještě detailnější testování probíhá na úrovni plánování a navrhování funkcionality pro samotné moduly a dílčí komponenty na úrovni zdrojového kódu. Touto fází končí plánování a vhodné rozdělování programu na dílčí podčásti. Zbývá poslední fáze, kde je třeba tyto podčásti naprogramovat. Aby bylo jasné, že dané komponenty a moduly jsou implementovány správně, využívá se sada jednotlivých testů, někdy nazývaná Unit testy. Tyto testy jsou velmi často vytvářeny přímo vývojářem systému k ověření, že správně implementoval co je očekáváno. V této části testování je třeba velmi dobře znát strukturu zdrojového kódu. Jedná se o jedny z nejdůležitějších testů při testování softwarového produktu, neboť se zde začínají testovat nejmenší části ze kterých je produkt sestrojen. Velká výhoda nalezení chyby v této části systému spočívá v tom, že je poměrně snadno opravitelná, bez hlubšího dopadu na ostatní části celého systému [11].

2.2 Testovací nástroje

Při testování softwarového produktu existují dva odlišné přístupy jakými se lze ubírat, a to testování manuálně nebo automaticky. Každý z těchto přístupů má své značné výhody i nevýhody. Při manuálním testování je vývoj testů poměrně náročný, zdlouhavý a vyžaduje značné množství testerů na vytváření těchto testů. Díky tomu existuje velká šance, že i do testu, který má ověřovat funkci programu bude zavedena chyba. Manuální testování je velmi výhodné pro menší projekty, kde by mohlo být vytvoření automatických testů zbytečně komplikované. Dalším případem, kde je vhodné využívat manuální testování alespoň částečně, jsou velmi kritické systémy, kde je potřeba zaručit patřičnou míru důvěry daného kódu a nelze se jen spoléhat na automatické testy. Vytvořené manuální testy je nutné opakovaně spouštět, aby se zaručilo, že přidání nové funkcionality nenarušilo již stávající funkce systému.

Na druhé straně lze k vytváření testů využívat automatické testovací nástroje. Tyto nástroje mají značnou výhodu ve spolehlivosti, neboť i po opakování spuštění na stejný problém pokaždé dají stejnou odpověď. Díky tomu jsou výsledky testů vždy přesné v závislosti na způsobu jakým byly vytvořeny. Jediné co je na začátku vyžadováno je specifikování testových případů na které se mají testy zaměřit. Pomocí toho není vyžadována detailní znalost daného problému, který chceme testovat, ale stačí znalost testovacího nástroje. Vše ostatní probíhá zcela automatizovaně. Přes tyto výhody automatizované testovací nástroje nemohou zcela nahradit práci testera, ale mohou poskytnout cenný nástroj k dosažení co nejvyšší kvality softwarového produktu.

Na trhu existuje celá řada nástrojů pro automatizované testování softwaru, ale pro potřeby této práce jsou zde zmíněny jen ty nástroje, které umožňují testování softwaru sloužícího k vytváření hardwarové reprezentace obvodů. Jejich přehled je znázorněn v tabulce 2.1. Tyto nástroje se často liší zejména v možnostech testovacích modulů. Mohou obsahovat moduly zaměřené na tvorbu testovacích scénářů, moduly na správu nalezených defektů či moduly na generování samotných testů. Čím větší množství jednotlivých modulů je nabízeno, tím vyšší je míra univerzálnosti a použitelnosti daného testovacího nástroje. Tyto přídavné moduly ale sebou přináší i vyšší pořizovací náklady na takový testovací nástroj. Alternativním přístupem poté mohou být nástroje, které jsou zdarma, případně pod otevřenou licencí, a jejich vývoj provádí samotní uživatelé v rámci komunity programátorů. Poslední alternativou jsou firemní interní řešení na správu testů. Jedním z takových řešení je právě tato práce, která si klade za cíl vytvořit automatický testovací nástroj pro testování logických obvodů pro letecký průmysl vyvíjených společností Honeywell.

název produktu	rok uvedení na trh	krátký popis produktu
IST-FPGA	2008	integrovaný testovací framework pro práci a testování FPGA obvodů
ScanWorks	2003	tetovací framework pro FPGA ovody s širokou nabídkou modolů pro testování
VUnit	2003	open source framework pro testování obvodů napsaných ve Verilogu
ChipVORX	2013	framework od společnosti Gopel elektronic pro hloukové testování schémat v FPGA
HAVEN	2011	jedná se o univerzální testovací framework vyvinutý na univerzitě VUT
MyHDL	2010	jedná se nejen o vývojové prostředí kde jde schéma vytvořit přímo v jazyce python, ale také ho zde přímo otestovat
SEU Simulation Framework	2011	testovací framework pro FPGA od společnosti Xilinx

Tab. 2.1 seznam optimalizačních nástrojů

2.3 Testování schématu pro programovatelná hradlová pole

Obdobně jako testování klasických softwarových aplikací je nezbytné testovat i schémata pro programovatelná hradlová pole. Zde jsou základní dva přístupy podle kterých se testování provádí. V prvním přístupu se jedná o možnost testování přímo ve vývojovém prostředí ve kterém je schéma vytvářeno. Tak se získají výstupní odezvy na spouštěné schéma programu přímo ze simulačního prostředí. Tento přístup má výhody v tom, že nepotřebuje mít připojený výsledný například FPGA čip a data do něj nahrávat před spuštěním simulace a testováním. Nevýhodou je jeho nižší rychlost v porovnání s testováním složitějších obvodů ve srovnání se simulováním přímo na FPGA čipu.[14] Při nahrání dat do čipu by se měly získat stejné výsledky simulace jako při testování pomocí simulátoru. Pokud by bylo třeba simulovat chybné chování obvodu pomocí zasílání chybných vstupních signálů je výhodnější nahrání dat přímo na čip FPGA. Tímto přístupem lze zajistit, že výstupy odpovídají reálnému chování čipu při nevhodných vstupních impulsech místo jejich simulování ve vývojovém prostředí.

Po výběru vhodného přístupu ke spouštění schématu vytvořeného pro programovatelná hradlová pole zbývá určit, jak detailně se bude samotné testování provádět. Při testování zvoleného schématu je nutné nejprve určit, zda se bude testovat za běhu systému nebo při vypnutém systému. Testování za běhu systému se často také nazývá online testování, při vypnutém stavu systému bývá v literatuře označováno jako offline testování. [13] Dalším aspektem při výběru testování je komplexita schématu, kterou lze určit, zda testovat celý obvod nebo jen jeho dílčí části, které jsou pro testování dostatečně robustní. V této diplomové práci je využita kombinace offline testování v kombinaci s testováním podčástí celého schématu. Důvodem je značná rozsáhlost reálného schématu popisující celou základní desku leteckého motoru.

Výsledné testování je poté velmi obdobné testování softwarového produktu. Nejprve je nutné připravit sadu testovacích vektorů pro jednotlivé logické členy v daném

schématu. Každý jednotlivý člen je pak otestován v patřičném rozsahu svých hodnot a zkoumá se, zda je jeho chování stejné, jaké uvádí specifikace logického členu či nikoliv. Kromě toho dochází navíc ke zkoumání komunikace s ostatními bloky, zda nedochází k neočekávanému chování. Problém při testování logických schémat může nastat v případě jejich modifikace. I malá změna ve schématu může mít velmi rozsáhlý dopad na testy, neboť je potřeba znovu zkontrolovat, jestli obvody ve schématu po provedené změně správně mezi sebou komunikují.[14]

3 Soubory využívané při modelování v Simulinku

Jednou z možností modelování logických obvodů je využití vývojového prostředí Matlab a jeho rozšíření Simulink od společnosti Mathworks Inc. Program Matlab byl na trhu poprvé představen v roce 1984 jako nástroj pro snadnější řešení matematických problémů. Při jeho postupném vývoji došlo k vytvoření rozšíření zvaného Simulab, které bylo představeno v roce 1991. Toto rozšíření bylo poté v roce 1992 znovu upraveno a přejmenováno na Simulink. [8]

Simulink je nástroj pro simulování, modelování a analyzování dynamických logických obvodů. Avšak většina vestavěných systémů a systémů pracujících s reálným časem se kterými se lze setkat v běžném životě představují kombinované systémy. Pro tyto systémy je typické, že mohou mít dynamické i statické chování. Dynamické systémy je možné následně modelovat za pomoci Simulinku. Samotné modely poté mohou být sestaveny z předem definovaných logických bloků obsažených v knihovnách, nebo mohou být napřímo definovány pomocí sady rovnic [8], podobně jak to bylo popsáno v kapitole 1.5.

Při vývoji modelů popisujících chování logických obvodů ve vývojovém prostředí Simulink jsou využívány soubory typu *.mdl (model description language), ve kterém jsou modely uloženy. Tyto soubory obsahují kompletní informace o daném obvodu z pohledu jeho logické struktury. Samotný soubor *.mdl se skládá ze dvou částí, které postupně definují jednotlivé nezbytné součásti modelovaného obvodu.

První část popisuje model, jeho jméno a sadu parametrů, které slouží k jeho jednoznačné identifikaci. V této části se nacházejí dvě podsekce, které obsahují možnosti k předdefinování sady parametrů pro daný obvod. V první podsekci jsou popsány všechny logické bloky, které jsou v obvodu využívány s jejich předdefinovanou hodnotou. To znamená, že pokud je zde definován blok, konstanta s hodnotou jedna, tak na všech místech v obvodu mu bude tato hodnota nastavena. Druhá podsekce je zcela dobrovolná a popisuje seznam všech předdefinovaných popisů či komentářů pro jednotlivé bloky. Pokud bude v obvodu někde použit blok pro sčítání a nastaví se komentář, zobrazí se u všech bloků stejného typu v daném obvodu.

Ve druhé části nastává popis samotného systému. I v této části je struktura rozdělena na tři podsekce, které obsahují důležité informace. V první podsekci jsou definovány všechny bloky nacházející se v daném obvodu. Tyto bloky mohou být vstupní/výstupní signály nebo logické bloky sloužící k realizaci daného schématu. Kromě případů, kdy je blok referencován do příslušné knihovny, ve které byl vytvořen, je možné zde uvést podrobný popis, jak se daný blok vytvoří bez využití knihoven. Tento postup je použit pouze ve speciálních případech nebo když se v knihovně bloků potřebný blok nenachází. V následující podsekci je seznam všech spojů, které vzájemně propojují jednot-

livé bloky či vstupní/výstupní signály. V poslední části je dobrovolný seznam komentářů pro jednotlivé bloky. Pro lepší představu je struktura *.mdl souboru znázorněna na následujícím pseudokódu.

```
Model {
  <Model Parameter Name> <Model Parameter Value>
    <Models parameters descriptions>
  BlockDefaults {
    <Block Parameter Name> <Block Parameter Value>
      <Blocks parameters descriptions>
  }
  AnnotationDefaults {
    <Annotation Parameter Name> <Annotation Parameter Value>
      <Annotations parameters descriptions>
  }
  System {
    <System Parameter Name> <System Parameter Value>
      <Systems parameters descriptions>
    Block {
      <Block Parameter Name> <Block Parameter Value>
        <Blocks parameters descriptions>
    }
    Line {
      <Line Parameter Name> <Line Parameter Value>
        <Lines parameters descriptions>
      Branch {
        <Branch Parameter Name> <Branch Parameter Value>
          <Branchs parameters descriptions>
      }
    }
    Annotation {
      <Annotation Parameter Name> <Annotation Parameter Value>
        <Annotations parameters descriptions>
    }
  }
}
```

Code 3.1 struktura mdl soubouru

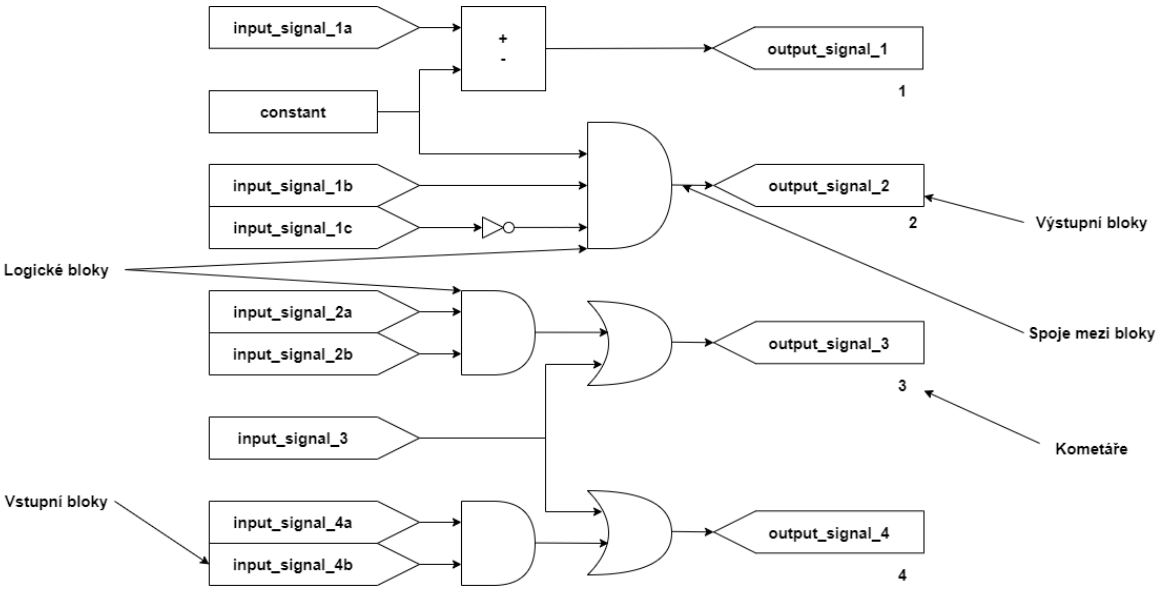
3.1 Modelové schéma obvodu v *.mdl

Při sestrojování nového logického schématu ve vývojovém prostředí Simulink se nejčastěji využívají předem vytvořené bloky. Tyto bloky se nacházejí v řadě knihoven a to jak základních, které jsou přímo zabudované v nastavení Simulinku, tak i řady uživatelských pro specifické a značně složitější bloky. Knihovny obsahují kompletní podrobné informace o chování daných bloků a není proto nutné je znovu implementovat. Práce s takovou knihovnou je poté celkem intuitivní a stačí pouze daný blok nahrát do grafického uživatelského prostředí zkráceně GUI (Graphical User Interface) a nastavit mu následně potřebné parametry. Díky tomu dochází ke značné úspoře času při vývoji schématu v porovnání s vývojem v klasickém modelovacím nástroji.[15] Následuje seznam nejčastěji využívaných bloků při modelování:

- základní logické bloky OR, AND, NOT, XOR, NAND a NOR
- bloky které se zpožďují s časem například CONFIRM SEC

- bloky na detekci chyb
- bloky pro konstanty a absolutní hodnoty
- bloky upravující maximální rozsah signálu
- derivační a integrační bloky
- děličky, násobičky, sčítačky signálu
- klopné obvody a bloky na filtraci signálu
- porovnávací, hystereze bloky a bloky na signálový zisk
- bloky pro subsystémy a speciální bloky typy StatusWord
- bloky k porovnání minimální či maximální hodnoty signálu

Pro názornější představu je na obr. 3.1 znázorněno schéma jednoduchého logického obvodu, který je složen ze dvou dílčích logických celků. Schéma obsahuje pět základních částí, které je nutné při jeho sestavení a následném testování správně zohlednit a analyzovat. Jedná se o vstupní a výstupní signály, logické bloky, spoje mezi jednotlivými bloky a komentáře pro dané bloky.



Obr. 3.1 schéma jednoduchého logickoho obvodu

Vstupní signály představují data vstupující do daného schématu. Mohou být získána dvěma způsoby. Prvním je vstup přímo z externího snímače například pro monitorování tlaku v nádrži s palivem. Druhý způsob představuje situaci, při které je na vstup přiveden signál, který byl vypočítán v jiné části schématu. U těchto signálů je nutné

rozhodnout, zda se jedná o signály s přepisovatelnou či nepřepisovatelnou hodnotou. Přepisovatelné signály jsou takové, u kterých není třeba znát způsob jejich výpočtu v předešlém schématu, protože bylo zaručeno a otestováno, že mohou nabývat libovolné hodnoty, nejčastěji logické nuly i jedničky. Nepřepisovatelné signály jsou takové, u kterých to zaručit nelze a pro výpočet jejich hodnoty je nutné vypočítat kompletní schéma. Pro práci s výstupními signály se tyto záležitosti neřeší a pouze se pro ně definuje jejich vstupní signál získaný ve schématu a jeho výsledná hodnota. Ukázka *.mdl kódu pro vstupní a výstupní signály je znázorněna na následujících ukázkách kódu. První kód představuje vstupní signál, druhý kód poté výstupní signál.

```
Block {
    BlockType    From
    Name    "FROM_01"
    Description    "Variable FROM block"
    Position    [95, 400, 245, 420]
    ShowName    off
    CloseFcn    "tagdialog Close"
    GotoTag    "output_02"
  }
```

Code 3.2 vstupní signál

```
Block {
    BlockType    Goto
    Name    "Goto_12"
    Position    [580, 350, 730, 370]
    DropShadow    on
    ShowName    off
    FontSize    14
    FontWeight    "bold"
    FontAngle    "italic"
    GotoTag    "output_12"
    TagVisibility    "global"
  }
```

Code 3.3 výstupní signál

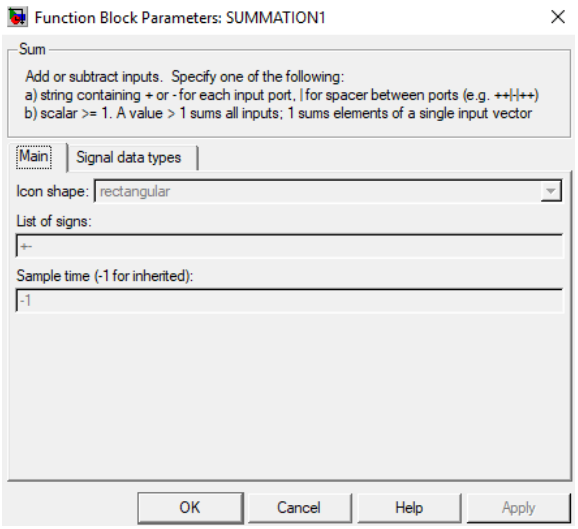
Logické bloky představují hlavní logickou podstatu samotného schématu. Pomocí nich dochází ke správnému vyřešení patřičného problému. Na obr. 3.1, který znázorňuje velmi jednoduché schéma, jsou využity bloky popisující konstantu, porovnávací blok na rovnost, blok na sčítání, logický OR, AND a NOT. Tyto bloky patří mezi základní bloky a jejich implementace je obsažena ve vestavěné knihovně. Pro jejich správné použití je ale nezbytné správně definovat příslušné parametry popisující jejich chování. Ukázka nastavování těchto parametrů je pro sčítací blok zobrazena na obr. 3.2. Pokud je potřeba použít blok, který není v základní ani uživatelské knihovně, je nutné ho celý znovu implementovat v subsystému, což představuje kód na stovky řádků pro každý takový obvod. Ukázka *.mdl kódu pro sčítací blok je znázorněna na následujícím kódu, kde nejprve ukázka předdefinování bloku a poté jeho volání.

```
Block {
   BlockType      Sum
   IconShape      "rectangular"
   Inputs      "++"
   InputSameDT      on
   OutDataTypeMode      "Same as first input"
   OutDataType      "sfix(16)"
   OutScaling      "2^0"
   LockScale      off
   RndMeth      "Floor"
   SaturateOnIntegerOverflow  on
   SampleTime      "-1"
}
```

Code 3.4 předefinování těla bloku

```
  Block {
    BlockType     Sum
    Name     "SUM_1"
    Ports     [2, 1]
    Position     [240, 44, 260, 86]
    ShowName     off
    Inputs     "+-"
  }
```

