České vysoké učení technické v Praze
Fakulta elektrotechnická

DIPLOMOVÁ PRÁCE

Testování studentských úloh na automaty

Praha, 2010 **Autor: Lukáš Dibelka**

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra řídicí techniky

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Bc. Lukáš Dibelka**

Studijní program: Elektrotechnika a informatika (magisterský), strukturovaný
Obor: Kybernetika a měření, blok KM1 - Řídicí technika

Název tématu: **Testování studentských úloh na automaty**

Pokyny pro vypracování:

1. Prostudujte si možnosti testování konečných automatů.
2. Navrhnete aplikaci vhodnou pro potřeby předmětu Struktury počítačových systémů, která dovede generovat posloupnosti logických signálů a dokáže otestovat navržený automat.
3. Aplikaci realizujte v jazyce C#.NET pro Windows.

Seznam odborné literatury:

Dodá vedoucí práce

Vedoucí: Ing. Richard Šusta, Ph.D.

Platnost zadání: do konce zimního semestru 2009/10

L.S.

prof. Ing. Michael Šebek, DrSc.
vedoucí katedry

doc. Ing. Boris Šimák, CSc.
děkan

V Praze dne 30. 11. 2009

Prohlášení

Prohlašuji, že jsem svou diplomovou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW atd.) uvedené v přiloženém seznamu.

V Praze dne 7.1.2010

podpis

Poděkování

Děkuji především Ing. Richardu Šustovi, Ph.D. za odoborné vedení a pomoc, kterou mi poskytl při vypracování mé diplomové práce.
Dále bych chtěl poděkovat všem, kteří mě podporovali ve studiu.

Abstrakt

Cílem této diplomové práce je zautomatizovat proces kontroly při odevzdávání samostatných studentských prací, což představuje navrhnout kontrolní experimenty pro tyto úlohy a vytvořit testovací soustavu pro jejich provedení. Jako studentské úlohy budou uvažovány hardwarové implementace asynchronních konečných automatů, konkrétně detektorů posloupností. Podstatou navrženého kontrolního experimentu je zjištění jestli studentské implementace přijímají jazyk, který mají uvedený ve svém zadání. V této práci je popsáno jak tyto experimenty fungují a jaké podávají výsledky. Dále je zde uveden popis řídící jednotky testovací soustavy implementující tyto experimenty, generátoru ladících signálů, komunikačního rozhraní se simulačním prostředím MATLAB - Simulink a generátoru zadání studentských úloh. Součástí práce je příloha s návodem na vytvoření Matlabovského modelu konečného automatu studentské úlohy.

Abstract

The aim of this diploma thesis is to automatize checking process during committing student's works. It means to create checking experiments for these works and create testing system for providing experiments. Students works will be hardware implementations of asynchronous finite machines, exactly sequence detectors. The aim of proposed checking experiments is to discover if students implementations of finite machines accept language, which is said in their submission. In this work there is description of these experiments and theirs results. Further this work describes control unit implementing these experiments, generator of debugging logic signals, communication interface with simulator environment MATLAB - Simulink and generator of student's excercises. A part of these diploma thesis is appendix with instructions for creating MATLAB model of finite machine of student work.

Obsah

Seznam obrázků

Kapitola 1

Úvod

V průběhu studia odborných předmětů řeší studenti často mnoho praktických úloh. Musí zpracovat jejich zadání, navrhnout pro ně řešení a poté je realizovat určeným způsobem. V předmětech kolem počítačových systémů se nezřídka jedná o úlohy, které je nutné realizovat fyzicky, tzn. s využitím nějakého hardware. Jednotlivé úlohy slouží k praktickému procvičení probírané látky a ověření znalostí studentů, což se samozřejmě neobejde bez kontroly správnosti řešení, která je úkolem odborných asistentů. Často je to však zdlouhavý a opakující se proces vyžadující zbytečnou alokaci učitele. Tento způsob kontroly navíc nemusí být vždy úplně spolehlivý. Cílem této práce je tuto kontrolu automatizovat, tzn. seznámit se s metodami testování hardwarových obvodů realizujících konečné automaty, navrhnout způsob testování/kontroly hardwarových implementací vybraných studentských úloh předpokládaných složitostí dle 2.3 a vytvořit automatický systém pro jejich kontrolu.

Úlohy, jejichž kontrolou se budu v této práci zabývat, slouží k procvičení návrhu a implementace konečných automatů viz 2.2, tedy úloh zpracovávajících regulární výrazy (více o této problematice je uvedeno kapitole 3). Vyhodnocení správnosti řešení a implementace dané úlohy je testovací experiment, což je proces při kterém se testovaný systém vystaví zkušebním podnětům, zároveň se zaznamenávají odezvy systému na tyto podněty a porovnávají se s správným (požadovaným) chováním. Požadované chování je zaneseno v zadání úlohy (modelu), testovaným systémem bude studentova hardwarová implementace této úlohy. Popis testovacího experimentu, jeho souvislosti s kontrolním experimentem a přehled základních testovacích metod konečně stavových systémů uvádím v kapitole 4. Podrobný popis studentských úloh a testovacích experimentů, které jsem pro ně navrhl, je v kapitole 5.

Na zadání studentských úloh jsou kladeny nároky jako nepřílišná složitost (počet

stavů a přechodů) výsledného konečného automatu a eliminace možností k plagiátorství, tzn. variabilita jednotlivých zadání. Ideálním stavem by bylo "extra" zadání pro každého studenta, což však není většinou možné z důvodů složitosti jejich vymýšlení a navíc by to do procesu kontroly přineslo další komplikace. Učitel by se vždy musel s daným zadáním nejdříve seznámit a až poté by byl schopen rozhodovat o správnosti jeho řešení a implementace. Vhodným kompromisem je tedy rámcové zadání společné pro všechny s tím, že každý student má ve svém konkrétním zadání nějakou individuální variaci. Takto formulované úlohy se navíc dobře sdělují testovací aplikaci a následně testují. Typickou studentskou úlohou je např. asynchronní detektor posloupností, ve které každý student realizuje konečný automat akceptující jiné vstupní posloupnosti. Popis této úlohy a její testovací experiment je uveden v kapitolách 5.1, resp. 5.2.

Pro automatickou kontrolu implementací zadaných úloh jsem navrhl systém viz 5.3, který se skládá z řídící jednotky (PC) a rozhraní pro buzení konkrétního hardware a čtení jeho výstupů 5. Řídící jednotkou 5.3.1 je SW aplikace napsaná v jazyce C# běžící v prostředí Microsoft .NET, která se stará o vytvoření testovacího experimentu, jeho provedení a vyhodnocení výsledků. Vstupem pro testovací experiment je druh studentské práce (např. asynchronní kódový zámek 5.1.1) a popis některých vlastností (např. klíč). Výstupem je rozhodnutí, jestli hardwarová implementace studentské úlohy vyhovuje svému zadání nebo ne. Testovací soustava bude dostupná studentům, takže testovací experiment budou provádět sami a učiteli pouze ukážou, jestli test proběhl úspěšně. Testovací experiment musí být proveditelný na cvičení (nejlépe několikrát), takže nesmí trvat víc než několik minut.

Pro podporu studentů ve fázi návrhu svého řešení jsem vytvořil propojení řídící jednotky se simulačním prostředím MATLAB - Simulink viz 5.4. Student si tak nejdříve může provést testovací experiment na modelu (vytvořený v MATLAB toolboxu StateFlow) svého řešení a ověřit jestli se ubírá správným směrem. Pro tyto účely jsem v řídící jednotce vytvořil modul pro ruční generování vstupních signálů a čtení výstupů (osciloskop/datalogger). Je jím možné generovat ladící vstupní signály (periodické, nebo uživatelsky definované - posloupnosti logických hodnot) a zároveň zobrazovat odezvy na ně viz 5.3.6. Návod na vytvoření matlabovského modelu asynchronních detektorů posloupností a zprovoznění komunikace modelu s řídící jednotkou testovací soustavy uvádím v příloze B.