Code 3.5 volání bloku



Obr. 3.2 nastavení parametrů pro sčítací blok

Spoje mezi jednotlivými signály a bloky představují nezbytnou součást schématu bez kterých nejde správně spustit. Nejčastěji popisují pouze základní cestu mezi dvěma prvky ve schématu, ale v některých případech může být spojení implementované pomocí větvení. Větvení představuje případ, kdy z jednoho výstupního bodu signál putuje do několika výstupních prvků, kterých může být i několik desítek. Postupné větvení a zanořování představuje případný problém při automatickém parsování zdrojového souboru, kdy je třeba na tuto skutečnost dbát při rekurzivním prohledávání všech vnořených větví. Ukázka *.mdl kódu pro základní spoj mezi dvěma bloky je znázorněna na následujícím kódu.

```
Line {
  SrcBlock    "FROM_01"
  SrcPort     1
  Points      [75, 0; 0, -20]
  DstBlock    "OR2"
  DstPort     2
}
```

Code 3.6 spojení dvou bloků

Poslední část v uvedeném schématu představují komentáře. Tyto komentáře mohou být dvojího typu. První typ představuje komentáře číslující pořadí vykonávání jednotlivých bloků ve schématu. Pokud tam číslování není, pak při testování jednotlivých podčástí schématu nerozhoduje v jakém pořadí budou vykonány. Druhou skupinou jsou komentáře u logických bloků. Zde může jít o informaci o netypické změně parametrů logického bloku. Například při definování bloků na zpožďování běhu signálu daným schématem je zde nejčastěji komentář o jak velké zpoždění se jedná, bez nutnosti hledání této informace přímo ve zdrojovém kódu pro dané schéma. Ukázka *.mdl kódu pro komentáře určující pořadí vykonávání částí chématu je znázorněna na následujícím kódu.

```
Annotation {
  Name      "7"
  Position      [965, 625]
  UseDisplayTextAsClickCallback  off
}
```

Code 3.7 komentář

3.2 Parsování

Proces parsování umožňuje analyzovat vstupní soubor pomocí předem dané množiny jasných pravidel, také nazývaných gramatika jazyka. Analyzovaná data jsou poté uložena do interní reprezentace ve formě orientovaného grafu. V podstatě se jedná o proces analyzování znaků ze vstupního souboru na základě gramatické struktury jazyka, ve kterém je vstupní soubor vytvořen. Samotná funkce Parseru se však skládá ze dvou hlavních částí. Jedná se o lexikální a syntaktickou analýzu. [25]

3.3 Lexikální analýza

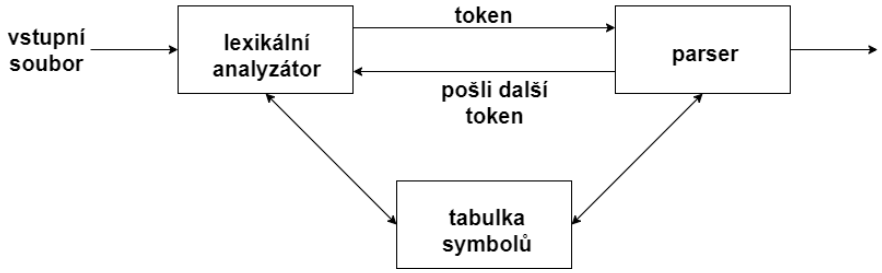
Lexikální analýza pro potřeby této práce slouží k převodu zdrojového textu na postupnou řadu symbolů. Aby mohl lexikální analyzátor správně fungovat, je nezbytné mu nejprve definovat vstupní abecedu symbolů, které se mohou v daném jazyce vyskytovat. Jakmile je abeceda daná, určí se množina symbolů, které se budou v rámci analýzy rozpoznávat. [25] V této fázi je nutné zvolit, zda bude analyzátor rozpoznávat symboly ve formě case-sensitive nebo case-insitive. To znamená, zda bude rozlišovat malá a velká písmena či nikoliv.

Jakmile jsou tyto parametry pro analyzátor stanoveny, nastává jeho hlavní činnost ve které začne převádět vstupní zdrojový soubor na posloupnost symbolů. Tyto výstupní symboly se nazývají tokeny. Tokeny jsou poté na základě konečného automatu vyhodnocovány a následně přeposílány k dalšímu zpracování do syntaktické analýzy. Pro určení jednotlivých stavů konečného automatu slouží tabulka symbolů, která obsahuje seznam všech přijímaných tokenů. Tato tabulka bývá společná jak pro lexikální, tak pro syntaktickou analýzu. [25] Pro potřeby této práce si lze pod parametrem konečného automatu představit klíčové slovo reference, které má v *.mdl souborech roli informovat překladač, že daný blok zde není implementovaný, ale odkazuje se na knihovní funkce.

3.4 Syntaktická analýza

Syntaktická analýza pro potřeby této práce zjišťuje, které symboly neboli tokeny získané z předchozí lexikální analýzy patří k sobě. To je docíleno pomocí ověřování, zda konstrukce vytvářejí gramaticky správné věty či nikoliv. [25] To v případě *.mdl souborů znamená, zda jednotlivé bloky obsahují všechny své parametry.

Bez nich by daný blok nebyl správně naprogramovaný. Jakmile jsou tokeny správně přiřazeny, jsou následně uloženy do interní struktury v podobě derivačního stromu, který si lze představit jako orientovaný graf. Jednotlivé složené tokeny představují kompletní strukturu daného bloku. Celý seznam přijímaných tokenů je sdílen spolu s lexikálním analyzátorem za pomoci tabulky symbolů. [25] Vzájemná komunikace mezi lexikální analýzou a syntaktickou analýzou je znázorněna na obr. 3.3.



Obr. 3.3 schéma parsování vstupního souboru [16]

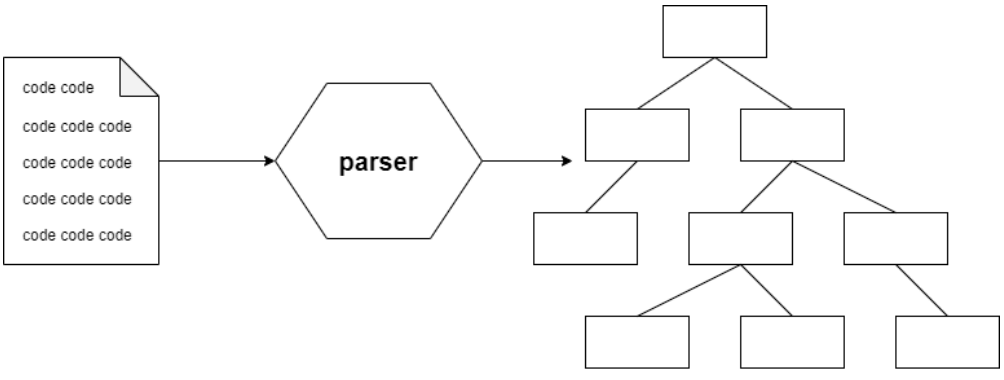
3.5 Parsování *.mdl souboru

Při parsování *.mdl souboru existuje několik možných postupů, jak daný soubor zpracovat, aby bylo možné získaná data využít pro následné zpracování, například při automatizovaném testování. Soubory lze parsovat buď pomocí řady programů vyvinutých pod licencí svobodného softwaru, nebo pomocí sady scriptů, které vytvořila komunita uživatelů a nabídla je k širokému využití. Poslední možností je vytvoření vlastního programu, který bude sloužit k parsování *.mdl souborů.

První kategorii představují programy publikované pod svobodnou softwarovou licencí sloužící mimo jiné k parsování *.mdl souborů. Jeden z takových programů se nazývá ConQAT. Tento nástroj byl poprvé představen v roce 2007 týmem výzkumných pracovníků na technické univerzitě v Mnichově. Část programu ConQAT sloužící k parsování souborů je napsána v jazyce java a je uložena v knihovních fukncích, odkud je možné jej snadno zavolat. Samotný program poté umožňuje zpracovávat nejen soubory s příponou *.mdl, ale i novější typy souborů *.slx využívající vývojové prostředí Simulink.[17]

Druhou kategorií, pomocí níž je možné parsovat *.mdl soubory, jsou uživatelské scripty, které uživatelé Simulinku v minulosti vytvořili, a následné zdrojové kódy svých programů uveřejnili na webové stránce github.com. Nejpropracovanější z těchto scriptů je MDLparsetool.py, vytvořený uživatelem s přezdívkou Steven Yue.[18] Tento script představuje poměrně univerzální nástroj na parsování *.mdl souborů, ale jeho nevýhoda, proč nebyl zvolen pro potřeby této práce spočívá v tom, že nedokáže zpracovávat složitější bloky, které jsou přímo vytvořeny v kódu *.mdl, a nejsou volány z knihovních funkcí. Kromě toho pro potřeby této práce jsou využívány i speciální knihovny pro Simulink, vyvinuté společností Honeywell, se kterými si také neporadí.

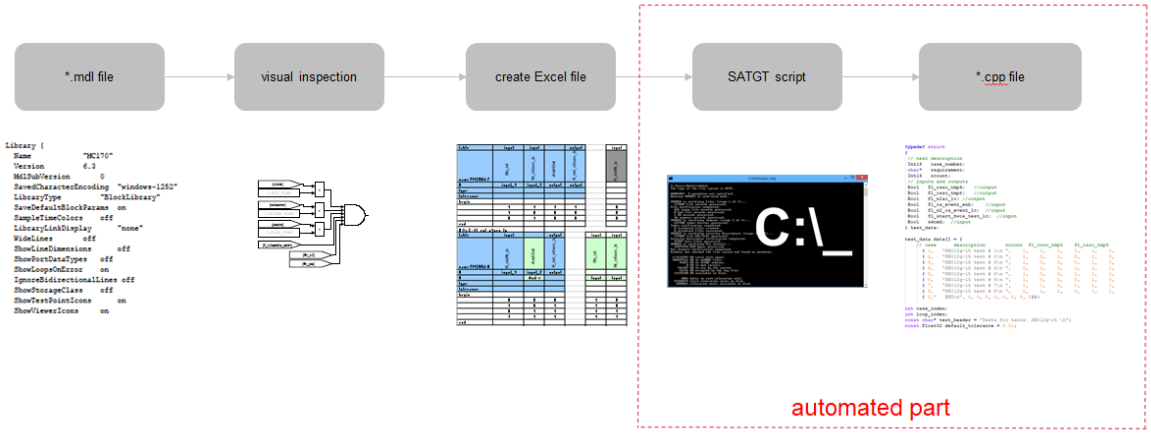
Poslední kategorii představují Parsery, které si uživatel vytvoří sám přímo pro potřeby dané skupiny *.mdl souborů. Tato varianta byla zvolena i v této práci, neboť předchozí zmíněné programy nabízely pouze univerzální řešení, ale nedokázaly zcela pokrýt potřeby testovaných modelů v této práci. Proto bylo rozhodnuto, že časově výhodnější bude provést vlastní implementaci, než zkoumat a rozšiřovat volně dostupné Parsery. K vlastnímu řešení bylo nejprve nutné prozkoumat implementaci všech knihovních bloků využívaných při modelování schémat ve společnosti Honeywell. Na základě získaných znalostí byl vytvořen program na parsování *.mdl souborů, které tyto knihovny využívají. Parser poté transformuje vstupní soubor pomocí sady pravidel a tabulek symbolů až do podoby orientovaného grafu, se kterým je dále pracováno při automatickém testování. Schéma fungování takového Parseru je zobrazeno na obr. 3.4.



Obr. 3.4 schema Parseru využitého v diplomové práci

4 Přehled funkcionality systému

Systém pro testování logických schémat využívaných při programování hradlových polí, který tato práce řeší a který je momentálně využíván v praxi je založen na řadě postupných fází. Tyto fáze se skládají z posloupnosti manuálních kroků, ve kterých je nutná přítomnost testera a sady scriptů, které poté z výsledků jeho práce vytvářejí výsledný test v jazyce C++. Pro názornost je tento postup znázorněn na obr. 4.1, kde je i vyznačena část, která je prováděna automaticky.



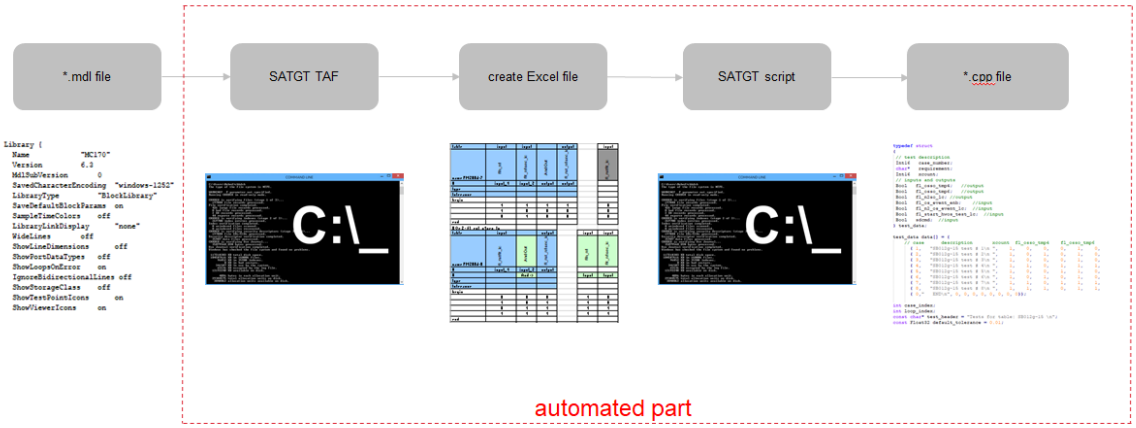
Obr. 4.1 schema současného postupu při testování logického schématu

Práce testera v tomto systému spočívá v tom, že nejprve provede kontrolu správnosti vstupního souboru typu *.mdl. Na základě jeho grafického zobrazení v programu Simulink poté naplánuje proces testování. V tomto procesu na základě certifikovaných testovacích šablon sestaví řadu testovacích scénářů, které musí být splněny pro správné verifikování funkčnosti daného schématu. Z této řady testovacích scénářů jsou poté sestaveny testovací vektory, které jsou uloženy do výstupního souboru typu *.xls. Tento typ souboru je stále zachováván z důvodu zpětné kompatibility systému, který je ve vývoji přes 30 let. Kromě zpětné kompatibility tento typ souborů umožnuje i názorné zobrazení testovacích vektorů, které jsou po testerovi následně zkontrolovány minimálně dvěma dalšími testery, aby bylo zaručeno, že výsledné vektory jsou správné a dle normy DO-178 revize B otestovány. [24]

Jakmile je soubor *.xls vytvořen a řádně zkontrolován, dochází k jeho předání programu na transformaci testových vektorů do testů v jazyce C++, které jsou následně uloženy ve formátu *.cpp. Důvodem, proč tester vytváří testy ve formátu *.xls spočívá také
v tom, že jsou dostatečně přehledné a případné odhalení chyby v testu je zde poměrně jednoduché a časově nenáročné. Pokud by byl vygenerován test přímo ve formátu *.cpp bez mezistupně ve formátu *.xls, pak by bylo odhalování chyby značně kompli-

kované. Testy v C++ mohou být velmi rozsáhlé v závislosti na testovaném schématu. To v praxi znamená až tisíce řádků obsahujících desítky datových struktur plných testovacích vektorů. Nalezení a následná oprava chyby v takovém systému je značně časově náročná.

Tato diplomová práce si klade za cíl zefektivnit proces vytváření testovacích vektorů ve formátu *.xls bez nutnosti zásahu testera. Tento proces bude spočívat v automatickém vyhodnocování vstupních schémat ve formátu *.mdl a jejich následném zpracování podle sady předem stanovených pravidel a testovacích šablon do výstupního souboru *.xls. Navrhovaný postup bude odpovídat tomu, jak by postupoval tester, pokud by dané schéma testovat manuálně. Jakmile budou vytvořeny testovací vektory v souboru *.xls, dojde k zavolání programu na transformování těchto dat do vstupního kódu v jazyce C++. Pro názornost je výsledný postup při testování znázorněn na obr. 4.2.



Obr. 4.2 požadované schéma posloupnosti testování logického schématu

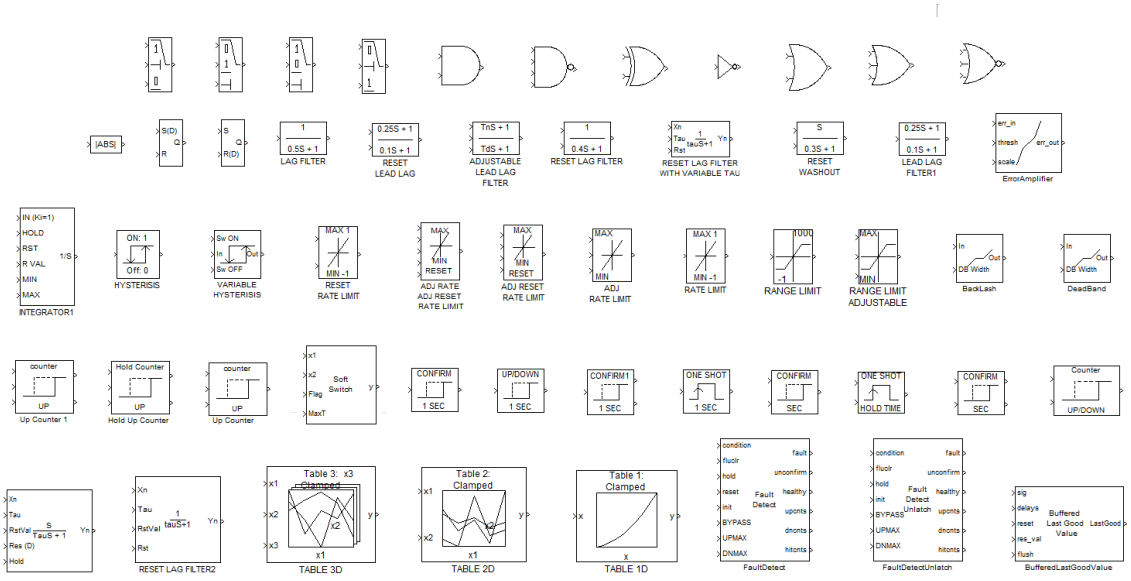
4.1 Vstupní soubor

Vstupní data, která budou do programu přiváděna, jsou uložena v souborech *.mdl. Charakteristika tohoto datového typu byla stručně popsána v kapitole 3. Nejdůležitější část vstupního souboru představují logické bloky, které se dají rozdělit na tři typy. Ty je třeba správně identifikovat a následně správně otestovat. Bloky mohou být popsány jako základní logické bloky, komplikovanější logické bloky nebo jako složité logické bloky.