Proto aby se zadání studentských úloh nemusely vytvářet ručně, jsem vytvořil jejich automatický generátor viz 5.14.

Kapitola 2

Studentské úlohy v oboru logické řízení

V této kapitole uvádím základní motivaci pro tuto diplomovou práci, vysvětluji a definuji pojmy *studentská úloha* a *kontrolní experiment*.

V průběhu studia předmětů, jejichž náplní jsou počítačové systémy, řeší studenti mnoho praktických úloh. Studenti musí zpracovat zadání, navrhnout jejich řešení a poté je implementovat pomocí určených prostředků. Často se jedná o úlohy, které je nutné implementovat fyzicky, tzn. pomocí hardwarových prvků. Touto implementací pak vzniká digitální elektronický obvod.

Účelem těchto úloh je naučit studenty danou problematiku, což se neobejde bez ověření, jestli se tak skutečně stalo. Tato kontrola je úkolem učitelů a odborných asistentů, kteří musí pozorováním chování implementace zadané úlohy rozhodnout, jestli student splnil zadání. Tento způsob kontroly má samozřejmě řadu nevýhod jako :

- zbytečná časová alokace vyučujícího, který by mohl místo kontroly pomáhat studentům s řešením.
- velká časová náročnost spojená s kontrolou a odevzdáním studentských prací.
- nepřesná kontrola s velkou pravděpodobností "přehlédnutí" chyby v řešení a realizaci.

Logickým vyústěním této situace je snaha o zautomatizování procesu kontroly. Mým cílem tedy bylo vytvořit testovací soustavu, která na základě popisu úlohy dokáže vytvořit kontrolní experiment, provést ho a rozhodnout o správnosti řešení úlohy.

2.1 Vlastnosti studentských úloh

Studentská úloha slouží k ověření znalostí studentů a k praktickému procvičení probírané látky. Nejedná se o návrh a řešení nějakého systému za účelem nasazení do praxe nebo pro komerční využití. Nemusí tedy vyhovovat předpisům, normám apod. Další výhodou je, že zadání je možné přizpůsobit mnoha faktorům, např. aby se výsledná práce dobře kontrolovala.

Dalším zjednodušením je zanedbání vlivu času. Součásti, ze kterých studenti sestavují své práce, jsou považovány za ideální. Zanedbávají se tedy časová zpoždění jednotlivých obvodových prvků. Součásti jsou považovány za ideální i z pohledu své funkce. Předpokládá se, že veškeré chyby jsou způsobeny špatným řešením/implementací daného zadání.

Zadání studentských úloh musí eliminovat (nebo alespoň potlačovat) možnosti plagiátorství. Ideálním případem by bylo zvláštní zadání pro každého studenta. Z časových důvodu (příprava, individuální konzultace, kontrola) to však není možné. Kompromisem je společné rámcové zadání pro všechny a částečná modifikace, různá pro každého studenta. Příkladem může být úloha asynchronního kódového zámku 5.1.1, kde všichni studenti řeší jednu úlohu, ale každý student realizuje jinou odemykací posloupnost. Ideální zadání tedy ověří praktické dovednosti studentů, znemožní plagiátorství a umožní snadnou ověřitelnost správnosti jeho řešení.

2.2 Studentské úlohy v oboru logické řízení

Jednou z oblastí studentských úloh v předmětech vyučující počítačové systémy jsou logické systémy.

Definice 2.1: Logickým systémem se rozumí systém, který realizuje logické funkce (přiřazení mezi závislými a nezávislými logickými proměnnými). Logický systém je definován pomocí následujících množin : ►

1. množina logických hodnot {1,0}
2. množina všech vstupních vektorů $X = \{1, 0\}^n$ n-vstupů
3. množina všech výstupních vektorů $Z = \{1, 0\}^m$ m-vstupů

4. množina všech vnitřních stavů $Z = \{1, 0\}^k$ k-stavových proměnných

5. čas - dynamický systém

Logické systémy se dělí do dvou základních skupin: na tzv. kombinační a sekvenční logické systémy. Kombinační obvody jsou charakterizovány tím, že výstupní stav systému závisí pouze na okamžitých stavech (kombinaci) vstupních logických proměnných. Při jejich změně dochází ke změně výstupního stavu se zpožděním, daném jen dobou průchodu signálu přes použité elektronické obvody. Sekvenční logické systémy (systémy s mezipamětí, klopné obvody) generují výstupní stav na základě hodnoty vstupních logických proměnných a předchozí hodnoty výstupu. Výstup těchto obvodů je tedy definován jen tehdy, je-li definována časová posloupnost (sekvence) změn vstupních hodnot. Sekvenční obvody jsou obvodové implementace konečných automatů 3.2. Pojem *studentská úloha* pro potřeby této práce je definován jako :

Definice 2.2: Studentskou úlohou se značí hardwarová implementace konečných automatů. ►

Studentské úlohy nesmí být příliš složité (rozlehlé), mají hlavně ověřit logické uvažování studentů a ne klást velké nároky na jejich realizaci, tzn. konečné automaty studentských úloh nesmí mít příliš velké množství stavů (max. do 10) a přechodů.

2.3 Kontrola správnosti řešení studentské úlohy

Kontrolou se rozumí ověření správnosti studentské úlohy definované v 2.2. Kontrola je tedy experiment ve smyslu následující definice :

Definice 2.3: Kontrolní experiment znamená proces nad studentskou úlohou, jehož výsledkem je rozhodnutí o správnosti hardwarové implementace této úlohy. ►

Kontrolní experiment může provést člověk (učitel), který pozorováním chování implementace toto rozhodnutí učiní, nebo může být proveden automatickými prostředky.

Kapitola 3

Úvod do problematiky jazyků a konečných automatů

V této kapitole vysvětluji některé pojmy z teorie formálních jazyků a konečných automatů, které jsou nutné pro další výklad. Dále zde uvádím souvislost mezi regulárními jazyky (výrazy) a konečnými automaty, některé vlastnosti konečných automatů a vysvětlení pojmů *synchronní/asynchronní automat*. Citace v této kapitole jsem čerpal převážně z [7].

3.1 Formální jazyky a jejich konečná reprezentace

Abeceda je libovolná konečná množina $\mathbb{I}$. Prvky této množiny se nazývají *znaky*. Příkladem abecedy je například množina logických hodnot $\{1, 0\}$. *Slovo* v nad abecedou $\mathbb{I}$ je libovolná konečná posloupnost znaků této abecedy, např. $\{10001\}$. Slovem je i prázdné slovo ε. Se slovy je možné provádět základní operace zřetězení $u.v(uv)$, mocnina $u^n(u^0 = \lambda, u^1 = u, u^{n+1} = u^n.u)$, délka ($|u|$). Množina všech slov nad abecedou $\mathbb{I}$ se značí $\mathbb{I}^*$. *Formální jazyk* L nad abecedou $\mathbb{I}$ je libovolná množina slov nad touto abecedou, např. $\{1, 0, 101, 11\}$ je jazyk nad abecedou $\{1, 0\}$.

Jazyky mohou obsahovat nekonečný počet slov, např. jazyk L nad množinou $\{1, 0\}$, který neomezuje délku jednotlivých slov. Specifikace množiny všech jeho slov se nazývá *konečná reprezentace* (vhodná interpretace jazyka). Konečná reprezentace, které odpovídá právě jeden jazyk se nazývá *formální gramatika*. Pro každý jazyk však gramatika existovat nemusí [7]. V případě konečného jazyka (konečný počet slov) je gra-

matikou výčet těchto slov.

Formální gramatika je struktura pro popis formálního jazyka, umožňuje vygenerování všech slov patřících do daného jazyka. Definice gramatiky, klasifikace gramatik do skupin a jejich vlastnosti jsou uvedeny v [7].

Poznámka: V této diplomové práci se budu zajímat pouze o regulární gramatiky a s nimi spjaté regulární jazyky. Je to z důvodu jejich úzké návaznosti na konečné automaty, jak se ukáže dále. □

3.2 Konečný automat a regulární jazyky

Chování mnoha systémů (zařízení, některé hry, komunikační protokoly atd.) kolem nás lze popsat pomocí konečného množství stavů a přechodů mezi nimi. Např. jednoduchý systém s žárovkou a spínačem, který se chová podle slovního popisu : *"sepnutím spínače se rozsvítí žárovka, rozepnutím spínače žárovka zhasne"*, se nachází ve dvou stavech, mezi kterými přechází na popud nějaké veličiny. Systém, který má konečné množství stavů, se nazývá *konečně stavový systém* a lze jej matematicky popsat *konečným automatem*. Konečný automat M je matematický model počítače s konečnou pamětí, který je schopen reagovat (akceptovat) na určité slova [7]. Množina všech těchto slov, se nazývá *jazyk akceptovaný automatem M*. Definice konečného automatu je :

Definice 3.1: Konečným automatem M se značí uspořádaná pětice

$$M = (S, \mathbb{I}, \delta, s_0, F), \tag{3.1}$$

►

- S je konečná neprázdná množina stavů (stavový prostor),
- $\mathbb{I}$ je konečná neprázdná množina vstupních symbolů (vstupní abeceda),
- δ zobrazení $S \times \mathbb{I} \longrightarrow S$ (přechodová funkce),
- s_0 je *počáteční stav*,
- $F \subseteq S$ je množina *koncových stavů*.

Rošířená přechodová funkce $\delta^* : S \times \mathbb{I} \longrightarrow S$ je definována induktivně vzhledem k délce slova z $\mathbb{I}$:

- $\delta^*(\mathbf{s}, \varepsilon) = \mathbf{s}$ pro každý stav $s \in S$, kde ε je prázdné slovo,
- $\delta^*(\mathbf{s}, w\mathbf{s}) = \mathbf{s} \begin{cases} \delta(\delta^*(\mathbf{s}, w), \mathbf{a}) & je - li\ \delta^*(\mathbf{s}, w) \text{ i } \delta(\delta^*(\mathbf{s}, w), \mathbf{a}) \text{ definováno,} \\ \text{není definováno} & jinak. \end{cases}$

Zápis $\delta^*(\mathbf{s}, w) = p$ znamená, že automat M přejde ze stavu $\mathbf{s}$ "na slovo" w (postupným přečtením slova w znak po znaku) do stavu $\mathbf{p}$. Jazyk přijímaný (akceptovaný) konečným automatem M označovaný jako $L(M)$, je tvořen právě všemi slovy, na které automat z počátečního stavu do některého z koncových stavů:

$$L(M) = \{w \in \mathbb{I}^* | \delta^*(\mathbf{s_0}, w) \in F\} \tag{3.2}$$

Jazyk přijímaný konečným automatem M, označovaný jako $L(M)$, je tedy jazyk všech slov, které jsou tímto automatem přijímány.

Pro komunikaci konečného automatu s okolím slouží jeho vstupy a výstupy. Vstupy slouží pro čtení slov jazyka $L(M)$. Výstupy slouží pro prezentaci stavů (*vnějších*) okolí. Stavu automatu pak odpovídá nějaká *"výstupní veličina"*. V příkladu s žárovkou je to svit žárovky. Takovým konečným automatům se říká *konečný automat s výstupem*. Stavy které nelze z vnějšku rozeznat (nejsou popsané nějakou výstupní veličinou) se označují jako *vnitřní*. Modifikace konečného automatu 3.1 o výstupní veličinu znamená přidání výstupní množiny O (výstupní abeceda) do definice konečného automatu. Závislost mezi aktuálním stavem a výstupy udává *výstupní funkce* λ. Konečný automat tedy popisuje chování konečně stavového systému relací mezi vstypy a výstupy v závislosti na čase. Obvod realizující konečný automat se nazývá sekvenční obvod. Podle způsobů definice výstupní funkce λ se rozšířené definice konečného automatu dělí na :

Definice 3.2: *Mealyho automat* je konečný automat, jehož výstup je spojen s přechody mezi stavy. Formálně je popsán jako konečný automat rozšířený o následující parametry►

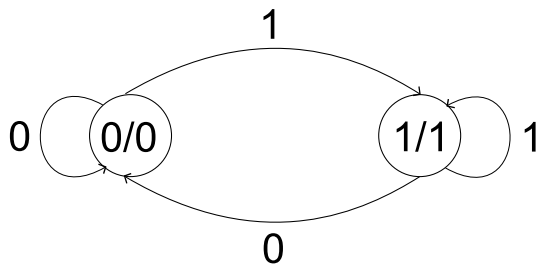
- O^* je výstupní abeceda,
- $\lambda : S \times \mathbb{I} \longrightarrow O^*$ je výstupní funkce.

Definice 3.3: *Mooreho automat* je konečný automat, jehož výstup je určen aktuálním stavem. Formálně je popsán jako konečný automat rozšířený o následující parametry ►

- O^* je výstupní abeceda,
- $\lambda : S \longrightarrow O^*$ je výstupní funkce.

V případě Mealyho automatu je výstup v určitém čase závislý na vstupu a stavu v tomto čase . Moore definuje výstupní funkci jako funkci stavu v daném čase.

Konečný automat M může být názorně reprezentován přechodovým diagramem, což je orientovaný graf, ve kterém vrcholy odpovídají jednotlivým stavům a hrany představují možné přechody mezi stavy. Každá hrana je označena vstupním symbolem, který inicuje přechod. Každý vrchol je označen výstupem v daném stavu (platí pro Moore). Pro systém popsaný výše je přechodový diagram obr. 3.1



Obrázek 3.1: Přechodový diagram systému s žárovkou ovládanou spínačem.

Automat se často reprezentuje pomocí stavové tabulky. Každý řádek odpovídá jednomu stavu (s nějakým označením). Každý sloupec odpovídá jednomu vstupnímu znaku. Každý stav má ve svém řádku uvedený následující stavy, do kterýh se přejde na dané vstupní znaky. Stavová tabulka pro systém popsaný v příkladu je (Moore)

s/y	a	$\bar{a}$
1/0	2	1
2/1	2	1

Obrázek 3.2: Stavová tabulka systému s žárovkou ovládanou spínačem.

3.2.1 Regulární jazyky a výrazy

Regulární jazyk je formální jazyk, jehož slova se rozpoznávají tak, že načtením každého znaku se provede změna stavu v závislosti na předchozím stavu a přečteném znaku. Pokud je výsledkem přečtení celého slova tzv. *koncový stav*, patří toto slovo do tohoto jazyka. Vztah mezi regulárním jazykem a konečným automatem udává Mihyll-Nerodova věta viz [7], která říká, že jazyk rozpoznatelný konečným automatem je regulární.

Regulární výraz je formalismus, který se používá pro specifikaci regulárních jazyků.