4.2 Základní logické bloky

V této kategorii se nacházejí bloky, které se ve schématech používají nejčastěji, a jejichž implementace je provedena v knihovních funkcích. Soubor *.mdl obsahuje pouze základní informace o struktuře těchto bloků a řadu parametrů, které určují jejich název,

pozici ve schématu, kolik mají vstupních a kolik výstupních portů, ale hlavně odkaz, ve které knihovně jsou implementovány. Do této kategorie patří bloky AND, OR, NOT, XOR, filtry, časově proměnlivé bloky, přepínací bloky a mnohé další. Kompletní seznam všech zástupců bloků z této kategorie je uveden na obr. 4.3



Obr. 4.3 seznam zástupců základních logických bloků

Kromě řady parametrů, které se povinně nacházejí v kódu každého bloku, se zde nachází i řada dobrovolných parametrů, které jsou dvojího typu. První typ představují parametry, které mohou být definovány v části BlockDefault popsané v kapitole 3. Druhým typem jsou takové, které nesou extra informace o daném bloku, ale pro správné testování daného bloku nejsou potřeba. Informace o těchto dobrovolných parametrech daného bloku nejsou aplikovatelné pouze na základní logické bloky, ale i na komplikovanější a složité logické bloky popsané v sekcích 4.3 a 4.4. Pro názornou ukázku o struktuře základního logického bloku typu hystereze je zde uvedena jeho implementace v souboru *.mdl bez volitelných parametrů a s řadou volitelných parametrů.

```
Block {
    BlockType:  Reference
    Name:  HYSTERESIS
    Ports:  1, 1
    Position    900, 194, 955, 256
    SourceBlock  EnginesLib/AES Symbol\nLibrary/HYSTERESIS
}
```

Code 4.1 blok hystereze bez volitelných parametrů

```
Block {
    BlockType:  Reference
    SourceType  Hysterisis1
    Name:  HYSTERESIS
```

```
    OnSwitchValue  on
    Ports:  1, 1
    OffSwitchValue  off
    OnOutputValue  1
    OffOutputValue  0
    OutputDataTypeScalingMode   All ports same datatype"
    Position    900, 194, 955, 256
    OutDataType  sfix(16)
    OutScaling  2^0
    ConRadixGroup  Use specified scaling
    SourceBlock  EnginesLib/AES Symbol\nLibrary/HYSTERESIS
    on  0.99
    off  0.01
}
```

Code 4.1 blok hystereze s volitelnými parametry

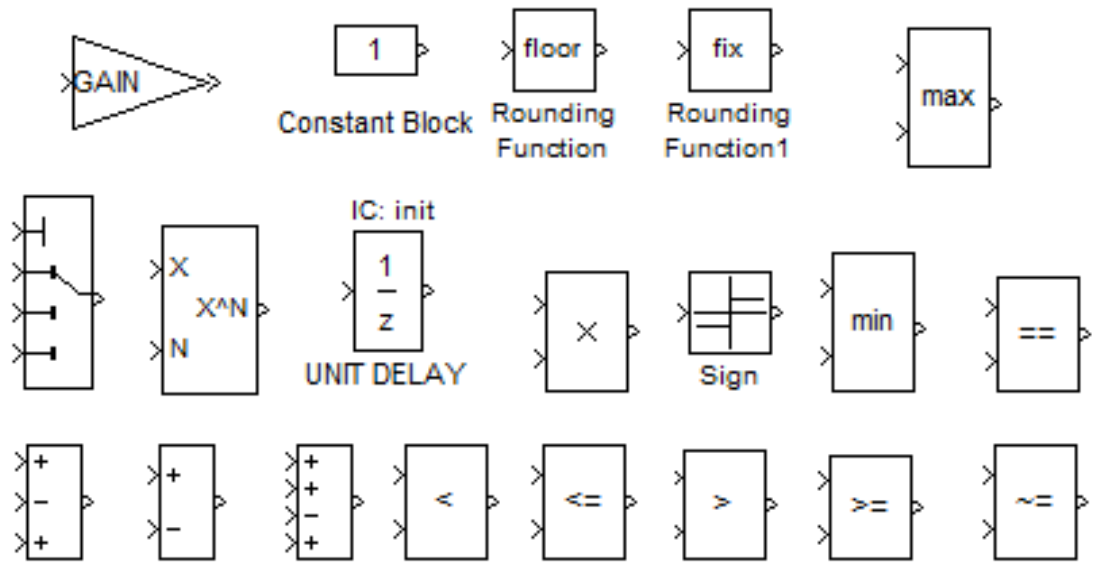
Ukázka implementace bloku hystereze na předešlých příkladech ukazuje, že oba dva případy mají stejnou vypovídající hodnotu v grafickém zobrazení v programu Simulink, ale zcela odlišnou implementaci v *.mdl. Tyto parametry je nutné vzít v úvahu, a při automatickém testování softwaru pomocí této práce s touto skutečností počítat.

4.3 Komplikovanější logické bloky

Komplikovanější bloky se narozdíl od základních logických bloků neukazují na knihovní funkce, ale jsou v kódu plně implementovány. To znamená, že oproti základním blokům se nezapisují klíčovým slovem reference, ale vždy názvem daného bloku. Do této kategorie patří bloky typu signálového zisku, konstantní bloky, porovnávací bloky, násobící a sčítací bloky, zpožďující bloky a mnohé další. Seznam všechy zástupců bloků z této kategorie je uveden na obr. 4.4

Bloky této kategorie mají složitější vnitřní strukturu v kódu *.mdl, a obdobně jako základní bloky obsahují řadu povinných parametrů a řadu dobrovolných. Narozdíl od základních bloků jsou zde povinné parametry rozděleny na dvě podčásti. První podčást je obsažena v části kódu začínajícího BlockDefault a její zpracování je nezbytně nutné pro správné zpracování chování daného bloku. Na následující ukázce části kódu pro blok suma je znázorněno rozdělení povinných parametrů na dvě podčásti. První ukázka kódu znázorňuje sekci BlockDefault a poté druhá ukázka samotné použití v rámci chování schématu.

```
 Block {
   BlockType     Sum
   IconShape     "rectangular"
   Inputs     "++"
   InputSameDT     on
   OutDataTypeMode     "Same as first input"
   OutDataType     "sfix(16)"
   OutScaling     "2^0"
   LockScale     off
   RndMeth     "Floor"
   SaturateOnIntegerOverflow  on
   SampleTime     "-1"
}
```



Obr. 4.4 seznam zástupců komplikovanějších logických bloků

Code 4.3 definování bloku na sčítání

```
Block {
    BlockType    Sum
    Name    "SUM_10"
    Ports    [3, 1]
    Position    [24, 445, 160, 86]
    ShowName    off
    Inputs    "+++"
  }
```

Code 4.4 volání bloku na sčítání

4.4 Složité logické bloky

Poslední kategorií vstupních bloků, se kterými se lze setkat, jsou složité logické bloky. Ty jsou svým využitím v rámci schématu značne specifické, a z toho důvodu pro ně nejsou vytvořeny knihovní funkce. Místo toho dochází k podrobné implementaci v místě výskytu daného bloku. Tyto implementace jsou prováděny v hlavním systému formou subsystémů. Subsystém představuje rozsáhlejší blok, ve kterém je popsána celá jeho struktura. Do této kategorie patří speciální případy klopných obvodů, bloky na zachytávání chybných paritních bitů a mnohé další. Seznam všechy zástupců bloků z této kategorie je uveden na obr. 4.5.

Bloky nacházející se v této kategorii jsou svou implementací považovány za nejkomplikovanější. Zdrojový kód takového bloku se nenachází na několika řádcích, ale na několik desítkách až stovkách řádků kódu. I přes značnou rozsáhlost je ke správnému

fungování daného bloku potřeba zjistit jen řada nezbytných informací. Mezi tyto informace patří typ bloku, název bloku, počet vstupních a výstupních signálů a rozhraní daného bloku. Na následující ukázce části kódu pro Status Word lze vidět část definující základní informace o daném bloku a část o samotné implementaci. Kompletní kód se rozkládá na zhruba tisíc řádků a z toho důvodu je zde uvedena jen malá část popisující deklaraci hlavního chování a následné vnitřní struktury.

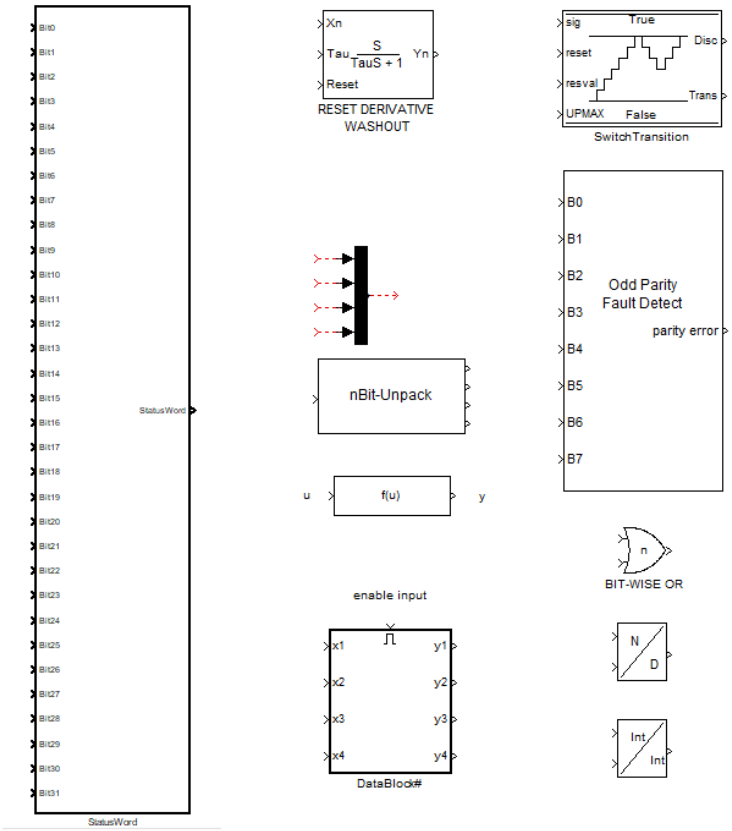
```
Block {
    BlockType    SubSystem
    Name         "StatusWord_01"
    Ports        [32, 1]
    Position     [300, 50, 800, 600]
    FontSize     8
    TreatAsAtomicUnit    off
    MinAlgLoopOccurrences    off
    ...
        Block {
            BlockType     Inport
            Name     "Bit_0"
            Position     [10, 20, 30, 40]
            IconDisplay        "P_Num"
        }
        Block {
            BlockType     Inport
            Name     "Bit_1"
            Position     [20, 40, 60, 80]
            Port     "P_NUM_01,,
        ...
}
```

Code 4.5 část kodu pro Status Word

4.5 Výstupní soubor

Výstupní soubor obsahuje sadu testovacích vektorů, pomocí nichž lze plně prokázat správnost testovaného schématu. Testovací vektory nejsou zprvu psány přímo do jazyka C++ z důvodu značné rozsáhlosti a možnému zdroji chyb při implementaci takového testu. Místo toho jsou testovací vektory vytvářeny pomocí programu Excel a ukládány ve formátu *.xls. Jednotlivé testovací scénáře, které musí tyto testové vektory splňovat, odpovídají doporučení vycházejících z dokumentu DO-178 revize B [24], který v leteckém průmyslu slouží jako sada doporučení, podle kterých lze správně otestovat vyvíjený systém, aby jej bylo možné následně certifikovat. Jakmile jsou testovací vektory vytvořeny a uloženy v souboru *.xls, nastává jejich automatická konverze pomocí certifikovaného programu do jazyka C++.

Za správnou tvorbu výstupního souboru je zodpovědný tester. Ten za využití svých znalostí a sady certifikovaných šablon pro testování jednotlivých logických bloků provede správnou tvorbu testovacích scénářů. Šablony, které jsou při vývoji využívány, se dělí do dvou kategorií. První kategorie představuje časově neměnné bloky, pro něž je testování poměrně snadné a stačí využívat předpřipravených šablon. Druhou skupinu představují bloky časově proměnlivé. Při práci s touto skupinou bloků je nutné správně



Obr. 4.5 seznam zástupců složitých logických bloků

zohledňovat případy, ve kterých nastává například časové zpoždění či předbíhání systému.

Prvním krokem při testování je tedy správná identifikace jednotlivých logických bloků a určení správného proudění signálů přes jednotlivé bloky v systému. Jakmile jsou všechny bloky správně identifikovány nastává část, ve které je nutné vytvoření testovacího scénáře pomocí testových vektorů, které dokáží plně otestovat daný blok. Tato část je tou nejdůležitější v rámci celého vývoje. Jednotlivé typy šablon a kompletní testovací scénář budou podrobněji rozebrány v následujících podkapitolách.

4.5.1 Časově neměnné bloky

Ke správnému testování logického schématu je nutné využívat adekvátní testovací vektory. První testovanou skupinu bloků lze souhrně označit jako časově neměnné bloky. Pro bloky této kategorie je typické, že jejich správné testování vychází z pomocných certifikovaných šablon obsahujících odpovídající testovací vektory a jejichž změny nejsou ve větší míře vyžadovány. Mezi zástupce této kategorie lze řadit bloky AND, OR, XOR, NOT, násobičky, sčítačky, klopné obvody, děličky a další.

Jednotlivé šablony obsahují sadu povinných parametrů, které jsou pro všechny testovací vektory totožné. Jedná se především o název testovaného bloku, jména, případně i typy vstupních a výstupních signálů. Typem signálu se rozumí případ, zda se jedná o konstantní nebo proměnlivý signál. Jakmile je blok správně identifikován a je pro něj správně vybrána testovací šablona, dochází k jejímu vložení do testovacího scénáře. Ukázka testovacího vektoru pro násobící blok je zobrazena na obr. 4.6

table	input	input	output	Comments
name Test-1	INPUT_01	INPUT_02	OUTPUT_01	
#	Input 1	Input 2	output	
min	-80	26		
max	10	200		
type				
tolerance				
begin				
	10	200	2000	MaxOUT
	-80	200	-16000	MinOUT
	10	0	0	
	0	200	0	
end				

Obr. 4.6 testovací vektory pro ověření násobičky

V některých případech může nastat situace, kdy šablona zcela neodpovídá řešenému schématu a je třeba ji dodatečně upravit. Takovým případem může být situace pro testování logického bloku sčítačky, která nemá pouze dva vstupy pro které je vzorová šablona vytvořena, ale například dvacet vstupů. V takovém případě je nutný zásah testera, který odborně šablonu upraví, aby odpovídala testovanému bloku.

4.5.2 Časově proměnlivé bloky

Druhou skupinu představují časově proměnlivé bloky. U této skupiny není možné pouhé aplikování předpřipravené certifikované šablony. Vytváření testovacích vektorů probíhá aplikováním šablony a následuje její úpravou, aby odpovídala aktuálně testované situaci. Bloky patřící do této kategorie jsou například filtry, zpožďovací či předbíhající bloky, případně přepínače závislé na čase.

Parametry testovacích vektorů obdobně jako u časově neměnných bloků vycházejí ze samotné testovací šablony. Základní parametry jako jméno bloku, nebo i typ vstupních a výstupních signálů, zůstávají nezměněny. Odlišným parametrem je počet kroků, které jsou v rámci jednoho testovacího scénáře vykonány. Například pokud se pracuje s blokem, který je ve schématu zpožďování a je potřeba získat určitou specifickou hodnotu, aby byl daný blok plně pokryt a zkontrolován, může dojít k přidání nových

testovacích vektorů pro daný blok. Ukázka časově závislého bloku, pro který byla při dalším využití šablona rozšířená, aby plně otestovala daný blok je uvedena na obr. 4.7.

table	input	output		output
name Test-2	INPUT_01	CONSTANT	Timer	OUTPUT_01
#	input	time limit		output
min		1		
max				
type				
tolerance				
begin				
	0	1	0	0
execute 99	1	1	0.99	0
	1	1	1	1
	1	1	1	1
	0	1	0	0
end				

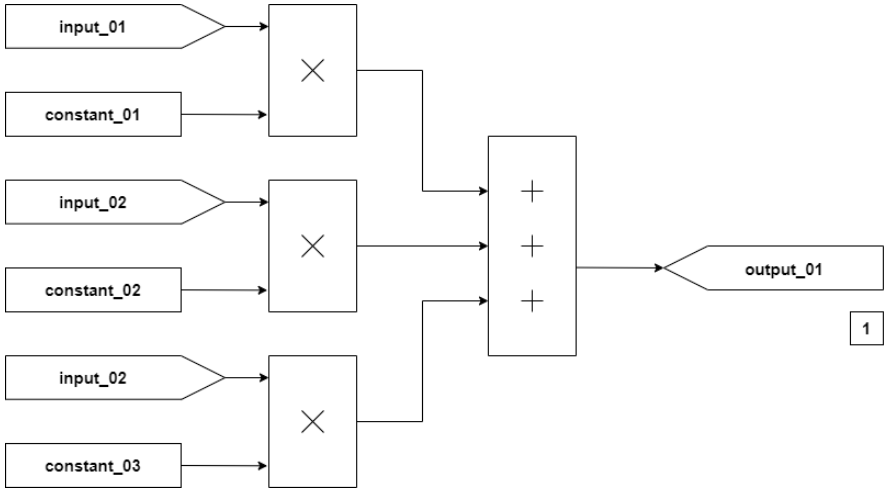
Obr. 4.7 testovací blok závislý na čase

4.5.3 Testovací scénář pro část schématu

Testování jednotlivých bloků, kde by signál ze vstupu rovnou vycházel na výstup daného logického schématu, je velmi málo. Častěji nastává situace, kdy je nezbytné v rámci jednoho testovacího scénáře otestovat více bloků najednou. Například při zpracování schématu znázorněného na obr. 4.8 je pro ověření správnosti výstupního signálu označeného output01 nezbytně nutné ověřit správné fungování logického programovatelného bloku sčítačky, a také zda mu všechny násobičky správně předávají vstupní data.

Popsaný postup je nezbytné provést nejen pro testování bloku sčítačky, ale také pro každou jednotlivou násobičku, aby bylo možné ověřit, že ve všech případech svého deklarovaného chování nenastane problém a že její výstupní hodnoty jsou správné a následující blok s nimi umí bez problému pracovat. Typicky je takové chování ověřeno testováním na krajních hodnotách přípustného rozsahu daného bloku společně s náhodnými hodnotami v tomto definovaném rozsahu.

Výsledný testovací scénář obsahuje více bloků, které na sebe navzájem navazují. Tyto bloky vycházejí z testovacích šablon, ale pouze pro aktuálně testovaný blok je šablona využita plně a pro ostatní je využita jejich zkrácená verze. Každá zkrácená verze ale musí mít zachovanou logickou podstatu daného bloku, aby bylo zajištěno, že dané schéma bude daným testovacím scénářem plně pokryto. Ukázka rozsáhlejšího testovacího scénáře pro schéma z obr. 4.8 je uvedeno na obr. 4.9.



Obr. 4.8 schéma popisující jednocuhý logický obvod

table				output		input	constant	input	constant	input	constant
name Test-11	MUX_01	MUX_02	MUX_03	OUTPUT_01		INPUT_01	CONST_01	INPUT_02	CONST_02	INPUT_03	CONST_03
#	+	+	+	output							
#	Mux_01-->	Mux_02-->	Mux_03-->			Input 1	Input 2	Input 1	Input 2	Input 1	Input 2
min	-50	-200	-25	-250							
max	0	0	0	0							
type											
tolerance											
begin											
	0	0	0	0	MaxOUT	1	0	1	0	2	0
	-25	-50	-200	-275	MinOUT	1	-25	1	-50	2	-100
	-0.0002	-0.0004	-0.0016	-0.0022		1	-0.0002	1	-0.0004	2	-0.0008
	-0.0004	-0.0008	-0.0032	-0.0044		1	-0.0004	1	-0.0008	2	-0.0016
	-5	-10	-40	-55		1	-5	1	-10	2	-20
end											

Obr. 4.9 ukázka testovacích vektrorů pro sčítačku

4.5.4 Spustitelný test

Jakmile je sada testovacích scénářů připravena a odpovídá testovanému scénáři, dochází na poslední fázi vytváření testu. V této fázi se využívá speciální program zvaný transformSTG script vyvinutý společností Honeywell, který převede testovací vektory vytvořené v souboru *.xls do spustitelného testu v jazyce C++. Spustitelný test obsahuje sadu datových struktur, ve kterých jsou obsaženy jednotlivé testovací scénáře. Všem scénářům byla zachována jejich vypovídací hodnota, pouze došlo k jejich přepisu do jiné podoby. Ukázka výsledného testu obsahujícího testovací scénář přepsaného do jazyka C++ je možné vidět na následující ukázce kódu.

```
 test_vektors_data data[] = {
 // Info output_01 input_1 input_2 input_3 const_1 const_2 const_3
{ 1,  "Test-01 test # 1\n ",0,1,1,2,0,0,0},
{ 2,  "Test-01 test # 2\n ",-275,1,1,2,-25,50,-100},
{ 3,  "Test-01 test # 3\n ",-0.0022,1,1,2,-0.0002,-0.0004,-0.0008},
{ 4,  "Test-01 test # 4\n ",-0.0044,1,1,2,-0.0004,-0.0008,-0.0016},
{ 5,  "Test-01 test # 5\n ",-55,1,1,2,-5,-10,-20},
{ 0,"  END\n", 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}};
```

Code 4.6 ukázka výsledného testu

Na předešlé ukázce je možné podrobně vidět, jakým způsobem probíhá testování logického bloku sčítačky, který je ve schématu propojen se třemi logickými bloky v podobě násobiček. Samotné násobičky se ve vygenerovaném testu nenacházejí, nacházejí se zde pouze jejich vstupní hodnoty.