Hodnotou regulárního výrazu je určitá množina slov (regulární jazyk) viz [8], přednáška 5. Syntaxe regulárních jazyků je dána definicí 2.57 [7], str. 41. Souvislost mezi regulárními jazyky, výrazy a konečnými automaty uzavírá věta 2.60 [7], str. 43 s následným důkazem.

3.2.2 Některé vlastnosti konečných automatů

Popis některých důležitých vlastností konečných automatů. Podrobnější popis v kapitole 2.1.3 Mihyllova-Nerodova věta [7] a v přednášce č. 2 [8].

Vnější ekvivalence automatů říkáme, že dva konečné automaty A a B jsou *ekvivalentní* z vstup/výstupního pohledu, jestliže přijímají stejný jazyk, tj. $L(A) = L(B)$.

Homomorfizmus Nechť A_1 a A_1 jsou konečné automaty, pak zobrazení $h : S_1 \rightarrow S_1$ je automatovým homomorfismem jestliže :

1. $h(\mathbf{s_1}) = \mathbf{s_2}$ stejné počáteční stavy
2. $h(\delta_1(\mathbf{s}, x)) = \delta_2(h(\mathbf{s}), x))$ stejné přechodové funkce
3. $\mathbf{s} \in F_1 \Leftrightarrow h(\mathbf{s}) \in F_2$ stejné koncové stavy

"Prostý a na" homorfizmus $h : S_1 \rightarrow S_1$ se nazývá izomorfizmus, pro každý stav $\mathbf{p} \in S_1$ se hledá právě jeden stav $\mathbf{q} \in S_2$.

Ekvivalence stavů stavy $\mathbf{p}, \mathbf{q}$ jsou *ekvivalentní* ($\mathbf{p} \sim \mathbf{q}$) jestliže : $\forall w \in I^* \delta^*(\mathbf{p}, w) \in F \Leftrightarrow \delta^*(\mathbf{q}, w) \in F$. Výpočty z ekvivalentních stavů nelze rozlišit.

Dosažitelnost stavů Stav $\mathbf{s}$ je *dosažitelný*, jestliže $\exists w \in I^* \delta^*(\mathbf{s_0}, w) = \mathbf{s}$

Redukovaný automat konečný automat je redukovaný, jestliže :

1. nemá nedosažitelné stavy
2. žádné dva stavy nejsou ekvivalentní.

Minimální konečný automat M rozpoznávající jazyk L je konečný automat s nejmenším počtem stavů, který tento jazyk rozpoznává. Konečný automat je minimální pokud je redukovaný a nemá nedosažitelné stavy [7]. Algoritmus pro nalezení minimálního konečného automatu je uveden v [7], str. 30. Stav automatu představuje pozici ve čteném slově a jejich počet těmto pozicím přímo odpovídá. Souvislost mezi počtem stavů minimálního konečného automatu a jeho přijímaného jazyka je uvedena ve větě 2.29 [7], str. 26.

3.2.3 Synchronní x Asynchronní konečný automat

Synchronní automat má vedle svých "normálních" vstupů (pro čtení přijímaného jazyka) ještě jeden speciální vstup značený jako *hodinový*. Hodinový signál udává okamžiky změn vnitřních stavů. V každé aktivní fázi hodinového signálu automat na základě vstupního znaku a současného stavu přechází do následujícího stavu.

V případě asynchronního automatu dochází k změně vnitřních stavů okamžitě se změnou vstupního znaku. Automat tedy zůstává v nějakém stavu **s** dokud nedojde k změně na vstupu. S tím však přichází určitá návrhová omezení. Asynchronní automat není ze své podstaty schopen reagovat na bezprostředně se opakující znaky. Dále je pro zajištění správné funkce asynchronního automatu nutné dodržovat *fundamentální* režim, který dle [9] zavádí podmínky na vstupní posloupnost :

1. Vstupní posloupnost se liší (mezi dvěma sousedními znaky) pouze v jedné proměnné.
2. Změna vstupní posloupnosti pouze v ustáleném stavu.
3. Každý přechod končí v ustáleném stavu.
4. Změna pouze v jedné vnitřní proměnné během jednoho přechodu.

Vstupem asynchronního automatu může být pouze *fundamentální slovo* definované jako :

Definice 3.4: Jako *fundamentální slovo* w délky n se značí posloupnost znaků z nějaké abecedy I taková, že pro každý znak $\lambda_i \in w$ platí :

- $\lambda_i \neq \lambda_{i+1}$ pro $i = 0$,
- $\lambda_i \neq \lambda_{i-1} \wedge \lambda_i \neq \lambda_{i+1}$ pro všechna $0 < i < n - 1$,
- $\lambda_i \neq \lambda_{i-1}$ pro $i = n - 1$

a zároveň vyhovuje první podmínce fundamentálního režimu. ►

Pro potřebu výkladu v kapitole 5 definuji pojem *množina všch fundamentálních slov nad abecedou tvořenou logickými hodnotami* $\{0, 1\}$ jako :

Definice 3.5: Množinou všech fundamentálních slov $\mathbb{F}$ nad abecedou $\mathbb{I}$ tvořenou znaky $\{0, 1\}$ je myšlena množina všech fundamentálních slov, vytvořených z této abecedy. ►

Kapitola 4

Testování konečně stavových digitálních systémů

V této kapitole vysvětluji pojem *testovací experiment* pro konečně stavový digitální systém, jeho souvislost s kontrolním experimentem, dále zde uvádím přehled základních testovacích metod těchto systémů a podrobnější vysvětlení funkcionálního testování.

Testování je experiment, při kterém je systém vystaven zkušebním podnětům, zároveň je zaznamenávána odezva systému na tyto podněty a porovnávána se správným (požadovaným) chováním systému (odezvou). Správné chování systému je zaneseno v modelu systému (specifikaci). Modelem konečně stavového systému může být jeho konečný automat.

Podstatou každého testu, nezávisle na úrovni abstrakce (informace přenášená v systému), je vygenerování vstupních podnětů, záznam odezvy na ně a její porovnání s požadovanou odezvou.

Projevem nesprávného chování je jiná odezva než požadovaná, což je způsobeno nějakou chybou v systému. Základní dělení chyb je na :

chyby v návrhu nekompletní nebo nekonzistentní specifikace, nesprávné řešení požadovaného chování

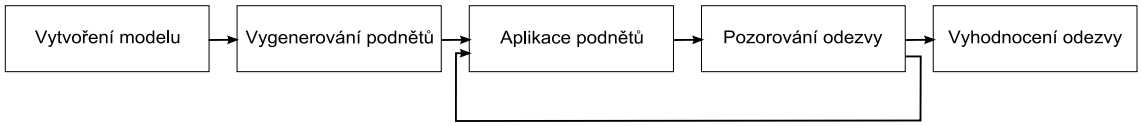
výrobní chyby nesprávné HW komponenty, chybné spoje, nesprávně použitá technologie

fyzické chyby zkraty, nevodivé spojení

Tomuto základnímu rozdělení chyb odpovídá také hrubé dělení jednotlivých testovacích metod.

4.1 Testovací experiment pro konečně stavový systém

Testovací experiment pro konečně stavový systém se skládá z vytvoření modelu systému ve formě konečného automatu, vygenerování testovací sekvence testovacím algoritmem, aplikace sekvence na objekt testu, pozorování odezvy na ni a rozhodnutí o tom, jestli systém testu vyhověl. Jedná se tedy o přímovazební smyčku viz obr. 4.1.



Obrázek 4.1: Blokové schéma testovacího experimentu.

Vygenerování podnětů znamená vypočtení vektoru testovacích podnětů (vstupů) a odezev na ně. Vstupem pro generátor podnětů je co nejpřesnější specifikace testovaného objektu, tzn. jeho matematický model. Postupná aplikace testovací sekvence a sledovnání odezvy na ni je úkolem buď člověka nebo *testovací soustavy.* Pokud tato soustava provede test automaticky, pak se jedná o automatizovanou testovací soustavu definovanou jako :

Definice 4.1: Automatizovaná testovací soustava je označení pro soubor hardwarových a softwarových prostředků, které na základě co nejmenšího množství vstupních informací provede na něm testovací experiment nad testovaným systémem. ▶

Poznámka: Termín testovací experiment v dalším textu této práce je ekvivalentní ke kontrolnímu experimentu dle 2.3, proto budou oba termíny vyjadřovat totéž. □

4.2 Přehled testovacích metod digitálních konečně stavových systémů

Testovací metody závisí na omezeních, kritériích kterými je hodnocen testovaný objekt a na chybách, které v něm chceme nalézt. Rozdělení testů může být například :

- okamžik provádění testu

  - při běžné funkci (online testování)
  - jako vedlejší aktivita (offline testování)

- zdroj testovacích podnětů
  - systém samotný (self-testing)
  - externí zařízení (tester)

- cílová oblast testu
  - chyby návrhu (kontrola návrhu)
  - výrobní chyby
  - fyzické chyby

- testovaný objekt
  - součástka
  - deska
  - komplikovaný systém

- způsob generování podnětů
  - předpřipravené podněty
  - generované během testu (adaptivní)

- dostupnost vstupů/výstupů
  - vnější piny
  - vnitřní piny (in-circuit testování)

Výběr metody pak závisí na zvolených kritériích a na tom, co a kdy chceme pomocí testu o testovaném objektu zjistit.

4.2.1 Funkcionální testování konečně stavových digitálních systémů

Podstatou funkcionálního testovaní konečně stavového systému je určit, jestli chování systému odpovídá konečnému automatu (přechodový diagram, stavová tabulka), kterým

je popsán. Požadované chování systému je tím pevně dáno, funkcionální test pak odhalí jakoukoliv odchylku od tohoto chování. Funkcionální test může být aplikován jak na model (prototyp, simulace), tak na reálný systém (hotová deska, modul apod.). V prvním případě odhalí návrhové chyby, v druhém jak návrhové tak fyzické chyby. V konečném automatu není zanesena informace o implementaci výsledného systému, takže funkcionální test nedokáže odhalit příčinu chyb ani jejich typ.

Specifikace systému je reprezentována konečným automatem A viz 3.2. Problém funkcionálního testování poté přechází na analýzu (pozorování V/V chování) hardwarové implementace tohoto automatu M. Při analýze konečných automatů se řeší tyto typy problémů

Nastavení výchozího stavu každý testovací expmeriment začíná stanovením výchozího stavu. Bez jeho znalosti není možné provést žádnou další analýzu. Tento problém je řešen např. pomocí *nastavovacích*, *synchronizačních* sekvencí nebo *resetovacích* vstupů.

Identifikace stavu a verifikace stavu slouží pro identifikaci a verifikaci jednotlivých stavů. Testovací sekvence řešící tento problém se nazývá *charakteristická sekvence*. Sekvence která ověří konkrétní stav se nazývá *UIO* (unique input/output) sekvence. Existence ani jedné sekvence pro daný konečný automat není zaručena [2].

Srovnávací testování slouží pro stanovení ekvivalence specifikace systému A jeho implementace B. B je "black box", který umožňuje pozorovat pouze své V/V chování. Problém sestrojení testu, který řeší jestli B je správná implementace systému A řeší *srovnávací testování* (v anglické literatuře conformance testing, machine verification popř. fault detection).

Pořadí uvedených problémů odpovídá pořadí akcí, které se provádí při funkcionálním testu. Nejdříve je nutné systém přivést do výchozího stavu, tento stav rozpoznat a ověřit. Poté je možné provést srovnávací test implementace se specifikací.

Problém srovnávacího testování je bez určitých předpokladů a omezení neřešitelný. Pro každou testovací sekvenci je totiž možné setrojit automat, který produkuje stejný výstup jako A, ale není k němu ekvivalentní [2]. Těmito omezeními jsou :

1. Všechny stavy jsou dosažitelné. Při nesplnění by se mohlo stát, že automat B bude ve výchozím stavu, ze kterého nebude možné dosáhnout některé jiné stavy.

2. implementace B systému A se v čase nemění a má stejnou vstupní abecedu jako A

3. Implementace B nemá více stavů než specifikace A. Toto omezení vychází z předpokladu, že chyby v implementaci jsou dvojího druhu, a to *výstupní* chyba, která způsobí špatné hodnoty na výstupu při jednom nebo více přechodů a *přechodová* chyba, která způsobí přechod do špatného následujícího stavu. Testují se tedy pouze stavy, které jsou uvedeny v specifikaci.

4. Konečný automat A je minimální.

5. Konečný automat je plně specifikovaný.

Podrobný popis funkcionálního testování a algoritmů k nalezení sekvencí řešících jednotlivé problémy je uveden v [2].

4.2.2 Shrnutí

Výběr testovací metody záleží na tom, co a kdy je třeba testovat. Testování může být součástí každé fáze procesu návrh-výroba-život jakéhokoliv systému. Vývojový pracovník obdrží zadání jeho funkcionality. Analyzuje ho a navrhne jeho realizaci. Poté může na funkcionálním modelu provést srovnání své implementace se zadáním. Po výrobě již konkrétního obvodu se provede test na detekci výrobních a fyzických chyb. V poslední fázi se provede HIL (Hardware-In-Loop) testování při nasazení v laboratorních podmínkách nebo v praxi. Realizovaný systém také může obsahovat diagnostické mechanizmy pro detekci případných poruch v průběhu svého normálního provozu.

Pro účely testování (kontroly) studentských úloh dle definice 2.2 je použito offline funkcionální testování externím testerem viz 5.3.

Kapitola 5

Testování studentských úloh

V této kapitole uvádím popis kontrolních experimentů, které jsem navrhl pro pro vybrané studentské úlohy odpovídající definici 2.2. Dále popisuji mnou vytvořenou řídící jednotku (testovací aplikace) testovací soustavy dle 4.1 k automatickému provedení kontrolního experimentu pro tyto úlohy.

Na začátku každého úkolu je přesné slovní zadání, z něj se formuluje matematický popis zadání a následně matematický model řešení. Úkolem studenta je přijít na tento model a provést jeho hardwarovou implementaci. Pro testovací aplikaci je vstupem matematický popis (specifikace úlohy), na jehož základě si testovací aplikace model vytvoří kvůli výpočtu testovacích sekvencí.

Student vytváří "prototyp" zadaného úkolu, u kterého se předpokládá maximální spolehlivost použitých hardwarových prvků. Cílem testování je zjištení jestli studentova implementace vyhovuje zadání, tzn. kontrolní experiment ve smyslu 2.3.

Jako studentské úlohy budou uvažovány úlohy ve smyslu definice 2.2. Podstatou testovacího experimentu bude kontrola, jestli konečný automat implementace úlohy B přijíma stejný jazyk L jako konečný automat A daný zadáním úlohy.

5.1 Asynchronní detektor posloupností

V následující kapitole uvádím popis a příklady úloh, které jsou zamýšleny pro výuku logického řízení a pro které jsem navrhl testovací experimenty.