II. PRAKTICKÁ ČÁST

5 Architektura vyvíjeného systému

Cílem této kapitoly je navrhnou kompletní strukturu vyvíjeného systému pro automatické testování logických programovatelných bloků. V úvodní části je nezbytné správné zvolení programovacího jazyka, ve kterém bude celý program vyvíjen. V následujících podkapitolách dojde k definování požadavků na chování vyvíjené aplikace a jejímu rozdělení na jednotlivé modulární podčásti ze kterých se bude skládat. Tyto jednotlivé moduly budou detailně popsány a rozebrány z pohledu jejich vyžadovaného chování. V poslední části dojde k definování potřebných knihoven, které je vhodné při vývoji využít a určení řady validačních testů, které je nezbytné vykonat ke správnému ověření chování testovacího systému.

5.1 Programovací jazyk Python

Vývoj programovacího jazyka Python započal koncem devadesátých let dvacátého století a první verze byla vydána v roce 1991 s označením 0.9. Za vývojem jazyka stojí programátor Guido van Rossum, který v té době pracoval pro Národní výzkumný ústav pro informatiku a matematiku sídlící v Amsterdamu. Jazyk Python vychází z jazyka ABC a původně měl sloužit pro snadnou práci s operačním systémem Amoeba. Samotný název jazyk získal podle britského komediálního pořadu Monty Python's Flying Circus. [19]

V roce 1995 po vydání Python verze 1.2 přestoupil Guido van Rossum na nové pracoviště CNRI (Corporation for National Research Initiatives) sídlící v USA, kde i nadále pokračoval ve vývoji jazyka. Díky tomu všechny následující verze jazyka jsou již vydaný pod licencí CNRI. Zde ale nastal problém v tom, že licence CNRI není kompatibilní s GNU (General Public License) pod kterou Guido van Rossum chtěl Python vydat. Z toho důvodu využil nabídky spolupráce ze strany FSF (Free Software Foundation) a společně vytvořili Python Software Foundation Licenci, pod kterou bude Python nadále vydáván a která je plně kompatibilní s GNU. [19] Díky tomu je možné jazyk Python zdarma používat, rozšiřovat a nabízet vytvořený software i pro komerční účely.

Na přelomu tisíciletí došlo k poslednímu výraznému milníku ve vývoji jazyka. V roce 2000 byla založena společnost ByOpen, která vydala Python ve verzi 2.0. Nová verze jazyka představila řadu novinek, ale za největší lze považovat Garbage collector pro snadnější správu paměti. V roce 2008, kdy došlo k vydání verze 3.0, nastala poměrně nepříjemná situace. Nová verze jazyka již nadále nepodporovala zpětnou kompatibilitu zdrojového kódu. Jinými slovy zdrojový kód vytvořený ve verzi dva již nebylo možné zkompilovat v novější verzi. Vzhledem k tomu, že řada rozsáhlých systémů již byla v původní verzi vytvořena a její transformace do nové verze by byla poměrně nákladná, jsou v současné době dvě používané verze. Jedná se o verzi 2.7 a 3.2. [19] Pro potřeby

této práce, z důvodu historického vývoje systému, se kterým je nutné, aby byla práce kompatibilní, je využita verze jazyka Python 2.7.

5.2 Python verze 2.7

Programovací jazyk Python ve verzi 2.7 představuje multiplatformní, interpretovaný, vysokoúrovňový, objektově orientovaný scriptovací jazyk, který je ve srovnání s řadou klasických jazyků, jako například Java či C++, více vhodnější pro rychle prototypování a psaní scriptů. Díky svým objektovým schopnostem se může porovnávat i se zmíněnými jazyky. Svou strukturou je často přirovnáván k jazyku Perl. V základní konfiguraci je Pytoh dodáván i se standardní knihovnou, která nabízí širokou nabídku vhodných funkcí. Kromě toho je na trhu celá řada knihoven třetích stran, a to jak opensource, tak komerčních knihoven, které se dají při vývoji softwaru využívat. Této možnosti jde nejvíce využít při vývoji softwaru s grafickým uživatelským rozhraním. Při využití základních nástrojů a dodávané knihovny to jazyk Python neumožňuje, ale po přidání knihoven třetích stran je to možné. [20] Vytváření výsledného zdrojového kódu není vázané na žádný vývojový editor a je možné jej vytvořit v libovolném textovém editoru. Díky tomu je jazyk Python velmi multiplatformní. Zdrojové kódy je možné vytvářet a poté spouštět na celé řade operačních systému jako jsou MS Windows, Linux, Mac OS, Android a mnohých dalších. [21]

5.2.1 Charakteristika jazyka Python:

Práce s pamětí je jazyce Python řešena automaticky pomocí garbage collectoru obdobně jako u jazyka Java. Programátor díky tomu nemusí hlídat, zda došlo ke správnému uvolnění paměti v situacích, kdy již není využívána, jako je tomu u jazyka C případně C++. Python si vede seznam referencí na jednotlivé objekty a hlídá počty odkazů směřující na ně. [21] Jakmile počet odkazů na daný objekt klesne na nulu, dochází k uvolnění paměti, kterou daný objekt zabírá.

Silné dynamické typování umožňuje vykonávání sady příkazů za běhu systému, narozdíl od statického typování, kde je vykonání sady příkazů možné pouze v době kompilace. Dynamické příkazy nevyžadují deklarování datových typů pro své proměnné, neboť ty jsou v jazyce Python přímo odvozeny od hodnoty proměnné, které jsou pro jednotlivé příkazy definovány. Díky tomu mohou nastat případy, kdy jedna proměnná může nést hodnoty různých datových typů. Nicméně díky silnému dynamické typování nemůže nastat situace, kdy jednotlivá proměnná bude měnit svůj datový typ v průběhu běhu programu, kromě případů kdy programátor provede přetypování proměnné. [19]

Díky ošetření silné typovosti není možné potom bezproblémově sčítat dva různé datové typy.

Multiplatformní scriptování je velkou výhodou jazyka Python, díky níž je možné implementování nejen složitých a rozsáhlých projektů, ale také vytvoření řady menších scriptů, které mají široké využití. Využití může být jak na webových serverech k zautomatizování menších úkonů, či k usnadnění práce na klasickém počítači ve vybraném operačním systému. To vše je možné díky multiplatformní podstatě jazyka, která umožňuje snadnou přenositelnost napříč operačními systémy.[21]

Objektová orientace v případě Pythonu znamená, že při vývoji je možné využít řady procedurálních, funkcionálních a objektových paradigmat. To znamená, že při práci je kladen velký důraz na tvorbu objektů, které obsahují jak samotná data, tak i funkce, případně metody. Při práci s jazykem je nutné vzít v potaz, že Python na skoro všechny své logické konstrukce nahlíží jako na objekty. Pod pojmem objekt si lze představit sadu proměnných a funkcí, které s danými daty pracují. Následným prototypem objektů jsou třídy, z nichž jsou všechny objekty odvozeny formou instancí. [19]

5.3 Požadavky na funkcionalitu systému

Jak již bylo popsáno v kapitole 4, od systému je vyžadováno, aby byl vytvořen pomocí programovacího jazyka Python. Tato potřeba je odůvodněna tím, že ostatní komponenty, se kterými program bude komunikovat jsou již vytvořeny v tomto jazyce. K těmto komponentám bude program buď přímo předávat výsledky daného testování logického bloku, nebo je naopak bude volat a využívat jejich funkcí. Komponenty, které budou takto volány, nejsou vytvořené formou knihovních funkcí, ale formou modulů, které je možné volat a jsou schopny plně spolupracovat s našim programem.

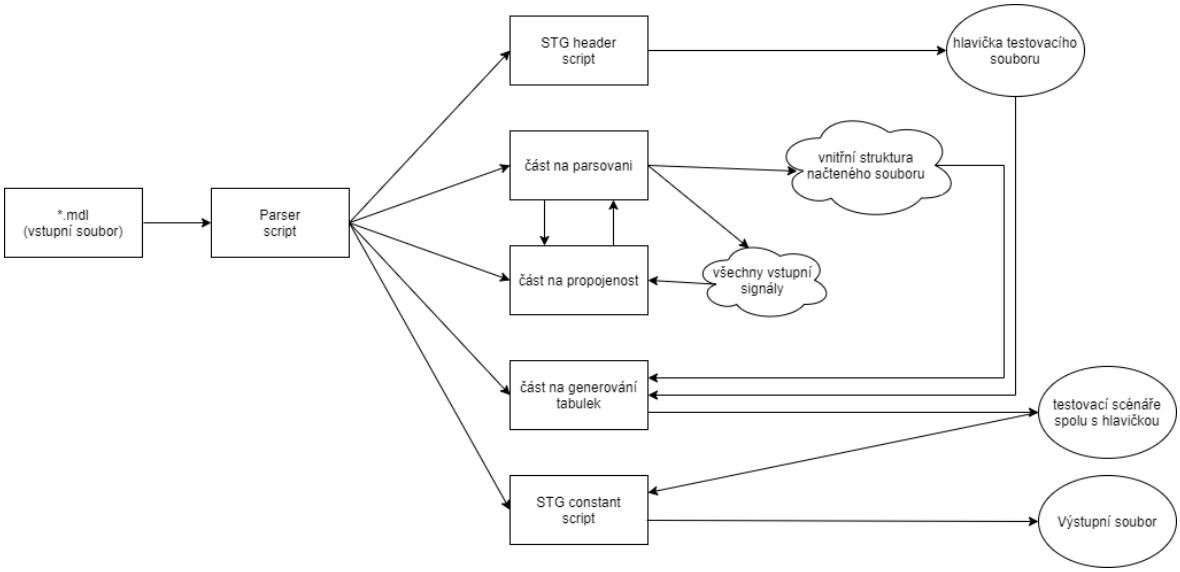
Platforma pro kterou bude systém vyvíjen, je MS Windows ve verzích XP, Windows 7 a Windows 10. Spustitelnost programu na všech platformách je zajištěna univerzálností jazyka Python a jeho multiplatformních možností využití. Mezi uživatele výsledné aplikace budou patřit jak samotní testeři, jejichž úkolem je ověřit správnost daného schématu, tak i vývojáři, kterým umožní rychlou kontrolu, že ve vyvíjeném systému neudělali neočekávanou chybu. Od systému není vyžadováno, aby plně nahradil práci testera, ale aby mu nabídl další nástroj k testování, který při své práci může využít.

Grafické uživatelské rozhraní vyvíjeného programu není vyžadováno, neboť v tomto případě by to aplikaci zbytečně zkomplikovalo a navíc je očekáváno pouze konzolové řešení. Parametry volání programu jsou očekávány v podobě zvolení souboru se sché-

matem, který je třeba otestovat a definováním jména souboru, jak se má výsledná sada testovacích vektorů jmenovat.

Při vývoji se neočekává využití placených komerčně dostupných knihoven třetích stran. Využití takových knihoven není přímo vyloučeno, ale je povoleno pouze v nezbytných případech, kdy daná funkcionalita nejde jinak docílit. Zmíněné omezení je zavedeno
z toho důvodu, že jakékoliv komerční využití takové knihovny musí být schváleno společností Honeywell a musí být prokázáno, že není jiné možnosti, než využít placené knihovny. Z toho důvodu například parsovací modul v úvodní části programu nevyužívá knihovních funkcí, ale je zde plně naprogramován.

Od vyvíjené testovací aplikace se očekává, že její struktura bude složená z řady dílčích podčástí. Tyto podčásti budou jak ve formě knihoven, které budou v aplikaci volány a následně využívány, tak i ve formě modulů. Od jednotlivých modulů je vyžadována určitá funkčnost, která je zobrazena na obr. 5.1. Zde je názorně vidět, že ke správnému fungování systému je nezbytné mít modul na sestavení hlavičky pro náš test, modul na parsování *.mdl kódu, modul na zjištění návaznosti daného schématu obsaženém v *mdl kódu vůči ostatním schématům, modul který generuje sadu testovacích vektorů, a také modul na speciální testování konstant nacházejících se v daném schématu.



Obr. 5.1 rozdělení vyvíjené aplikace na dílčí podčásti

Požadavky na jednotlivé moduly vycházejí z interních testovacích předpisů společnosti Honeywell a řídí se doporučením ze směrnice DO-178B. [24] Tyto předpisy a doporučení popisují postup při kterém dochází k testování logického schématu zleva doprava, shora dolů, není-li to v samotném schématu vyžadováno jinak, například číslováním pořadí logických bloků. Kromě toho je využíváno přesné dodržování testovacích šablon, jak bylo popsáno v kapitole 4. Pokud jsou dané postupy dodrženy, je možné

z daných testovacích vektorů vygenerovat výsledný test. Detailnější požadavky na funkčnost jednotlivých modelů budou popsány v následující podkapitole 5.4.

Pro lepší představu o adresářové struktuře vyvíjeného systému je zde znázorněna očekávaná adresářová struktura výsledného systému. Uvedeno je rozdělení aplikace na sadu jednotlivých scriptů a vlastních knihoven, které ke svému běhu využívají.

- Headerscript.py
- TAF.py
- Constantscript.py
- Transform.bat
- KNIHOVNY
  - template.py
- TESTY
  - sada pro Parser
  - sada finálních testů

5.4 Popis chování dílčích podčástí

Rozdělení systému na dílčí podčásti je nezbytným krokem při vývoji systému pro automatické testování logických schémat. Ideální přístup k řešení problému spočívá v oddělení načítací a parsovací části aplikace od zpracování a výsledného generování testovacích vektorů. Není nezbytně vyžadováno oddělení jednotlivých podčástí do samostatných souborů, ale pro zvýšení přehlednosti je takový krok doporučován. Ve vyvíjené aplikaci bude jeden řídící skritp, který je na obr. 5.1 identifikován jako init script. Ten má na starost jednotlivé volání případných podčástí a zajištění, že aplikace bude správně komunikovat.

Pravidla pro práci s jednotlivými moduly jsou jasně stanovená a jejich podrobnější popis je rozebrán v následujících kapitolách. V těchto podkapitolách je uveden seznam modulů, které jsou pro vyvíjený systém nezbytné a z toho důvodu je nutná jejich specifikace požadavků.

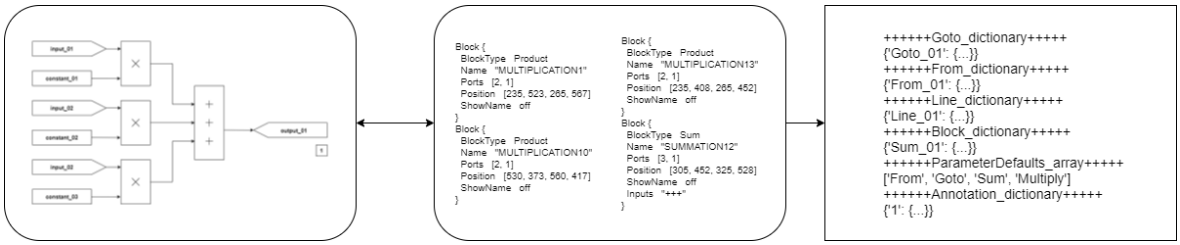
5.4.1 Parser

První důležitou podčástí vyvíjené aplikace je samotné načtení a zpracování vstupního *.mdl souboru obsahujícího zdrojový kód logického programovatelného schématu. Od Parseru je vyžadováno, aby byl schopen postupně procházet načtený soubor po

řádcích a získával nezbytné informace, které jsou důležité pro další podčásti vyvíjené aplikace. Jedná se o jednoznačné jméno testovaného schématu, seznam jeho vstupních a výstupních signálů, kompletní seznam všech vzájemných propojovacích kabelů v daném schématu, seznam všech logických bloků a případně všechny komentáře, nacházející se v daném vstupním souboru.

Ke správnému parsování vstupního souboru je nezbytná znalost všech logických bloků, které se mohou v daném souboru nacházet. Tuto informaci získáme z prostudování využívaných knihoven a to jak vestavěné Simulink.mdl, tak i specifické vyvinuté společností Honeywell s názvem EnginesLib.mdl. Prostudováním zmíněných knihoven je možné Parser připravit na všechny možné typy logických bloků, které vývojář může při svém vývoji využívat. Toho je docíleno vytvořením šablony pro každý logický blok, který může být použit a při parsování je nutné takový blok identifikovat a vhodně uložit do připravené proměnné.

Účel Parseru bude tedy spočívat v načítání vstupního souboru, který bude podle předem dané sady konečných pravidel rozkouskován a získaná data budou dále uložena buď do připravených proměnných například v případě názvu modelu, nebo do adekvátních datových struktur v případě načtení vstupních bloků. Pro větší představu je základní struktura Parseru naznačena na obr. 5.2.

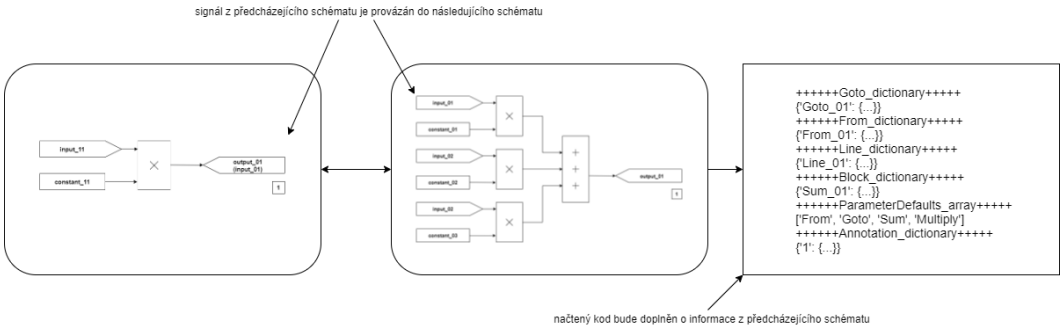


Obr. 5.2 základní podstata funkčnosti Parseru

5.4.2 Zkoumač návaznosti

Účelem této podčásti je zkoumání vzájemné návaznosti testovaného schématu vůči ostatním schématům. Tento krok je nezbytný z toho důvodu, že kompletní testovací schéma logického obvodu je velmi rozsáhlé a obsahuje stovky až tisíce logických bloků. Proto došlo k jeho rozdělení na dílčí podčásti, které jsou ve většině případů naprosto soběstačné a dají se simulovat bez potřeby jejich propojení se schématy z předchozí částí. Tato skutečnost je docílena pomocí přepisovatelnosti hodnot signálů na výstupech z logických bloků, které by následně vstupovaly do některého logického bloku v námi testovaném schématu. To znamená, že na výstupu může být po provedení logických operací přivedena například hodnota jedna i nula. Pokud takové chování lze zaručit, pak není potřeba jednotlivá schémata vzájemně propojovat.

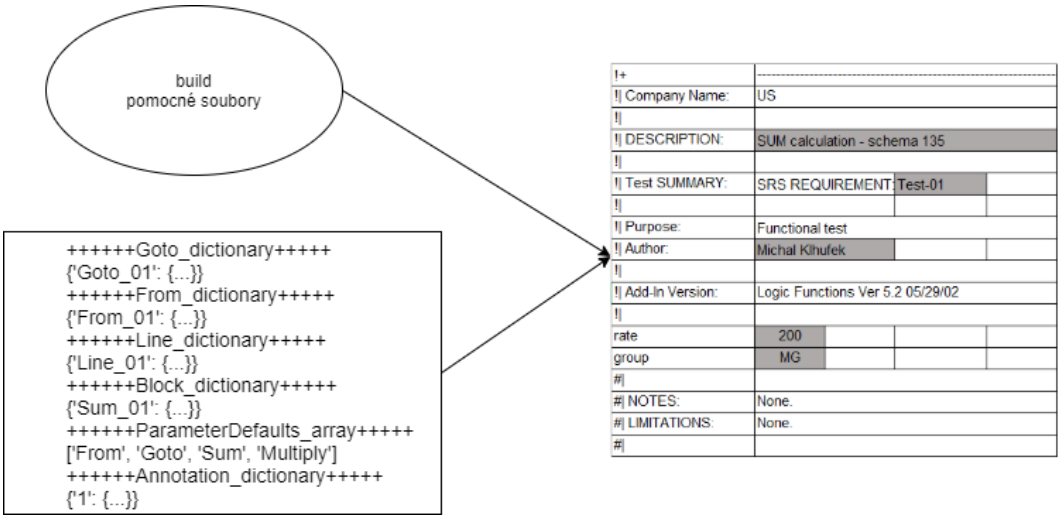
V případech, kdy u výstupního signálu v předcházejícím schématu nemůže být zaručena přepisovatelnost jeho hodnoty, je nutné v našem schématu, kde je takový signál převeden jako vstupní signál do některého z logických bloků, simulovat jeho kompletní schéma, které vede k získání jeho hodnoty. Pro větší názornost je daný problém znázorněn na následujícím obr. 5.3.



Obr. 5.3 ukázka propojenosti dvou schémat

5.4.3 Hlavičkový script

Hlavičkový script představuje modul, jehož funkcí je vygenerování prázdného výstupního souboru a následného doplnění správné hlavičky souboru podle patřičné šablony. Prázdná hlavička, která je z šablony dodána, obsahuje pouze základní kostru, ale chybí řada důležitých parametrů jako název testovaného schématu, skupina ke které testovací scénář patří (tato informace hraje důležitou roli při zkoumání návaznosti schématu na předcházející schémata), rychlost simulující běh reálného času či například název testera, který je zodpovědný za ověření správnosti daného schématu.



Obr. 5.4 ukázka fungování hlavičkového scriptu

Některé parametry (testovaného schématu) se dají získat ze zdrojového souboru *.mdl a jsou parsovací částí předány hlavičkovému scriptu, který je zde doplní. Problematičtější jsou například parametry rychlost hodinového kroku či skupina schémat, ke které testované schéma náleží. Jejich hodnota se nedá ze vstupního souboru zjistit. Zde je nezbytné prohledání pomocných souborů nacházejících se na síťovém disku. Rychlost hodinového kroku slouží ke správnému simulování času a spolu s návazností na okolní schémata tvoří důležitou součást hlavičky. Bez jejiho doplnění není možné testovací vektory správně spustit a otestovat jejich funkčnost. Pro představu je fungování hlavičkového scriptu znázorněno na obr. 4.4.

5.4.4 Generátor testovacích vektorů

Fáze vytváření testovacích scénářů a generování testovacích vektorů představuje nejdůležitější část celého systému. V této fázi je již vstupní soubor zcela načtený a uložený do vnitřní paměti. První krok, který je nezbytný nad těmito daty provést, spočívá v seřazení podle pořadí, ve kterém budou jednotlivé logické bloky testovány. Stručná ukázka fungování generátoru testovacích vektorů je zobrazena na obr. 4.5.

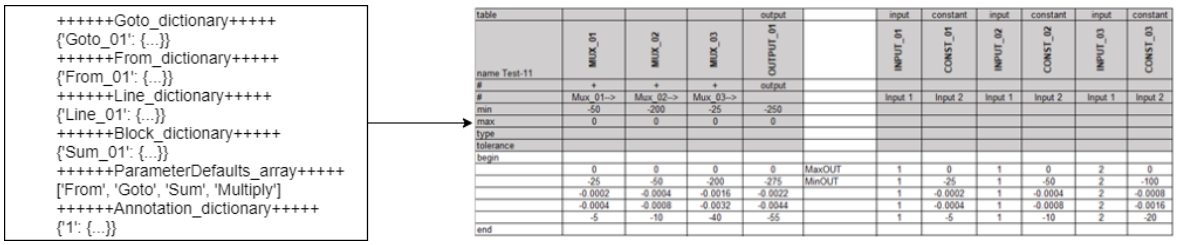


table				output		input	constant	input	constant	input	constant
name Test-11	MUX_01	MUX_02	MUX_03	OUTPUT_01		INPUT_01	CONST_01	INPUT_02	CONST_02	INPUT_03	CONST_03
#	+	+	+	output							
#	Mux_01-->	Mux_02-->	Mux_03-->			Input 1	Input 2	Input 1	Input 2	Input 1	Input 2
min	-50	-200	-25	-250							
max	0	0	0	0							
type											
tolerance											
begin											
	0	0	0	0	MaxOUT	1	0	1	0	2	0
	-25	-50	-200	-275	MinOUT	1	-25	1	-50	2	-100
	-0.0002	-0.0004	-0.0016	-0.0022		1	-0.0002	1	-0.0004	2	-0.0008
	-0.0004	-0.0008	-0.0032	-0.0044		1	-0.0004	1	-0.0008	2	-0.0016
	-5	-10	-40	-55		1	-5	1	-10	2	-20
end											

Obr. 5.5 ukázka generování testovacích vektorů

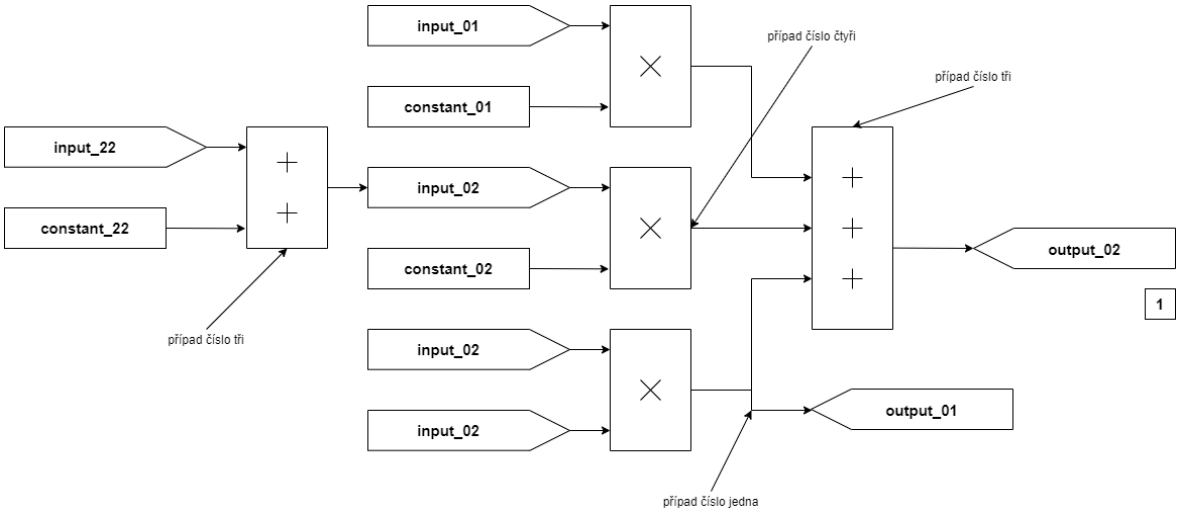
Jakmile jsou data správně seřazená, přichází čas na jejich využití při testování. Při testování mohou nastat celkem čtyři případy se kterými si aplikace musí poradit.

1. Logický blok má vstupní signál, jehož hodnota není potřeba získávat žádnými logickými bloky na pozadí a jehož výstupní signál nevstupuje do žádného dalšího logického bloku. V takovém případě dojde pouze k vyhledání šablony sloužící pro testování daného logického bloku. Pro názornou ukázku je daný problém znázorněn na obr. 4.6.

2. Logický blok má výstupní signál, který nevstupuje do žádného dalšího logického bloku, ale má vstupní signál, jehož hodnota je potřeba získávat logickými bloky na pozadí. V takovém případě je nezbytně nutné nasimulovat všechny bloky, které jsou podstatné pro získání vstupní hodnoty. V takovém případě dojde k volání sady šablon, které slouží pro testování všech logických bloků.Pro názornou ukázku je daný problém znázorněn na obr. 4.6.

3. Logický blok má vstupní signál, jehož hodnota není potřeba získávat žádnými logickými bloky na pozadí, ale má vstupní signál, jehož hodnota je potřeba získávat logickými bloky na popředí. V takovém případě je důležité nasimulovat všechny bloky, které jsou potřeba pro získání vstupní hodnoty. Tak dojde k volání sady šablon, které slouží pro testování všech logických bloků. Pro názornou ukázku je daný problém znázorněn na obr. 4.6.

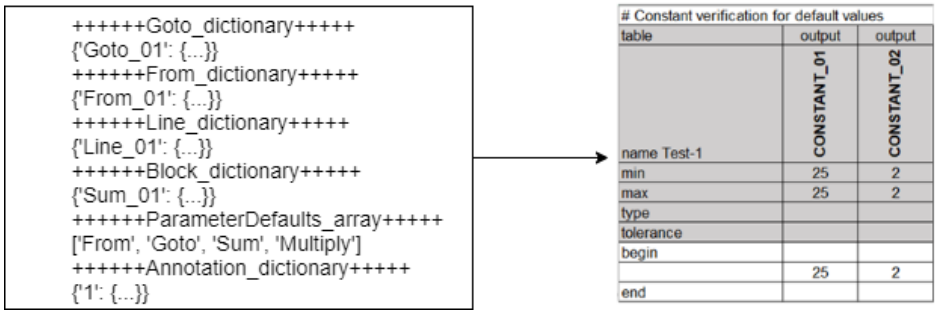
4. Logický blok má výstupní signál, který vstupuje do dalšího logického bloku a proto je nezbytně nutné provést testování všech následujících bloků až do situace, kdy je nalezen blok, jehož výstupní signál již nevstupuje do žádného dalšího logického bloku. Kromě toho také má vstupní signál, jehož hodnota je potřeba získávat logickými bloky na popředí. V takovém případě je potřebné nasimulovat všechny bloky, které jsou vyžadovány pro získání vstupní hodnoty. V takovém případě dojde k volání sady šablon, které slouží pro testování všech logických bloků. Pro názornou ukázku je daný problém znázorněn na obr. 4.6.



Obr. 5.6 ukázka všech případů se kterými si navyržený program musí poradit

5.4.5 Script na testování konstant

Poslední částí, která je důležitá ke správnému testování daného schématu je kontrola, zda všechny konstantní bloky nacházející se v daném schématu, kromě natvrdo zadrátovaných hodnot, mají možnost přepisování své hodnoty a to nejen na své standardní hodnoty, ale také v rámci rozsahu svého datového typu. Toto závěrečné testování má za cíl ověřit správnou implementaci daných bloků. Ukázka testování konstantních bloků na možnou přepisovatelnost jejich hodnot je znázorněna na obr. 4.7.



Obr. 5.7 ukázka generování testovacích vektorů pro konstanty

5.4.6 Knihovny

Kromě rozdělení samotné aplikace na sadu dílčích podčástí v podobě modulů či scriptů je vhodné využít i možnosti knihoven. V těchto knihovnách může být implementována určitá struktura, kterou je možné vzájemně předávat mezi jednotlivými moduly. Tyto knihovny potom lze dělit na interní, které budou implementovány v průběhu vývoje aplikace a externí, které představují existující knihovny a to jak open source tak i komerčně dostupné.

Interní Pro potřeby této práce je vhodné implementovat knihovnu, která bude obsahovat šablony všech logických bloků, které se nacházejí v knihovních funkcích jako je Simulink.mdl a EngileLib.mdl. Vzhledem k velké rozsáhlosti těchto bloků je velmi důležité je oddělit a jejich seznam poté uložit do vlastní knihovny. Tato knihovna může být volána z hlavní aplikace a za její pomoci lze správně parsovat vstupní soubor.

Externí Ne všechny konstrukce je vhodné zcela implementovat a právě v těchto případech se dá využít volně dostupných, případně komerčních knihoven. Jednou z takových knihoven je openpyxl. Tato knihovna slouží k usnadnění práce s *.xls soubory, které jsou výstupními soubory naší aplikace. Bez jejího využití by došlo ke zbytečnému zkomplikování aplikace a mohlo by se jednat o zdroj častých chyb a problémů. Kromě této knihovny jsou využity také re.py, sys.py, copy.py, traceback.py či colored.py.

5.4.7 Testy

Kromě rozdělení aplikace na jednotlivé části v podobě modulů, scriptů a knihoven je také potřeba navrhnout způsoby ověření správnosti vyvíjené aplikace. Primárně existují dvě hlavní části, jejichž chování je nezbytné otestovat. Jedná se o správnou funkci Parseru - zda byl vstupní soubor správně načten a zpracován, a poté zda samotné testovací vektory skutečně odpovídají testovanému logickému schématu.

Parser testy představují nezbytnou část aplikace, bez nichž nejde zaručit, že zpracování vstupního souboru proběhlo zcela bez problému. Doporučený postup spočívá v sestavení předem definované sady vstupních souborů a sady očekávaných výstupů. Při každém zásahu do funkcionality Parseru je potřeba spustit zmíněnou sadu testů, aby bylo zaručeno, že očekávané chování nebylo změněno.

V průběhu finální fáze dokončování aplikace sloužící k testování programovatelných logických obvodů je nezbytné provést její ověření, že splňuje všechny požadavky, které jsou na ni kladeny. Toto ověření je vhodné udělat pomocí předem připravených vstupních souborů, které by měly pokrýt jednak všechny logické bloky, které se ve vstupních souborech mohou nacházet, tak co možná nejvíce speciálních situací, které mohou ve schématech nastat, jako například jeho propojení s ostatními moduly. K těmto vstupním souborům by také měla existovat sada očekávaných výstupních hodnot pomocí nichž jde daný obvod adekvátně otestovat. Tato sada by měla projít kontrolou nejen autorem této práce, ale i ostatními testery, aby bylo prokázáno, že ověření je důkladné a správné. Jakmile by aplikace prošla zmíněnými testy lze ji považovat za vhodnou a použitelnou jako dodatečný nástroj při práci testera.

6 Implementace

V úvodu této kapitoly jsou nejprve popsány využívané implementační prostředky jako například zvolený programovací jazyk, vývojové prostředí, testovací operační systém a mnohé další. Ve druhé části se nachází následný popis implementace jednotlivých podčástí výsledné aplikace. V poslední části je uvedeno shrnutí problémů při implementaci spolu s možnými problémy, které vzniknuty při využívání aplikace.

6.1 Implementační nástroje

Jak již bylo popsáno v kapitole 5, při vývoji aplikace na testování logických programovatelných obvodů, byl pro implementaci zvolen programovací jazyk Python ve verzi 2.7. K vytvoření zdrojových kódů výsledné aplikace bylo využito vývojového prostředí Komodo IDE verze 11.0. Výhodou zmíněného vývojového prostředí je podpora jazyka Python, jak ve verzích 2.x, tak i ve verzích 3.x. Další výhodu prostředí Comodo je možnost vytváření multiplatformních aplikací a případné využívání řady dodatečných modulů, které prostředí nabízí. Při samotné práci je tak možné využívat řady funkcí známých z konkurenčních vývojových prostředí, jako je například automatické doplňování kódu či upozornění na nevyužití jednotlivých proměnných.[22]

Implementovaná aplikace je vyvinutá pod operačním systémem Windows 10 Pro 64bit s parametry vývojového počítače Intel Core i5-2540M (3.30 GHz), 8 GB DDR3, 512 GB SSD SATA. Pro následné potřeby a využití v praxi byla funkcionalita a správnost aplikace otestována i pro operační systémy Windows 7 a Windows Xp, které jsou nadále využívány v interní síti a proto je možné očekávat spuštění dané aplikace na těchto stanicích.

Jak již bylo popsáno v kapitole 5., vývoj grafického rozhraní nebyl vyžadován, a z toho důvodu je aplikace na testování logických obvodů vytvořena formou konzolové aplikace. Případné rozšíření na grafickou aplikaci je možné a proto je blíže popsáno v kapitole 7. popisující možná budoucí rozšíření. Ovládání aplikace je snadné a uživateli stačí v konzoli zadat následující parametry:

- python parser.py (jedná se o název aplikace)
- input.mdl (jedná se o vstupní testovací soubor)
- outrput.xls (jedná se o výstupní soubor obsahující testovací vektory)

Aby bylo možné aplikaci správně spustit je nezbytné, aby se nacházela na síťovém disku obsahujícím testovací schémata a dodatečné informace z buildu, díky nimž jsou stahovány potřebné informace ke generování hlavičky výstupního souboru a také pro

zjišťování informací o návaznosti na ostatní schémata. Pro potřeby této práce jsou dané údaje simulovány a nedochází k vyhledání a doplnění údajů ze síťového disku. Příklad spuštění aplikace je uveden na následující ukázce:

```
x:/ python parser.py input.mdl outpur.xls
```

Struktura zdrojových souborů odpovídá navrhovanému konceptu z kapitoly 5. Výsledná aplikace na testování logických obvodů je rozdělena na tři hlavní podčásti. Jedná se o hlavní spustitelný program, dodatkové moduly, které jsou aplikací volány, sada knihoven a to jak interních, vytvořených pro potřeby této práce, tak externích v podobě volně dostupných knihoven k širokému využití.

6.2 Knihovny

Při práci na programu bylo v prvním kroce nezbytně nutné provést podrobný průzkum obsahu knihoven Simulink.mdl a EnginesLib.mdl. Tyto knihovny představují základní stavební kámen pro tvorbu schémat logických programovatelných obvodů. V jednotlivých knihovnách jsou kompletně popsány všechny dostupné logické bloky. Z toho důvodu bylo v počátku důležité definovat podstatné parametry, které jsou podstatné k otestování, že daný blok je ve schématu správně použit a dané schéma odpovídá očekávanému chování.