Asynchronní detektor posloupností je stavový systém s konečným množstvím stavů, vstupů a jedním výstupem. Systém přechází ze stavu *"nedetekován"* do stavu *"detekován"*

přijmutím některého slova z přijímaného jazyka L definovaného reguláním výrazem E. Výstupní stavy (zvenku rozpoznatelné) jsou tedy pouze dva. Těmto stavům odpovídá rozdílná výstupní hodnota na jediném výstupu detektoru (automat je typu Moore). Stav *"detekován"* je zároveň koncovým stavem, tvoří tedy množinu koncových stavů F. Vstupní abecedou konečného automatu M je množina znaků, které vzniknou variací logických hodnot $\{1, 0\}$. Množina $\{\{00\}, \{01\}, \{10\}, \{11\}\}$ je vstupní abeceda detektoru se dvěma vstupy. Vstupní abeceda úlohy asynchronního detektoru bude v následujícím výkladu značena $\mathbb{I}$.

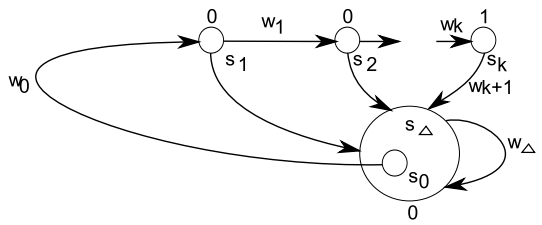
Pojem asynchronní automat byl vysvětlen v odstavci 3.2.3 a pro detektor posloupností představuje omezení v tom, že vstupní slova musí náležet do množiny fundamentálních slov $\mathbb{F}$ viz 3.5.

Podle přijímaného jazyka se úloha asynchronní detektor posloupností dělí na asynchronní kódový zámek s jedním 5.1.1 a více klíči 5.1.2 a asynchronní synchronizátor klíče/ů 5.1.3.

5.1.1 Asynchronní kódový zámek

Konečný automat M asynchronního kódového zámku přijímá regulární jazyk L obsahující slova, které jsou zakončené jedním postfixem w délky k. Postfixem je myšlena část slova, kterým tyto slova končí. Prefix je pak část slova před tímto postfixem. V případě kódových zámků je tento postfix klíč, kterým je zámek možné odemknout.

Asynchronní kódový zámek přejde do koncového stavu v čase $t + k$, pokud zahájí detekci klíče w délky k v čase t. Zjednodušený přechodový diagram konečného automatu M kódového zámku je na obr. 5.1.



Obrázek 5.1: Zjednodušený přechodový diagram automatu asynchronního kódového zámku.

Přechodový diagram obr. 5.1 popisuje konečný automat kódového zámku s klíčem délky k se stavem odemčen ve stavu $\mathbf{s_k}$ (výstup v log. 1). Do tohoto stavu se zámek dostane

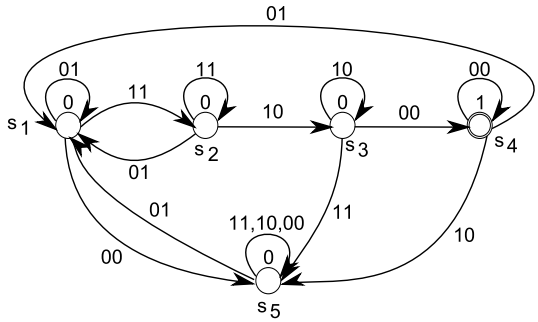
pouze přijmutím slova, které je zakončené posloupností znaků $w_0, w_1, ..., w_k$. Jakákoliv odchylka od této posloupnosti v jakémkoliv stavu způsobí přechod do výchozího stavu $\mathbf{s_\Delta}$, ze kterého vede přechod w_0 (první znak klíče) do stavu $\mathbf{s_1}$ čímž se zahájí nová detekce klíče w. Ze stavu $\mathbf{s_k}$ se jakoukoliv změnou na vstupu dostane zámek zpět do stavu zamčen.

Příklad 5.1 (Asynchronní kódový zámek): Nalezněte konečný automat pro asynchronní kódový zámek se dvěma vstupy A,B a s klíčem $w = \{01, 11\,10, 00\}$ délky $k = 4$. Kódový zámek musí přejít do koncového stavu v čase $t + k$, pouze pokud zahájil příjem klíče v čase t.

Řešení: Asynchronní zámek má 2 vstupy, tzn. $m = 2$. Vstupní abecedou v tomto příkladě jsou všechny kombinace logických hodnot na vstupech, tzn. $I = \{00, 01, 10, 11\}$, klíč je délky $k = 4$. Množina koncových stavů F je tvořena stavem $\mathbf{s_4}$, tedy $F = \{\mathbf{s_4}\}$. Konečný automat M musí přijímat jazyk definovaný regulárním výrazem :

$$E = ((\{11\} + \{10\} + \{01\} + \{00\})^*\{01, 11, 10, 00\}) \qquad \checkmark$$

Přechodový diagram konečného automatu řešící tuto úlohu je :



Obrázek 5.2: Přechodový diagram asynchronního kódového zámku s jedním klíčem.

Z přechodového diagramu lze vidět, že automat přejde do koncového stavu $\mathbf{s_4}$ (odemknut) pouze ze stavu $\mathbf{s_1}$ po rozpoznání slova zakončeného klíčem w. Jiné slovo akceptováno není a nezpůsobí přechod do koncového stavu $\mathbf{s_4}$.

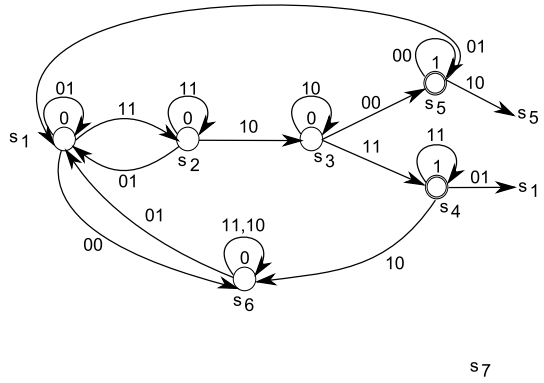
5.1.2 Asynchronní kódový zámek s více klíči

Nevýhodou jednoduchého kódového zámku je nízká variabilita tvaru přechodového diagramu. Zvýšením délky klíče přibudou stavy pro jeho přijmutí a nové přechody směřující

do výchozího stavu, ale tvar přechodového diagramu se razantně nezmění. Přidání jednoho vstupu (zvýšení počtu znaků vstupní abecedy) dojde k výraznému nárůstu počtu přechodů do výchozího stavu, ale prinicpiální tvar přechodového diagramu se zase nezmění. Zadání úlohy jednoduchého kódového zámku, tedy není ideálním zadáním z pohledu plagiátorství a spolupráce mezi studenty. Změnu v přechodovém diagramu způsobí zavedení více klíčů (postfixů) vedoucí k odemčení zámku.

Příklad 5.2 (Asynchronní kódový zámek s více klíči): Nalezněte konečný automat pro asynchronní kódový zámek se dvěma vstupy A,B a s klíči $W = \{\{01, 11, 10, 00\}, \{01, 11, 10, 11\}\}$ délky $k = 4$. Kódový zámek musí přejít do koncového stavu v čase $t + k$, pouze pokud zahájil příjem klíče z W v čase t.

Řešení: Konečný automat přijímá jazyk popsaný regulárním výrazem $E = ((\{11\} + \{10\} + \{01\} + \{00\})^*\{01, 11, 10\}(\{11\} + \{00\}))$. Jeho přechodový diagram je na obr. 5.3.



Obrázek 5.3: Přechodový diagram asynchronního kódového zámku s více klíči.