Jakmile byly všechny potenciální logické bloky jednoznačně identifikovány, došlo k sestavení řady šablon pro každý z těchto bloků. Když aplikace začne zpracovávat načtený soubor, je využito této řady šablon, které umožňují jednoznačně identifikovat bloky nacházející se v daném logickém schématu. Bloky ve zdrojovém vstupním kódu obsahují řadu parametrů, ale také řadu dobrovolných parametrů, které nejsou pro testování důležité a proto jsou ignorovány. Kromě důležitých údajů získaných ze vstupního souboru jsou pro každý blok také přidány interní specifické informace sloužící pro snadnější
a intuitivnější následné testování. Ukázka testovací šablony sloužící pro zpracování vstupního souboru je znázorněna na následující ukázce. Jedná o šablonu pro logický blok typu Hystereze

```
# VARIABLE_HYSTERESIS
'HYS': {
'Name': 'NIL',
'Ports': 'NIL',
'Position': 'NIL',
'SourceBlock': 'NIL',
'SrcPort': 'NIL',
'DstPort': 'NIL',
'REF_Inputs_MK': 'NIL',
'Inputs_MK': 'NIL',
'Outputs_MK': 'NIL',
'Type_MK': 'HYS'}}
```

Code 6.1 šablona pro blok Hystereze

Kromě šablon pro logické programovatelné bloky je důležité sestavit i řadu šablon pro ostatní prvky, které se mohou v daném schématu nacházet. Jedná se o šablony pro vstupní a výstupní signály společně s šablonami pro jednotlivé propojky mezi jednotlivými signály či bloky. Ukázka šablony sloužící pro zpracování vstupních, výstupních signálů a propojek mezi jednotlivými bloky či signály je znázorněna na následující ukázce.

```
# INPPUT
'From': {
'Name': 'NIL',
'Position': 'NIL',
'GotoTag': 'NIL',
'DstPort': 'NIL'}}
```

Code 6.2 šablona pro vstupní signál

```
# OUTPUT
'Goto': {
'Name': 'NIL',
'GotoTag': 'NIL',
'BackgroundColor': 'NIL',
'Position': 'NIL',
'Inputs_MK': 'NIL',
'SrcPort': 'NIL'}}
```

Code 6.3 šablona pro výstupní signál

```
# LINE
'Line': {
'SrcBlock': 'NIL',
'SrcPort': 'NIL',
'DstBlock': 'NIL',
'DstPort': 'NIL'}}
```

Code 6.4 šablona pro propojky mezi jednotlivými bloky

Pro zvýšení univerzálnosti a využitelnosti popsaných šablon došlo k vytvoření interní knihovny s názvem templatelib.py. Jedná se o knihovnu, jejíž využití je nutné pro správné parsování vstupního souboru a jejíž obsah plně odpovídá sadě dostupných logických bloků nacházejících se v knihovnách Simulink.mdl a EnginesLib.mdl. Kromě toho jsou v knihovně popsány i postupy, podle kterých správně parsovat bloky, které se nachází ve zmíněných knihovnách, ale jejich dodatečná implementace ve vstupním souboru je možná.

6.3 Ostatní využité knihovny

openpyxl.py Z důvodu chybějící nativní podpory při zpracování *.xls souborů ze strany Pythonu vznikla potřeba na využití externí knihovny, která by tuto práci usnadňovala. Zvolená knihovna má označení openpyxl a umožňuje snadnou práci se soubory typu *.xls, *.xlsx, ale dokonce i *.xml. Díky své univerzálnosti a otevřenosti v podobě open free licence našla své vysoké uplatnění u řady důležitých projektů.[23]

Pro potřeby této diplomové práce byla zvolena verze openpyxl 2.6.2, kterou je také nezbytné nainstalovat na stroj, kde bude daný program spuštěn a využíván. K instalaci jde využít například příkaz pip install openpyxl. Následné využití knihovny je velmi intuitivní a umožňuje řadu operací, kde ty nejdůležitější představují čtení a zápis do *.xls souborů. V praxi je knihovna nejvíce využívaná pro emailové klienty, kdy umožňuje propojení automatického odesílání emailů a čtení kontaktů z databáze uložení v *.xls souboru.[23]

re.py, os.py, sys.py, copy.py Kromě řady externích knihoven je možné v Pythonu využívat i řady vestavných modulů, které usnadňují samotnou práci se zpracovanými daty. Mezi tyto moduly patří zejména modul re.py, který slouží ke zpracování dat pomocí regulárních výrazů umožňujících zefektivnění a zrychlení běhu programu. Dalším modulem je copy.py, který v rámci této práce usnadňuje práci s datovým typem dictionary, na kterém je zpracování načtených dat v rámci této diplomové práce založeno. Modul os.py následně umožňuje práci s operačním systémem, například při načítání či ukládání potřebných dat na disk počítače. Posledním využívaným modulem je modul sys.py, z něhož je využita funkcionalita na zpracování vstupních parametrů získaných při spuštění programu.

traceback.py K větší informovanosti o případných chybách, které se ve vyvíjeném programu vyskytují je vhodné být dostatečně podrobně informován. K tomu se dá využít nástrojů, které nabízí modul traceback.py. Traceback by se dal do češtiny přeložit jako zpětný záznam a funkcí knihovních nástrojů tedy je informovat o případných chybách v programu formu zpětných funkcí, kde se chyba při běhu systému vyskytla. Jednotlivé chyby jsou poté zobrazeny v pořadí, v jakém se v programu vyskytly, neboli v pořadí v jakém byly ukládány na zásobníku chyb, který je následně vyprázdněn do výstupové konzole.

colored.py Poslední dodatečnou knihovnou, která byla v rámci vývoje programu využita, je knihovna colored.py. Tato knihovna není úplně důležitá k samotnému běhu programu, ale nabízí větší míru přehlednosti při automatickém testování bezchybnosti

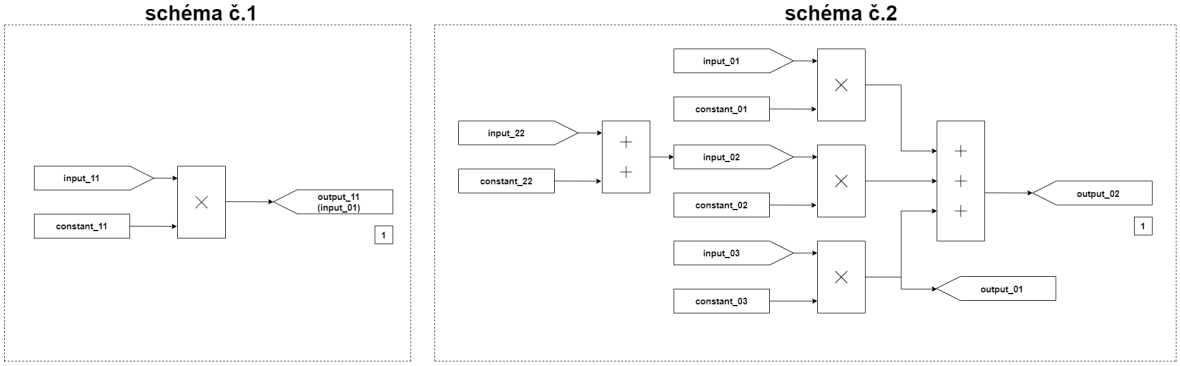
běhu aplikace při provádění testů.

Knihovna je taktéž dodávaná v open source licenci, což její využití usnadňuje. Samotné využití knihovny je poměrně intuitivní a slouží pouze k obarvování částí výstupního textu do konzole na základě předem daných pravidel. Pro potřeby této diplomové práce byla zvolena verze colored 1.4.2, kterou je také nezbytné nainstalovat na stroj, kde bude daný program spuštěn a využíván. K instalaci jde využít například příkaz pip install colored.

6.4 Implementace aplikační logiky

Implementace aplikační logiky programu na testování logických programovatelných obvodů vychází z požadavků na funkcionalitu systému popsaných v kapitole 5. Průběh implementace jednotlivých podčástí bude podrobně rozepsáno v následujících podkapitolách. Jednotlivé podčásti jsou doplněny i o názorné ukázky výstupní hodnoty, které jsou po dokončení jejich funkcionality získány.

Pro zvýšení názornosti a ukázky celkového systému bude funkcionalita aplikace demonstrována na zpracování následujícího propojeného schématu na obr. 6.1. Na obrázku
je zobrazeno schéma č.1, jehož výstupní signál představuje vstupní signál do schématu č.2. Tento případ představuje možnost vzájemného provázání schémat napříč jednotlivými vstupními soubory.



Obr. 6.1 ukázka propojenosti dvou schémat

6.4.1 Parser

První logickou podčást, kterou musí aplikace na zpracování logických programovatelných obvodů implementovat, je Parser. Tato podčást načítá vstupní soubor ve formátu *.mdl a za využití interní knihovny s názvem template.py dochází k jejímu zpracování.

Zpracování probíhá na základě rozhodovacího konečného automatu, který obsahuje sadu pravidel, jenž odpovídají všem logickým blokům, propojek mezi nimi a případně

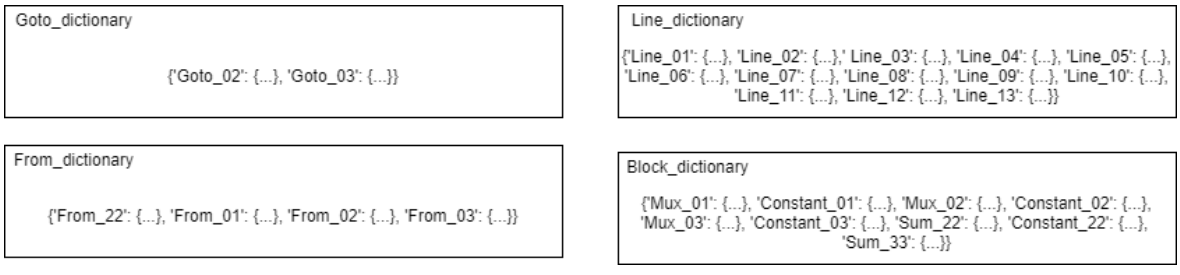
vstupním a výstupním signálům. Konečný automat je implementovaný přímo v programu parser.py a funkce, která má danou logiku na starosti se jmenuje fceparser. Nicméně funkce fceparser vyžaduje ke svému běhu další řadu podpůrných funkcí, které společně slouží k parsování vstupního souboru. Jakmile při zpracování Parser narazí na část schématu, která odpovídá pravidlům z konečného automatu, dojde k uložení všech nezbytných informací pro testování dané části do proměnné datového typu dictionary. V Parseru jsou implementovány celkem čtyři typy záznamů typu dictionary, které pro každý další svůj záznam vytvářejí záznam v dané proměnné.

Pro větší představu je na následující ukázce pseudokódu zobrazeno strukturální rozdělení do vnitřní struktury za využití dictionaries využívaných v této diplomové práci.

```
++++++Goto_dictionary+++++
{'Goto_01': {'Name': 'Goto_01', 'SrcPort': '1', 'GotoTag': 'output_01',
'BackgroundColor': 'NIL', 'Position': '[10,20,30,20]', 'Inputs_MK': 'S
um_01'}}
++++++From_dictionary+++++
{'From_01': {'Position': '[15,74,25,10]', 'DstPort': 'NIL', 'GotoTag':
'input_01', 'Name': 'From_01'}}
++++++Line_dictionary+++++
{'Line_01': {'SrcBlock': 'From_01', 'DstPort': '1', 'DstBlock': 'MUX_01
', 'SrcPort': '1'}}
++++++Block_dictionary+++++
{'Sum_01': {'Inputs': '+++', 'DstPort': '1', 'Name': 'SUM_01', 'SrcPort
': '3 1', 'Type_MK': 'Sum', 'REF_Inputs_MK': 'NIL', 'Outputs_MK': 'Got
o_01', 'Position': '[34,8,30,20]', 'Inputs_MK':
'Mux_01 Mux_02 Mux_03', 'Ports': '[3,1,0,0,0]'}}
```

Code 7.1 vnitřní struktura načteného schématu

Jakmile Parser projde kompletně celý načtený soubor, který v našem případě reprezentuje schéma z obr. 6.2., nastane vyhodnocení podle sady pravidel z konečného automatu. Pokud jsou správně vyhodnoceny všechny jednotlivé podčásti, dojde k jejich předání další části programu. Ukázka, jakým způsobem dochází k předání jednotlivých podčástí další části v našem systému je znázorněna v následující ukázce kódu.



Obr. 6.2 ukázka načtení vstupního souboru do vnitřní paměti

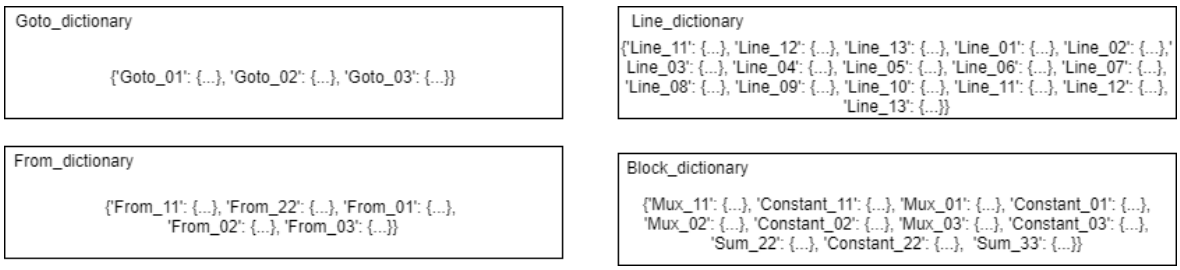
6.4.2 Zkoumač návaznosti

Druhou logickou podčást vyvíjené aplikace představuje rozšíření základní aplikace na zkoumání návaznosti testovaného schématu s využitím předchozího schématu, ve kte-

rém jsou určité hodnoty signálů získávány. Celý proces nejprve začíná zpracováním aktuálního schématu u kterého dojde ke zjištění, zda pro některé vstupní hodnoty není nastaven přepínač jejich přepisovatelnosti. Tato informace se získává z buildu na síťovém disku.

Jakmile je takový vstupní signál identifikován nastává vyhledání schématu, ve kterém se daný signál nachází jako výstupní. Tuto informaci lze získat z pomocných souborů nacházejících se na síťovém disku. Pokud jde dané schéma jednoznačně identifikovat, nastane vyhledání posloupnosti logických členů, které vedou k získání daného signálu. V praxi to znamená spouštění parseru i na dané schéma a následné vyhledání jen zkoumané posloupnosti logických bloků. Když je daná posloupnost nalezena, nastane její vložení do datové struktury vytvořené parserem.

Na ukázkových schématech znázorněných na obr. 6.3. je vidět, že výstupní signál z levého schématu vstupuje do schématu na pravé části. script na zkoumání návaznosti provede vložení konstrukce vedoucí k získání daného signálu a přidá tuto informaci do datového typu dictionary, který vytvořila předchozí část Parseru. Ukázka uložení dané informace do dictionary je znázorněna na následující ukázce kódu.



Obr. 6.3 ukázka rozšíření záznamu o data z propojeného schématu

6.4.3 Hlavičkový script

Třetí logickou podčást představuje hlavičkový script. Účelem této části je nejprve vytvoření prázdného výstupního souboru podle jména, které je získáno ze vstupního parametru při spuštění scriptu a následné vložení template pro hlavičku výstupního souboru. Samotný script není součástí hlavního testovací programu, ale je v něm volán ve chvíli, kdy je nutné vytvořit výstupní soubor a doplnit mu hlavičku se všemi potřebnými údaji, aby bylo možné po doplnění testovacích vektorů vyzkoušet jejich správnost.

Při vložení hlavičky z template je důležité doplnit řadu parametrů mezi které patří jméno autora, krok hodin simulující čas v daném schématu či skupina schémat, ke které právě testované schéma patří. Přesný název i s verzí testovaného schématu je získán ze vstupního *.mdl souboru a hlavičkovému scriptu je předán z Parseru. Jméno testera, který daný test vytváří je získáno ze systémových informací a závisí na

tom, kdo je momentálně přihlášený na stroji, kde se testovací vektory vytvářejí. Parametry jako hodnota hodinového kroku či skupina ostatních schémat, ke kterým testované schéma náleží se získávají z buildu případně z pomocných souborů nacházejících se na síťovém disku.

Jakmile hlavičkový script získá všechny údaje, nastává jejich vložení do hlavičky, která byla do výstupního souboru nahrána z templaty. Ukázka vyplněné hlavičky pro testované schéma je zobrazeno na obr. 6.4.

!+	----------			
!| Company Name:	US			
!|				
!| DESCRIPTION:	SUM calculation - schema 135			
!|				
!| Test SUMMARY:	SRS REQUIREMENT:	Test-01		
!|				
!| Purpose:	Functional test			
!| Author:	Michal Klhufek			
!|				
!| Add-In Version:	Logic Functions Ver 5.2 05/29/02			
!|				
rate	200			
group	MG			
#|				
#| NOTES:	None.			
#| LIMITATIONS:	None.			
#|				

Obr. 6.4 ukázka hlavičky zpracované pro uvedené schéma

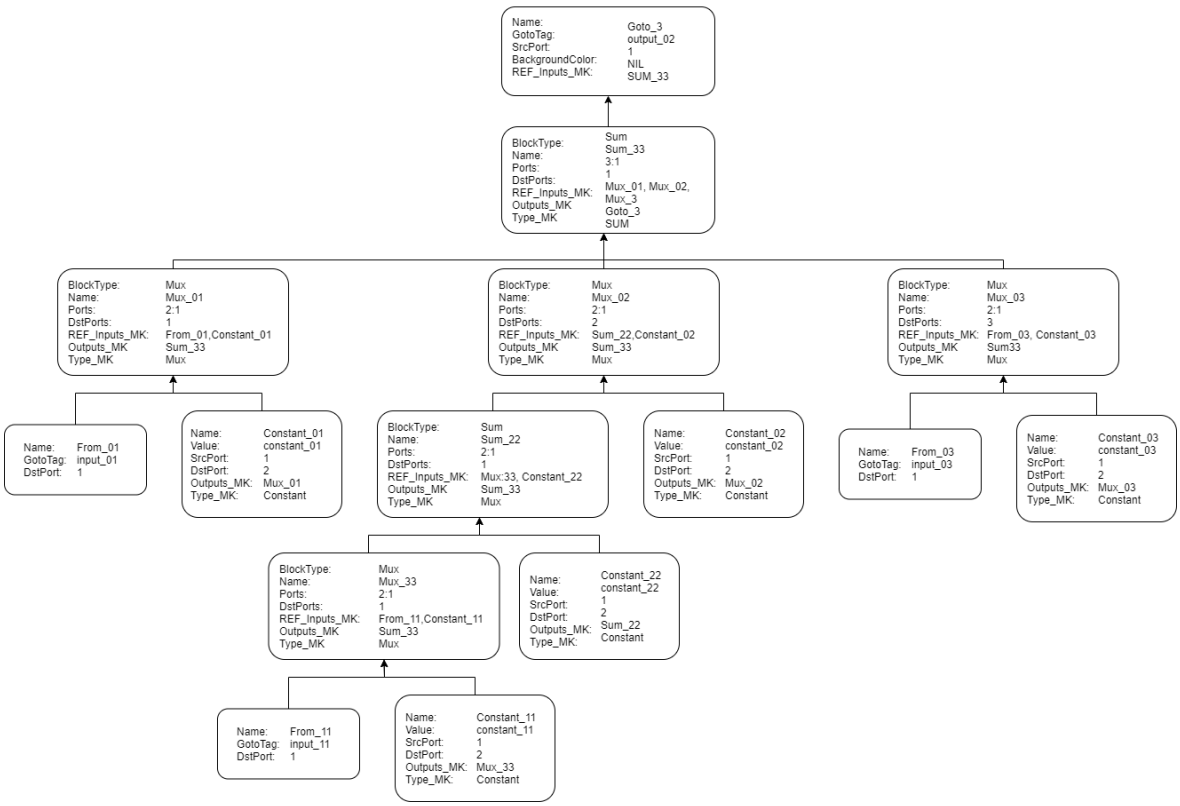
6.4.4 Generátor testovacích vektorů

Čtvrtou logickou podčástí, a také tou nejdůležitější, je část na vytváření testovacích scénářů a testovacích vektorů. Tato podčást na svém vstupu získává čtyři oddělené záznamy datového typu dictionary, které byly získány na základě parsování vstupního *.mdl souboru a případně upraveny, pokud je to zapotřebí podčástí systému na zkoumání vzájemné návaznosti.

Prvním krokem při vytváření testovacích scénářů je správné seřazení jednotlivých záznamů v dictionsaris. Seřazení probíhá podle parametru position, který se nachází u každého výstupního signálu z gotodictionary. Jednotlivé údaje jsou potom vyhodnoceny vůči levému hornímu rohu v daném schématu a na základě vzdálenosti od něj jsou seřazeny. Výsledné seřazení udává pořadí v jakém budou jednotlivé bloky testovány obdobně jako by postupoval tester při manuálním vytváření testovacích scénářů. Výjimku představují případy, kdy jednotlivé výstupní signály mají přímo dané v jakém pořadí se mají testovat a v takovém případě je seřazení provedeno na základě této informace.

Ve chvíli, kdy je seřazení dokončeno, přichází část vytváření testovacích scénářů a generování testovacích vektorů pro každý logický blok nacházející se v daném schématu.

Využití datového dictionary a upravení za pomoci proměnných dává možnost vygenerování grafové struktury, která udává, jakým způsobem bude každý blok testován. Pro ukázku je na následujícím obr. 6.5 vytvořena ukázka stromové struktury pro testování logického bloku sčítačky, který ke správnému testování výstupní hodnoty musí provést testování kompletní logiky, pomocí níž se sčítačka vypočítává.



Obr. 6.5 grafová struktura reprezentující načtené schéma

Jakmile je stromová struktura připravená a hlavičkový soubor připravil výstupní soubor nastává část, při které jsou ze šablon postupně doplňovány a následně upraveny jednotlivé logické bloky společně s patřičnými testovacími hodnotami pro ověření funkčnosti dané podčásti logického obvodu. Postup, podle kterého jsou jednotlivé testovací scénáře vytvářeny, vychází z kapitoly 5, kde jsou všechny čtyři možné případy, které mohou nastat. V aplikaci jsou dané případy řešeny pomocí rekurzivních funkcí, které se vzájemně mezi sebou volají a zanořují v případech, kdy je testování dané logické sekvence složitějšího rázu. Jakmile je testovací scénář vytvořen a plně odpovídá dané části schématu, je poté vložen do již připraveného výstupního souboru. Názorná ukázka výstupního testovacího scénáře společně s testovacími vektory pro logický blok sčítačky je zobrazena na obr. 6.6.

# Sum2 (+ + +) - OUTPUT_01					Comments										
table				output		input	constant	input	constant	input	constant	input	constant	input	constant
name Test-11	MUX_01	MUX_02	MUX_03	OUTPUT_01		INPUT_01	CONST_01	INPUT_02	CONST_02	INPUT_03	CONST_03	INPUT_22	CONST_22	INPUT_11	CONST_11
#	+	+	+	output				Sum-->				Input 1	Input 2		
#	Mux-->	Mux-->	Mux-->			Input 1	Input 2	Input 1	Input 2	Input 1	Input 2	Mux-->		Input 1	Input 2
min	-50	-200	-25	-250											
max	0	0	0	0											
type															
tolerance															
begin															
	0	0	0	0	MaxOUT	1	0	1	0	2	0	3	0	4	0
	-25	-50	-200	-275	MinOUT	1	-25	1	-50	2	-100	3	-200	4	-25
	-0.0002	-0.0004	-0.0016	-0.0022		1	-0.0002	1	-0.0004	2	-0.0008	3	-0.0016	4	-0.0002
	-0.0004	-0.0008	-0.0032	-0.0044		1	-0.0004	1	-0.0008	2	-0.0016	3	-0.0032	4	-0.0004
	-5	-10	-40	-55		1	-5	1	-10	2	-20	3	-40	4	-5
end															

Obr. 6.6 ukázka testovacích vektrorů sloužících k otestování sčítacího bloku

6.4.5 Script na testování konstant

Poslední částí systému je script na testování konstant a jejich přepisovatelnosti. Script není součástí hlavního programu a jeho volání probíhá ve finální fázi, když celý zbytek výstupního souboru je již adekvátně vytvořen. V tomto bodě dochází k volání scriptu, který na základě hodnot z datového typu dictionary obsahující kompletní seznam konstant a v návaznosti na informace z buildu provede nastavení konstant a jejich testování na základní hodnoty, které konstanta má obsahovat, tak i na případnou přepisovatelnost hodnot v dané konstantě.

Proces obdobně jako u generátoru testovacích vektorů doplní potřebné šablony do výstupního souboru a v závislosti na jednotlivých konstantách a jejich datovém typu následně provede jejich testování. Ukázka výstupního testování konstant pro schéma na obr. 6.1. je zobrazeno na obr. 6.7.

6.5 Problémy při implementaci řešení

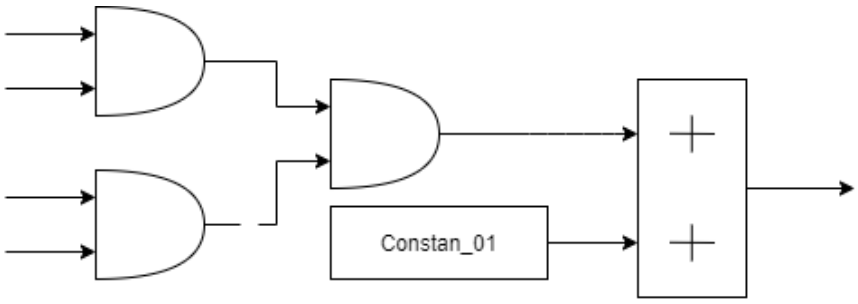
Při testování schémat obsahující logické programovatelné obvody existuje řada problémů se kterými je nutné počítat a na které musí být aplikace připravena. Tyto problémy mohou být dvojího charakteru. První představuje chyba v samotném testovaném schématu a to jak neúmyslná, která vznikne nepozorností při návrhu, tak hlubší, která znamená problémy v samotné logice navrženého schématu.

Chyby, které se nacházejí v samotném testovaném schématu a které vznikly nepozorností vývojáře, představují případy, které se snadno odhalují a následně opravují. Názorným příkladem takové chyby, se kterou si systém musí poradit, je například propojení všech bloků v daném schématu. Může se stát, že vývojář ve složitém obvodu propojí všechny jednotlivé bloky pomocí propojovacích kabelů, ale v jednom bodě přehlédne takové propojení. Díky vysokému počtu ostatních bloků v oblasti se daný problém dá přehlédnout, ale ve zdrojovém kódu schématu bude názorně vidět nepro-

# Constant verification for default value					
table	output	output	output	output	output
name Table-10	CONSTANT_11	CONSTANT_22	CONSTANT_01	CONSTANT_02	CONSTANT_03
min	1	1	250	150	-500
max	1	1	250	150	-500
type					
tolerance					
begin					
	1	1	250	150	-500
end					
# Constant verification for override capability					
table	constant	constant	constant	constant	constant
name Table-11	CONSTANT_11	CONSTANT_22	CONSTANT_01	CONSTANT_02	CONSTANT_03
min	1	1	250	150	-500
max	1	1	250	150	-500
type					
tolerance					
begin					
	1.0002	1.0002	250.05	150.03	-499.9
end					

Obr. 6.7 testovací vektory sloužící k ověření konstat ve schématu

pojenost jednotlivých bloků. Od aplikace se v takovém případě očekává informování testera o vzniklém problému a jeho povinností je vyžádat nápravu a opravu daného schématu. Názorná ukázka nepropojení všech bloků je zobrazena na obr. 6.8.



Obr. 6.8 ukázka nepropojenosti jednotlivých bloků

Větší problém představují chyby, které vznikají nepozorností vývojáře a mají hlubší charakter, jako například nevhodné použití logického bloku v dané části schématu. V takových případech tester ani tento program nemůže zaručit plné zkontrolování logiky daného schématu, neboť ne všechny stavy v daném bloku jsou simulací dosáhnutelné. Logický blok je poté označený jako plně neotestovatelný. Aplikace při nalezení bloku, který nelze plně otestovat informuje testera, který musí s tímto problémem počítat.

6.6 Možné problémy při použití aplikace

Mezi možné problémy, které mohou vzniknou při obsluhování aplikace patří zejména překlepy nebo neporozumění specifikace aplikace ze strany uživatele. Jako vstupní sou-

bor přijímá aplikace pouze název vstupního souboru například modelA.mdl, ale nikoliv název jeho aktuální upravené verze, který se na síťovém disku také často nachází v podobě modelA5.mdl. Kromě této nepřesnosti mohou nastat případy, kdy uživatel bude chtít vytvoření výstupního souboru ve formátu *.xlsx místo *.xls. Tato varianta také není přípustná, neboť systém musí být zpětně kompatibilní nejen se systémem Window XP, ale také s aplikací Microsoft Excel 2003. Z toho důvodu není výstupní soubor s příponou *.xlsx přípustný. V obou zmíněných případech je tester včasně informován a vyzván k nápravě své chyby.

7 Metodika testování správnosti aplikace

Jakmile je aplikace na testování logických programovatelných obvodů zdárně dokončena a splňuje všechny předem stanovené požadavky na její funkčnost, je nezbytné tato tvrzení ověřit patřičnou verifikovací její funkčnosti.

V aplikaci jsou dvě stěžejní místa, ve kterých může dojít k chybě při implementaci, a proto je nezbytné jejich důkladné otestování. V prvním případě se jedná o podčást sloužící k parsování vstupního souboru ve formátu *.mdl, a v druhém případě se jedná o podčást generující výsledné testovací scénáře obsahující samotné testovací vektory uložené v souboru typu *.xls. Oba zmíněné případy představují citlivá místa, která je nutné důkladně otestovat a to jak pomocí drobnějších testů, tak pomocí řady komplexních simulačních schémat, pro které budou přesně stanovené testovací vektory, které jsou očekávány. Jednotlivá testovací schémata budou mít předem vytvořené výstupní testovací scénáře, které odpovídají svými výstupními hodnotami danému schématu
a které budou vytvořeny za pomoci testera běžným postupem bez využití aplikace vyvinuté v rámci této diplomové práce. V následujícím kroce budou oba dva výstupy testování testerem i aplikací vzájemně porovnány a určena míra jejich podobnosti. V ideálním případě by měly být výsledky testování shodné.

Vzhledem k tomu, že vyvinutý systém bude spuštěn na schématech, která jsou svojí povahou velmi citlivá a důvěrná, není jejich zveřejnění mimo interní potřeby společnosti Honeywell možné. Právě proto byla pro potřeby této práce sestavena simulační logická programovatelná schémata, která jsou dodávána společně s prací na přiloženém CD. Nejedná se o skutečná schémata, která by se v praxi využívala na reálném hardwaru, avšak jejich složitost odpovídá reálným schématům. Na reálných schématech využívaných ve společnosti Honeywell byla práce ovšem také testována s pozitivními výsledky.

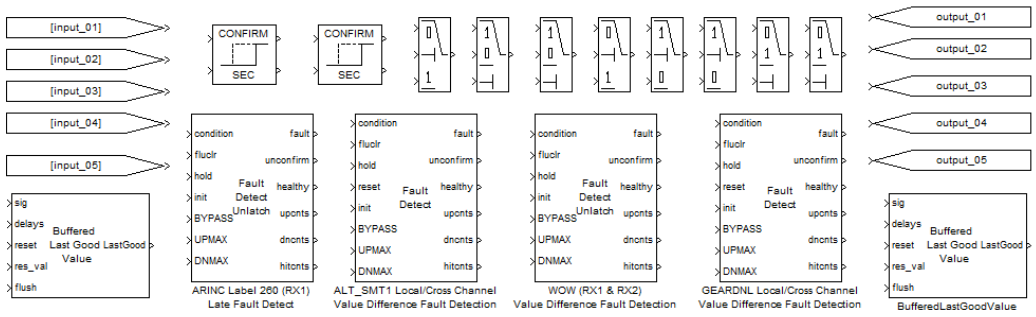
Primární testování probíhalo na stanici s operačním systémem Windows 10 Pro 64bit s parametry vývojového počítače Intel Core i5-2540M (3.30 GHz), 8 GB DDR3, 512 GB SSD SATA. Vzhledem k deklarovaným vlastnostem a zpětné kompatibilitě bylo testování aplikace provedeno také na MS Windows ve verzích XP a Windows 7 nacházejících se v interní síti společnosti Honeywell. Testování na všech zmíněných verzích systému Windows dopadlo úspěšně.

V následujících třech kapitolách budou nejprve podrobně popsány jednotlivé metody testování dílčích podčástí a to jak výstupů z Parseru, tak finálního výstupu, v konečné fázi aplikace doplněné o reálné výstupy získané z běhu aplikace. V poslední kapitole jsou shrnuty jednotlivé výsledky z dílčích podčástí z pohledu jejich úspěšnosti a rychlosti zpracování v porovnání s prací testera.

7.1 Testování Parseru

Parser představuje první zranitelné místo ve vyvýjené aplikaci, neboť právě v tomto bodě probíhá načtení vstupního souboru ve formátu *.mdl a jeho následná transformace podle pravidel daných konečným automatem do vnitřní struktury programu. Správná funkčnost Parseru je naprosto nezbytná pro další fungování aplikace, a z toho důvodu je vhodné provádět ověřování její správnosti při každém i sebemenším zásahu do zdrojového kódu Parseru.

Pro testování není nutné mít rozsáhlou sadu testů, stačí i menší sada testů, která dokáže plně pokrýt všechny možné případy logických bloků, vstupních a výstupních signálů, případně propojovacích kabelů, které se mohou v reálných schématech vyskytovat. Z toho důvodu byla sestavena skupina deseti vzorových schémat, která po logické stránce představují spustitelná schémata, ale vyskytují se v nich všechny možné případy jakým lze jednotlivé prvky ve schématu sestavit. Ukázka jednoho z testovacích schémat je zobrazena na obr. 7.1, kde je názorně vidět řada logických bloků jako jsou cofirm secy, přepínací bloky, ale také bloky na odhalování chybných bitů společně se vstupními a výstupními signály.



Obr. 7.1 testovací bloky pro ověření Parseru

Testování jednotlivých schémat probíhá za běhu systému. Systém v testovacím režimu pracuje zcela normálně jako kdyby zpracovával uživatelem zadané schéma, ale jakmile nastane fáze parsování vstupního souboru, aplikace neprovede parsování jen uživatelem zvolného vstupního souboru, ale také sady testovacích schémat pro které ověří,
že program i nadále funguje zcela správně a od posledního spuštění aplikace nedošlo k neočekávané změně ve zdrojovém kódu, která by vedla k problémům při parsování.

Jakmile systém provede fázi parsování pro všechny testovací schémata, uloží si jejich výsledné načtené struktury do proměnných typu dictionary, které porovná s předem připravenými soubory obsahujícími správné načítací hodnoty pro všechny testovací schémata. Ukázka testovacích dat, která jsou uložena pro patřičné schéma a která jsou následně porovnávána, jsou pro schéma na obr. 7.1 zobrazena na následujícím kódu.

```
==========================
++++++Goto_dictionary+++++
==========================
{'Goto24': {...}, 'Goto34': {...}, 'Goto13': {...}, 'Goto31': {...}, 'G
oto44': {...}}
==========================
++++++From_dictionary+++++
==========================
{'From69': {...}, 'From58': {...}, 'From60': {...}, 'From10': {...}, 'F
rom63': {...}}
==========================
++++++Line_dictionary+++++
==========================
{'Line_15': {...}, 'Line_17': {...}, 'Line_16': {...}, 'Line_19': {'...
}, 'Line_18': {...}}
==========================
++++++Block_dictionary+++++
==========================
{'GEARDNLLocal/CrossChannel\\nValueDifferen': {...}, 'SWITCH54': {...},
'SWITCH3': {...}, 'ARINCLabel260(RX1)\\nLateFaultDetect': {...}, 'SWITC
H1': {...}, 'BufferedLastGoodValue2': {...}, 'ADJCONFIRM1': {...}, 'ADJ
CONFIRM2': {...}, 'SWITCH': {...}, 'WOW(RX1&RX2)\\nValueDifferenceFault
Det': {...}, 'ALT_SMT1Local/CrossChannel\\nValueDiffere': {...}, 'SWITC
H25': {...}, 'BufferedLastGoodValue': {...}, 'SWITCH100': {...}, 'SWITC
H12': {...}, 'SWITCH10': {...}}
==========================
++++++ParameterDefaults_array+++++
==========================
['Constant', 'Goto', 'SubSystem']
```

Code 7.2 upravená data načteného schématu

Pokud se výstupní struktura dat shoduje s daty nacházejícími se aktuálně v proměnných datového typu dictionary, nastane vyhodnocení dílčího testu. K vyhodnocení testu je využito i grafických funkcí volně dostupné knihovny colored.py pro jazyk Python, která dá testerovi mnohem jasnější informaci, zda daný test prošel nebo neprošel. Jednotlivé stavy jsou díky knihovním funkcím odlišeny zelenou a červenou barvou. Ukázka stavu kdy všechny testy projdou je na následujícím kódu.

```
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
```

Code 7.3 zhodnocení výstupu testu

Vzhledem ke struktuře a rozsáhlosti jednotlivých testů je po každém jejich běhu plně zajištěno, že systém po stránce parsování vstupního souboru ve formátu *.mdl plně odpovídá předepsané specifikaci a jeho funkcionalita i po další editaci zdrojového kódu byla či nebyla zachována. Díky tomuto mechanismu je dodatečná editace a rozšiřování funkcionality systému nadále možná s jistotou kvalitního ověření provedených změn.

7.2 Testování generátoru testovacích vektorů

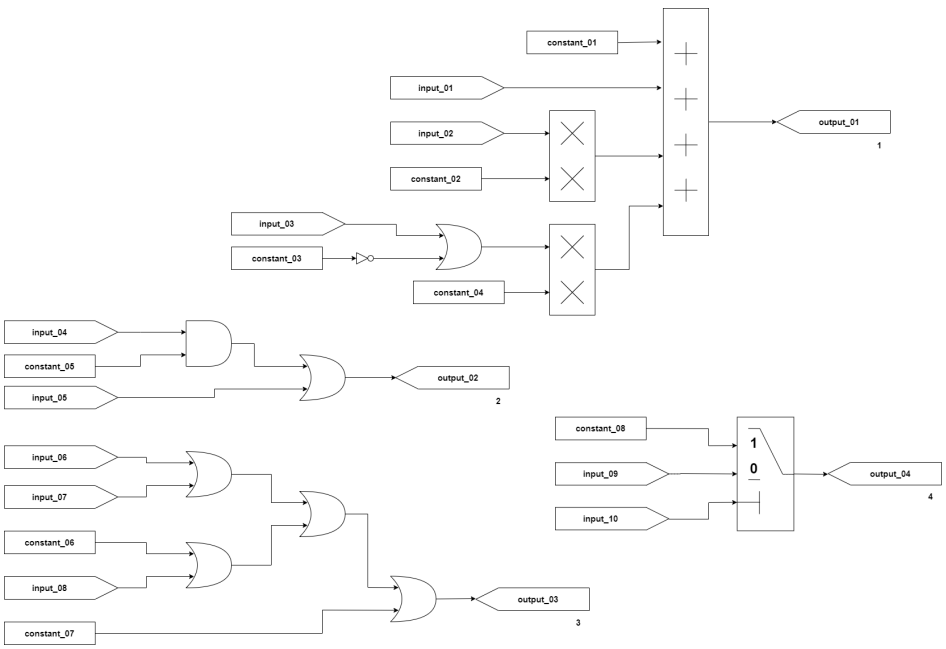
Druhou, a také stěžejní částí celého systému, je právě generátor testovacích scénářů obsahující testovací vektory. Tato podčást celého systému získává na svém vstupu na-

čtený a zpracovaný vstupní soubor v předem definované interní podobě. Generátor testovacích vektorů poté podle sady předem daných pravidel provede jejich transformaci do podoby výsledného souboru ve formátu *.xls.

Vzhledem k tomu, že generování vektorů představuje nejdůležitější součást celého systému je velmi důležité mít jistotu, že je prováděno správným a deklarovaným chováním. Z toho důvodu byla sestavena sada simulačních schémat, která jsou spustitelná a některá z nich se vyskytují i na reálném hardwaru. Pro každé z těchto schémat byla testery vytvořena odpovídající sada výstupních testovacích vektorů, které plně odpovídají danému schématu a jsou zcela správná. Jednotlivé deklarované výsledky jsou porovnávány s těmi, které jsou získány pomocí aplikace scriptu na dané schéma. V ideálním případě jsou výsledky pokaždé stejné. Testovací sada také slouží jako pojistka a ověření toho, že změna ve fungování hlavní logiky po případné úpravě kódu byla i nadále zachována. Kromě sady simulačních schémat byla aplikace otestována i na sadě reálně využívaných schémat, kde byly výsledky porovnány s očekávanými výstupními hodnotami.