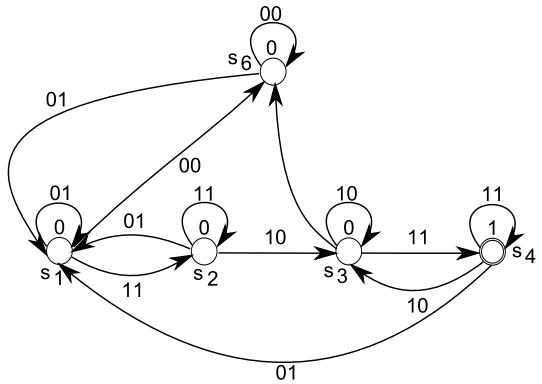
5.1.3 Asynchronní synchronizátor posloupností

Definice 5.1: *Synchronizace* vstupní posloupnosti/slova w znamená přechod konečného automatu do koncového stavu přečtením slova w délky k v čase t, pokud automat přijmul slovo w v intervalu $(t - k, t)$. ▶

Kódový zámek přejde do koncového stavu přečtením slov zakončených určitým postfixem/y w délky k v čase $t + k$, pouze pokud byla detekce zahájena přesně v čase t, takže vstupní slova nesynchronizují dle 5.1. Asynchronní synchronizátor čte stejná vstupní slova, ale nezávisle na čase začátku čtení.

Příklad 5.3 (Asynchronní synchronizátor posloupností): Nalezněte konečný automat, který provádí sychronizaci slova zakončeného posloupností $w = \{11, 10, 11\}$.

Řešení: Konečný automat přijímá jazyk popsaný regulárním výrazem $E = ((\{11\} + \{10\} + \{01\} + \{00\})^*\{11, 10, 11\})$. Není však omezen na začátek čtení postfixu. Přechodový diagram automatu je na obr. 5.4.



Obrázek 5.4: Přechodový diagram asynchronního synchronizátoru posloupnosti.

5.1.4 Počet stavů konečného automatu detektoru posloupností

Počet stavů minimálního konečného automatu je roven počtu tříd, do kterých konečný automat rozkládá přijímaný jazyk viz 3.2.2. K určení počtu stavů konečného automatu detektorů posloupností, tedy stačí určit počet tříd, do kterých rozkládají přijímané slova jejich regulární jazyky.

5.1.4.1 Asynchronní kódový zámek

Hledá ve vstupním slově shodu na slovo zakončené postfixem w délky k. Provádí tedy rozklad do k tříd (stavů), které odpovídají jednotlivým pozicím znaků v postfixu w a do jedné třídy (stavů) kam spadají slova jiné než zakončená w. Počet stavů v závislosti na délce tohoto postfixu je $n = k + 1$.

5.1.4.2 Asynchronní kódový zámek s několika klíči

Je detektor několika slov z abecedy, která je tvořena znaky z abecedy $\mathbb{I}$. Regulární výraz generující tyto slova je tak omezen pouze maximem slov, které je schopen z dané abecedy

vygenerovat (maximální počet postfixů). Díky fundamentálnosti slov lze z abecedy jejichž znaky jsou kódovány m bity vygenerovat $p = m^{k+1}$ postfixů délky k. Automat který by měl přijímat jazyk obsahující všechny slova zakončené těmito klíči by tedy měl maximálně $n = k \cdot p + 1$ stavů.

5.1.4.3 Asynchronní synchronizátor posloupností

Tato úloha se od asynchronního kódového zámku liší rozdílem v počátečním čase příjmu postfixu w, což však nevede na změnu počtu stavů oproti této úloze. Toto platí i pro případ synchronizace více postfixů.

5.2 Testovací experiment pro asynchronní detektor posloupností

Podstatou navrženého testovacího experimentu pro úlohy asynchronních detektorů je zjistit, jestli studentské hardwarové implementace jejich konečných automatů přijímají jazyk L, který mají uvedený ve svém zadání. Testovacím experimentem se tedy musí určit jestli implementovaný konečný automat :

1. přijímá jazyk L.
2. nepřijímá žádný jiný jazyk.

Navržený testovací experiment je tedy test na ekvivalenci vstup/výstup chování konečných automatů dle 3.2.2.

Srovnávací metoda [2] pro konečné automaty vyžaduje splnění všech 5 podmínek uvedených v 4.2.1. Z přechodových diagramů konečných automatů jednotlivých úloh lze vidět, že podmínka dosažitelnosti všech stavů je splněna. Podmínka časové neměnnosti se předpokládá. Podmínka na omezení počtu stavů říká, že testovací metoda vychází pouze ze známého a předpokládaného stavového prostoru daného konečným automatem specifikace úlohy. Podmínka minimality konečného automatu A je omezením pro testovací aplikaci, která si musí pro vygenerování testu vytvořit právě minimální konečný automat podle zadání dané úlohy. Problém je s částečnou specifikací přechodového diagramu. Tato podmínka nemůže být splněna kvůli omezením, které jsou kladeny na asynchronní konečné automaty viz 3.2.3. Pro takto specifikovaný systém nelze nalézet sekvence řešící

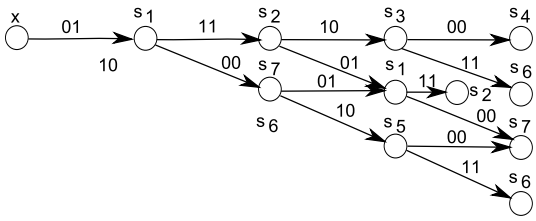
jednotlivé problémy uvedené v 4.2.1. Proto jsem navrhl vlastní testovací experimenty pro úlohy asynchronních detektorů posloupností jejihž popis uvádím dále.

Testovací experiment, který jsem navrhl pro výše uvedené úlohy se skládá ze tří částí. V první části testu se zjišťuje jestli studentská úloha nepřijímá slova zakončené jinými postfixy než definovanými v zadání. V druhé části se určí jestli úloha přijímá postfixy uvozené prefixy generovanými z abecedy $\mathbb{I}$ a v třetí části je ověřena správná ne/synchronizace postfixů.

Testovací experiment je vždy ukončen aplikací slova, po kterém by měl automat přejít do koncového stavu. Čímž se vyloučí opomenutí zacyklení automatu v některém z jeho stavů.

5.2.1 Testování variací kolem postfixů slov

Test variací kolem postfixů slouží pro ověření jestli studentská úloha přijímá jiné slova než zakončené postfixy dle svého přijímaného jazyka. V případě variace kolem jednoho postfixu je variací myšlena množina všech slov V, které začínají stejným znakem jako w a mají stejnou délku. Variace V pro postfix $w = \{01, 11, 10, 00\}$ je zobrazena na obr. 5.5. Tímto testem se odhalí, jestli konečný automat nepříjímá i slova zakončená jinými postfixy než zadanými.



Obrázek 5.5: Variace kolem postfixu (klíče) z příkladu 5.1

Počet slov této množiny je omezen délkou postfixu $|w| = k$, počtem vstupů m (počtem znaků vstupní abecedy) a podmínkami asynchronnosti slov 3.4.

Algoritmus testování celou množinou variací kolem postfixu je následující :

1. výpočet množiny variací V
2. aplikace slova končícího postfixem w
3. pokud není akceptováno, tak konec (úloha nevyhověla)

4. pokud je akceptováno jsme v koncovém stavu, $i = 0$
5. aplikace i-tého slova z V
6. pokud je akceptováno, tak konec (úloha nevyhověla)
7. pokud není, tak pro $i++ < |V|$ návrat na bod 5. a pro $i = |V|$ jdi dále
8. aplikace slova končícího postfixem w (test na zacyklení)
9. pokud není akceptováno, tak konec (úloha nevyhověla), pokud je tak konec (úloha vyhověla)

Délka d výsledné testovací posloupnosti v závislosti na délce postfixu k a počtu vstupů m je :

$$d =\leq m^{k-1} \cdot k + m^{k-1} \cdot (m-1)$$

První člen tohoto polynomu vyjadřuje součet délek všech slov v z množiny variací V. Pokud některé slovo v končilo takovým znakem, že po něm není možné (neplatila fundamentálnost slov) na vstup automatu přivést slovo začínající znakem w_1, pak je nutné mezi tyto dvě slova vložit *"mezislovo"*, které zaručí neporušení podmínek fundamentálnosti vstupních slov. Maximální možnou délku těchto mezislov udává druhý člen polynomu. Tabulka 5.1 ukazuje délku testu v závislosti na parametrech k a m.

	k=2	k=3	k=4	k=5	k=6	k=7	k=8	...
m=2	6	16	40	96	224	512	1152	...
m=3	12	45	162	567	1944	6561	21870	...
m=4	20	96	448	2048	9216	40960	180224	...
m=5	30	175	1000	5625	31250	171875	937500	...