V následujících kapitolách budou názorně zobrazeny dvě testovací schémata s jejich zobrazením. Jedná se o jedno jednodušší logické schéma a jedno rozsáhlejší schéma. Obě schémata neodpovídají reálným případům využívaným v praxi a jakákoliv podobnost s nimi je čistě náhodná.

7.2.1 Testovací schéma č. 1



Obr. 7.2 testovací schéma č. 1

Schéma znázorněné na obr. 7.2 představuje základní logický obvod, ve kterém se vyskytují bloky, které nejsou závislé na simulaci času běžícího v systému. Díky tomu je vytváření testovacích scénářů poměrně přímočaré a využívá sady šablon, které jsou patřičně upraveny a doplněny přímo do výstupního souboru. Problém zde může nastat pouze v případech, kdy je k vypočtení výstupního signálu potřeba simulovat kompletní logiku na pozadí, která v některých případech může být poměrně rozsáhlá.

Na obr. 7.3 je zobrazena část výstupního *.xls souboru obsahujícího testovací scénáře. Výsledný soubor se skládá z několika dílčích tabulek, jejichž rozsah zabírá desítky řádků *.xls souboru. Pro ukázku jsou zde uvedeny tabulky testující logický blok OR, z něhož vystupuje output03 v podschématu tři, a tabulka pro testování přepínače v podschématu čtyři, z něhož vystupuje output04.

# OR_2 - output_03										
table			output				input	input	input	input
Test -8	OR_04	Constant_07	output_03		OR_01	OR_02	input_06	input_07	Constant_06	input_08
#	OR-->		output		OR-->	OR-->	input_01	input_02	input_01	input_02
min										
max										
type										
tolerance										
begin										
	0	0	0		0	0	0	0	0	0
	1	0	1		1	0	1	0	0	0
	0	1	1		0	0	0	0	0	0
	1	1	1		1	0	1	0	0	0
end										

# Float Switch2 - output_04				
table	constant		input	output
Test -9	Constant_08	input_09	input_10	output_04
#	input_1	input_0	control	output
min				
max				
type				
tolerance				
begin				
	1	0	1	1
	0	1	1	0
	0	0	1	0
	1	1	1	1
end				

Obr. 7.3 testovací vektory pro bloky OR a přepínač

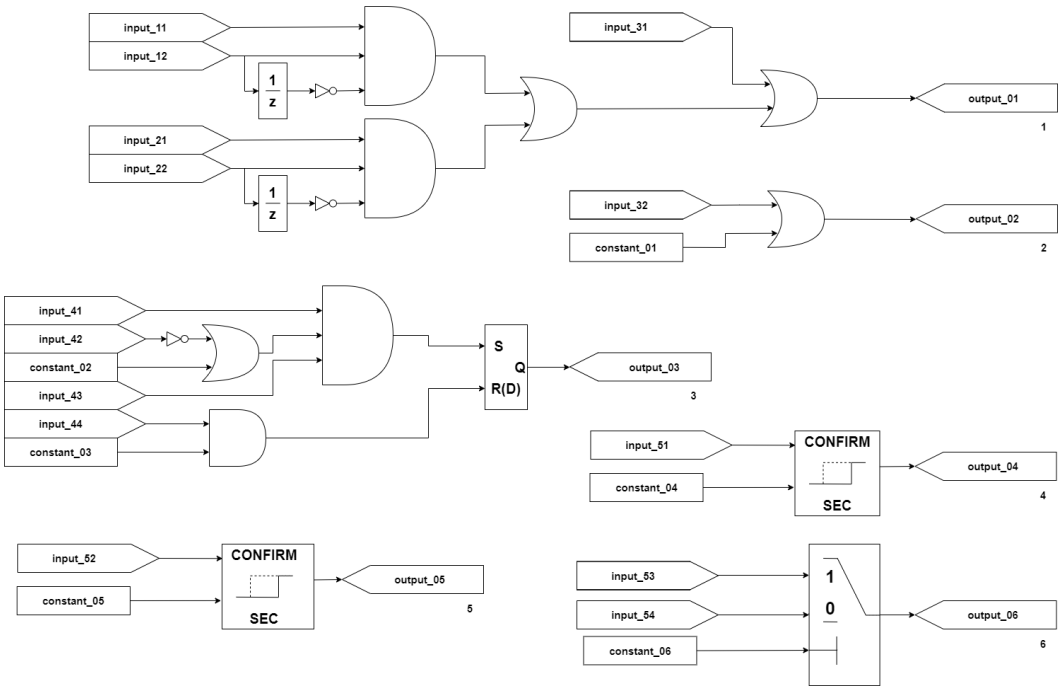
Zhodnocení práce při využití vyvíjené aplikace ve srovnání s prací odvedenou testerem je zhodnoceno v následující tabulce 7.1. Zde je vidět, že i přes značné zrychlení vývoje zapříčiněné využitím vyvíjené aplikace, není možné následující testovací vektory bez dalších kontrol využít při certifikaci software, kde se deklaruje, že je systém bezchybný. Vždy je nutné, aby výsledky práce scriptu byly zkontrolovány ostatními testery jako při klasickém vývoji pomocí testera. K úspoře tedy dochází pouze na straně vývoje SW, ale následná kontrola musí být provedena stejným způsobem jako dříve.

	porozumění schématu	rychlost otestování	Využitelnost při certifikaci	optimalizace testovacích vektorů
tester	1h	7h	ano	ano
SW	0.01h	0.02h	ne	ne

Tab. 7.1 srovnání práce testera a scriptu pro schéma č.1

7.2.2 Testovací schéma č. 2

Schéma znázorněné na obr. 7.4. představuje složitější logický obvod, ve kterém se vyskytují nejen bloky, které jsou závislé na simulaci času běžícího v systému, ale i bloky, které závislé nejsou. Díky tomu je vytváření testovacích scénářů pro takový obvod o něco složitější. Rozdíl spočívá v tom, že doplněné testovací šablony bloků musí být dodatečně upraveny ve větší míře, aby plně otestovaly simulovaný hodinový signál. Pro ostatní bloky zůstává situace podobná jako v přecházejícím testovacím scénáři a problém zde může nastat pouze v případech, kdy je k vypočtení výstupního signálu potřeba simulovat kompletní logika na pozadí, která v některých případech může být poměrně rozsáhlá.



Obr. 7.4 testovací schéma č. 2

Na obr. 7.5 je zobrazena část výstupního *.xls souboru obsahujícího testovací scénáře. Výsledný soubor se skládá z několika dílčích tabulek, jejichž rozsah zabírá desítky řádků *.xls souboru. Pro ukázku jsou zde uvedeny dvě tabulky testující logický resetovací blok, z něhož vystupuje output03 v podschématu tři, a tabulka pro testování confirm sec bloku v podschématu čtyři, z něhož vystupuje output04.

# Latch: Reset dominant - output_03												
table			output	input		input		input	input	input	input	
name 1-17	AND_03	AND_02	output_03	input_41	OR_02	input_43	NOT	Constant_02	input_42	input_44	Constant_03	
#	Set-->	Reset-->	Q	input_1	OR-->	input_3	NOT-->	input_2	input	input_1	input_02	
min			0									
max			1									
type												
tolerance												
begin												
	0	1	0	0	0	0	0	0	1	1	1	
	0	0	0	0	0	0	0	0	1	0	0	
	1	0	1	1	1	1	1	0	0	0	0	
	0	0	1	0	0	0	0	0	1	0	0	
	0	1	0	0	0	0	0	0	1	1	1	
	1	1	0	1	1	1	1	0	0	1	1	
end												

# Confirm sec - output_04				
table		output		output
name 1-18	input_51	Constant_04	Timer	output_04
#	Input	Time Limit	Timer	Output
min		0.5		0
max		0.5		1
type				
tolerance				
begin				
execute 2	0	0.5	0	0
execute 2	1	0.5	0.4	0
execute 2	1	0.5	0.5	1
execute 2	1	0.5	0.5	1
execute 2	0	0.5	0	0
end				

Obr. 7.5 testovací vektory pro časově závislé bloky

Zhodnocení práce při využití vyvíjené aplikace ve srovnání s prací odvedenou testerem je zhodnoceno v následují tabulce 7.2. Tabulka je velmi podobná tabulce u předešlého případu s tím, že jediné místo, kde se navzájem liší spočívá v množství času, které tester potřebuje při zpracování patřičného schématu. V ostatních parametrech jsou tabulky totožné a i dodatečná kontrola ostatními testery zůstává stejná. K úspoře tedy dochází pouze na straně vývoje softwaru, ale následná kontrola musí být provedena stejným způsobem jako dříve.

	porozumění schématu	rychlost otestování	Využitelnost při certifikaci	optimalizace testovacích vektorů
tester	0.75h	6h	ano	ano
SW	0.01h	0.02h	ne	ne

Tab. 7.2 srovnání práce testera a scriptu pro schéma č.2

7.3 Zhodnocení výsledků testování

Obě popsané metody testování prokázaly správnost celé aplikace a jejich znovu aplikovatelnost pro testování každé i sebemenší změny ve zdrojovém kódu představují velkou výhodu. Z pohledu rychlosti testování představují menší Parser testy běh

v rámci sekund ve srovnání s testy pro kontrolu celkové funkčnosti aplikace, kde rychlost je přímo úměrná složitosti testovacího schématu. Nicméně stále se jedná o výrazné zrychlení v porovnání s vytvářením testů za pomoci testera.

Hlavním účelem této práce bylo sestrojení nástroje pro správné generování testovacích scénářů obsahujících jednotlivé testovací vektory, což se na základě verifikování za využití testovacích sad podařilo prokázat. Výsledná míra přesnosti aplikace odpovídá požadovaným předpokladům a na základě ověřovacích testů se dá považovat za dostatečný nástroj, který testerovi pomůže při jeho práci.

V následující tabulkce 7.3 je shrnuta situace popisující pět schémat reálně se vyskytujích v praxi, pro které bylo provedeno zkoumání rychlosti zpracování vstupního schématu a následného vygenerování testovací sady. Jednotlivá schémata byla rozdělena do pěti skupin podle míry jejich složitosti. Poté bylo provedeno jejich zpracování testerem a scriptem a výsledky jejich práce zkontrolovány ostatními testery jako při běžném vývoji testu. Výsledky ukazují, že script lze využít ke zrychlení práce při přípravě testu, ale z důvodů doporučených ze směrnice DO-178 revize B [24] je nutné to vždy zkontrolovat dalšími testery. Kvůli tomu lze nástroj považovat pouze jako pomocníka při testování, ale ne samotnou náhradu testera.

úkol	rozsah složitosti schématu	množství logických bloků	doba testování tester	doba testování script	kontrola testu dalším testery	výsledné zrychlení vývoje testu
1.	jednoduché	25 ks	5 h	2 min	6 h (2x 3 h)	zrychlení o 5 h
2.	mírně rozsáhlejší	42 ks	6,5 h	2 min	8 h (2x 4 h)	zrychlení o 6,5 h
3.	rozsáhlé	64 ks	7,5 h	2,5 min	7 h (2x 3,5 h)	zrychlení o 7,5 h
4.	mírně složité	86 ks	12 h	2,5 min	10 h (2x 5 h)	zrychlení o 12 h
5.	složité	110 ks	16 h	3 min	14 h (2x 7 h)	zrychlení o 16 h

Tab. 7.3 srovnání práce testera a scriptu pro sadu schémat

ZÁVĚR

Cílem této diplomové práce byl rozbor, návrh a následná implementace nástroje, který bude sloužit jako podpůrný program při testování logických programovatelných schémat obvodů využívaných ve společnosti Honeywell. Stanovené cíle a požadavky program splňuje a je plně využitelný při práci, kde sice nemůže plně nahradit práci testera, neboť program není certifikovaný pro praktické využití, ale nabídne alespoň pomocné nástroje, které výslednou práci značně urychlí.

Poznatky z prvního bodu zadání, které se týkají principů modelování SW/FW pro programovatelná hradlová pole, jsou uvedeny v 1 kapitole. Problematiku druhého bodu zadání popisuje 2 kapitola, ve které se nachází základní popis současných možností a přístupů, které se využívají při vývoji a následném testování softwaru. Součástí této kapitoly je také popis různých metodik při samotném testování, jako je například black box, white box či gray box testování. Dále jsou zde popsány jednotlivé etapy životního cyklu vývoje softwaru společně s životním cyklem testování softwaru. Kromě toho je zde uvedena také řada nástrojů sloužící jako pomoc při samotném testování.

Praktickou částí řešení aplikace se nadále zabývají zbylé body zadání, kdy je nejprve nutné podle bodu tři správně definovat vstupní soubory a určit přístupy vedoucí k jejich správnému parsování. Jednotlivé postupy a přístupy jsou definovány ve 3 kapitole. Podle požadavků, které definuje bod čtyři zadání, jsou upřesněny požadavky na vstupní, výstupní soubory a funkcionalitu systému v rámci 4 kapitoly.

Problematiku pátého a šestého bodu zadání popisuje 5 kapitola a 6 kapitola, které uvádí nejprve návrh architektury vyvíjeného systému a následně vývoj a implementaci systému dle sestaveného návrhu. Zbývající bod zadání je popsán v poslední kapitole. Tato kapitola popisuje verifikace implementace pomocí testovacích vzorových příkladů a dále uvádí testování výsledné aplikace ve srovnání s prací testera.

Tato práce poskytuje bližší seznámení s technologií testování logických programovatelných hradel využívaných v praxi na systémech, od kterých se očekává jejich dlouhá životnost a zpětná kompatibilita. Díky tomu bylo možné vyvinout aplikaci od které se očekává, že bude bez problému fungovat na třech různých verzích operačního systému Microsoft Windows.

Implementované řešení umožňuje velké množství úprav, které mu v budoucnu zajistí vyšší funkcionalitu. Příkladem takového rozšíření může být například podpora nových knihoven obsahujících rozšiřující logické bloky či přidání grafického uživatelského rozhraní pro přívětivější ovládání.

SEZNAM POUŽITÉ LITERATURY

[1] *SCHAUMONT, Patrick R. A practical introduction to hardware/software codesign. New York: Springer, c2010. ISBN 978-1441959997.*

[2] *ŠŤASTNÝ, Jakub. FPGA prakticky: realizace číslicových systémů pro programovatelná hradlová pole. Praha: BEN - technická literatura, 2010. ISBN 978-80-7300-261-9.*

[3] *VRBA, Radimír. Digitální obvody. Brno: Vysoké učení technické, 2002. Učební texty vysokých škol. ISBN 80-214-2137-1.*

[4] *PINKER, Jiří a Martin POUPA. Číslicové systémy a jazyk VHDL. Praha: BEN - technická literatura, 2006. ISBN 80-7300-198-5.*

[5] *Pech, Jan. Programovatelné logické obvody* [online]. [cit. 2020-03-15]. Dostupný z `http://fpga.sweb.cz/`.

[6] *Abdullah, Hikmat Al-Hamdani. Design and Implementation of FPGA Based Software Defined Radio Using Simulink HDL Coder. Iraq: Engineering and Technology Journal, 2010. ISSN 1681-6900*

[7] *Vermeulen, Bart Goossens, Kees. Debugging systems-on-chip. New York: Springer, 2014. ISBN 9783319062419*

[8] *KARBAN, Pavel. Výpočty a simulace v programech Matlab a Simulink. Brno: Computer Press, 2006. ISBN 978-80-251-1448-3*

[9] *ROUDENSKÝ, Petr a Anna HAVLÍČKOVÁ. Řízení kvality softwaru: průvodce testováním. Brno: Computer Press, 2013. ISBN 978-80-251-3816-8*

[10] *PATTON, Ron. Testování softwaru. Praha: Computer Press, 2002. Programování. ISBN 80-7226-636-5*

[11] *MYERS, Glenford J., Corey SANDLER a Tom BADGETT. The art of software testing. 3rd ed. Hoboken, N.J.: John Wiley, c2012. ISBN 1118031962.*

[12] *JONES, Capers. Quantifying software: global and industry perspectives. Boca Raton: CRC Press/Taylor Francis Group, 2017. ISBN 9781138033115..*

[13] *N. Minas, M. Marshall, G. Russell A. Yakovlev, FPGA Implementation of an Asynchronous Processor with Both Online and Offline Testing Capabilities, 2008 14th IEEE International Symposium on Asynchronous Circuits and Systems, Newcastle upon Tyne, 2008.*

[14] *WANG, Laung-Terng, Charles E. STROUD a Nur A. TOUBA. System-on-chip test architectures: nanometer design for testability. Boston: Morgan Kaufmann Publishers, c2008. ISBN 9780123739735.*

[15] *HUNT, Brian R., Ronald L. LIPSMAN a J. ROSENBERG. A guide to MATLAB: for beginners and experienced users. 2nd ed. Cambridge, UK: Cambridge University Press, 2006. ISBN 0521850681.*

[16] *Hussein, Qasim. 1 Compiler technique (Part 1). Tikrit University College of computer sciences and mathematics, 2015.DOI: 10.13140/RG.2.1.2848.3921.*

[17] *Deißenböck, Florian, JavaDoc documentation* [online]. [cit. 2020-04-01]. Dostupný z `https://www.cqse.eu/download/conqat/simulink-javadoc/`.

[18] *Yue, Steven, Simulink Mode Parsing Tools* [online]. [cit. 2020-04-02]. Dostupný z `https://github.com/steventen/Simulink-Model-Parsing-Tools`.

[19] *Python Software Foundation, Python 2.7.18 documentation* [online]. [cit. 2020-04-16]. Dostupný z `https://docs.python.org/2.7/`.

[20] *SUMMERFIELD, Mark. Python 3: výukový kurz. Brno: Computer Press, 2010. ISBN 978-80-251-2737-7.*

[21] *LUTZ, Mark. Programming Python. 3rd ed. Sebastopol, CA: O'Reilly, 2006. ISBN 0596009259.*

[22] *ActiveState Software Inc., Komodo IDE 11 Documentation* [online]. [cit. 2020-04-22]. Dostupný z `http://docs.komodoide.com/Manual`.

[23] *Clark, Charlie. Gazoni, Eric. openpyxl Documentation, Release 2.4.9, 2017.*

[24] *United States. Federal Aviation Administration. RTCA, Inc., Document RTCA/DO-178B. [Washington, D.C.] :U.S. Dept. of Transportation, Federal Aviation Administration, 1993. ISBN 9780857291325..*

[25] *CHYTIL, Michal. Automaty a gramatiky. Vyd. 1. Praha: SNTL - Nakladatelství technické literatury, 1984. 331 s..*

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

FPGA	Field Programmable Gate Array
SW	Software
FW	Firmware
ASIC	Application Specific Integrated Circuit
PROM	Programmable Read Only Memory
EPROM	Erasable Programmable Read-Only Memory
PLA	Programmable Logic Array
PAL	Programmable Array Logic
PALASM	PLA Assembler
EEPROM	Electrically Erasable Programmable Read-Only Memory
GAL	Generic Array Logic
PLD	Programmable Logic Device
SPLD	Simple Programmable Logic Device
CPLD	Complex Programmable Logic Device
ALU	Arithmetic Logic Unit
HW	Hardware
HDL	Hardware description language
VHDL	VHSIC Hardware Description Language
ISO	International Organization for Standardization
IEC	International Electrotechnical Commission
HTML	Hypertext Markup Language
GUI	Graphical User Interface
CNRI	Corporation for National Research Initiatives
USA	United States of America
FSF	Free Software Foundation
GNU	GNU's Not UNIX
MS	Microsoft
OS	Operating System
TAF	Test automation framework
SSD	Solid-State Drive
CD	Compact Disc

SEZNAM OBRÁZKŮ

SEZNAM TABULEK