Tabulka 5.1: Závislost délky testu variací kolem jednoho postfixu na počtu vstupů m a délce postfixu k.

V případě asynchronního kódového zámku s více klíči končí přijímaná slova více postfixy, kterých může být maximálně $p = m^2 \cdot m^{k-1}$ viz 5.1.4.2. V testovacím experimentu se pak algoritmus pro variační testování opakuje pro každý postfix, takže maximálně p-krát, s tím že opakující se variace se aplikují pouze jednou.

5.2.2 Testování vnoření postfixů slov

V druhé části testu se musí zjistit, jestli studentská úloha přijímá vnořené postfixy. Tzn. jestli přijímá slova začínající prefixy generovanými z abecedy $\mathbb{I}$ a zakončenými definovanými postfixy.

Asynchronní detektory posloupností popsané v 5.1 přijímají slova, která musí být vždy zakončena určitým postfixem w, popř. množinou postfixů W. Prefix těchto slov může být libovolně dlouhý, tzn. mohou se v něm opakovat stejné sekvence znaků. Např. konečný automat asynchronního synchronizátoru, jehož jazyk je popsaný regulárním výrazem $(a+b+c+d)^*w$, kde $w = w_1w_2w_3 = cba$ přijímá libovolně dlouhé slovo zakončené trojicí znaků cba, např. $abcba$ nebo $abcabcba$. Konečné automaty detektorů rozkládají tyto slova do tříd, jejihž množství odpovídá počtu jejich stavů viz 3.2.2. Proto pro ověření přijímaných prefixů nemá smysl uvažovat testovací sekvence s prefixy, které jsou delší než počet stavů těchto konečných automatů, protože by se automat v průběhu čtení prefixu dostal do některého ze stavů, ve kterém již jednou byl.

Maximální délka uvažovaných prefixů tedy odpovídá počtu stavů n (v případě jednoho postfixu $n = k+1$) konečného automatu. Množina prefixů P pro tento test obsahuje všechny fundamentální prefixy délky 1 až n. V závislosti na počtu stavů n konečného automatu a počtu vstupů m ($\sqrt{znaku}$) je celkový počet fundamentálních prefixů p délky max. n roven výrazu :

$$p = \sum_{i=1}^{n} m^{i+1}$$

Testovací posloupnost pro detektor posloupností s l postfixy délky k, která pokryje prostor všech prefixů bude mít celkovou délku :

$$d = \sum_{i=1}^{n} m^{i+1} \cdot k \cdot l$$

Pořadí aplikace prefixů je voleno tak, aby nebylo nutné vkládat do testovací sekvence mezislova jako v případě testu variací kolem postfixů. Algoritmus implementující test vnoření postfixů pro celý prostor prefixů P je :

1. výpočet množiny prefixů P, $i = 0$
2. aplikace slova končícího postfixem $w \in W$
3. pokud není akceptováno, tak konec (úloha nevyhověla)

4. pokud je akceptováno jsme v koncovém stavu, $i = 0$
5. aplikace i-tého prefixu + nějakého postfixu $w \in W$
6. pokud není slovo přijímano, tak konec (úloha nevyhověla)
7. pokud je přijmuto, tak pro $i++ < |P|$ návrat na bod 5. a pro $i = |P|$ konec (úloha vyhověla)

Následující tabulka ukazuje délky testovacích sekvencí pro jeden postfix v závislosti na jeho délce k a počtu vstupů m.

	k=2	k=3	k=4	k=5	k=6	k=7	...
m=2	56	180	496	1260	3048	7140	...
m=3	234	1080	4356	16380	59022	206640	...
m=4	672	4080	21824	109200	524256	2446640	...
m=5	1550	11700	78100	488250	2929650	17089800	...

Tabulka 5.2: Závislost délky testu na vnoření jednoho postfixu v závislosti na počtu vstupů m a délce postfixu k.

Z tabulky 5.3 lze vidět, že délka testovací posloupnosti prudce roste jak s roustoucím množstvím znaků abecedy, tak s délkou přijímaného postfixu. Zavedením náhodného faktoru x je možné tento testovací soubor zkrátit na $p = \frac{1}{x}$, tzn. test pak neprojde celý prostor prefixů, ale vybere z něj pomocí generátoru náhodných čísel náhodné prefixy, které poté aplikuje na studentskou úlohu. S rostoucím náhodným faktorem x, tedy x-krát klesá pravděpodobnost odhalení chyby vnoření postfixu. Vliv náhodného faktoru je ukázán v kapitole 5.2.5.

5.2.3 Test synchronizace postfixů

Rozdíl v synchronizaci 5.1 mezi zámkem a synchronizátorem by se projevil například při čtení slova $01, 00, 01, 00, 01$ automatem, který přijímá postfix $01, 00, 01$. Výstupem zámku by byla posloupnost 00100, kdežto synchonizátoru 00101. Pro ověření správné synchronizace jsem navrhl následující algoritmus.

Množina obsahující všechny možnosti synchronizace postfixu w je dána množinou synchronizačních slov S :

$$S = \begin{cases} w_1 \dots w_i x w_j \dots w_k, & 0 < i < k \wedge 1 < j \leq i+1, \forall x \neq w_{i+1}; \\ w_1 \dots w_k w_j \dots w_k, w_1 \dots w_k x w_j \dots w_k & i = k \wedge 1 < j < k+1, \forall x \neq w_k. \end{cases}$$

Algoritmus pro ověření synchronizace postfixů je :

1. výpočet množiny synchronizačních slov S
2. aplikace slova končícího postfixem w
3. pokud není akceptováno, tak konec (úloha nevyhověla)
4. pokud je akceptováno jsme v koncovém stavu, $i = 0$
5. aplikace i-tého slova s_i z S
6. pokud nepřijmuto slovo $w \in s$, tak konec (úloha nevyhověla)
7. pokud přijmuto, tak pro $i++ < |S|$ návrat na bod 5. a pro $i = |S|$ konec (úloha vyhověla)

Operace přijímání slov $s \in S$ v následujícím algoritmu je závislá na povaze úlohy. Zámek nesmí provádět synchronizaci, kdežto synchronizátor musí. Všechny slova množiny S musí náležet do množiny fundamentálních slov $\mathbb{F}$ dle 3.4. Ty které tuto podmínku nesplňují nejsou v testovacím experimentu uvažovány. Pro synchronizátor více postfixů se test opakuje pro všechny $w \in W$.

Délka výsledné testovací sekvence v závislosti na počtu vstupů m a délce postfixu k je :

$$p = (m-1) \cdot \sum_{i=1}^{k-1} \sum_{j=k-i}^{k-1} (i+1+j) + m \cdot \sum_{j=1}^{k-1} (k+j+1) + \sum_{j=1}^{k-1} (k+j)$$

Následující tabulka ukazuje délky testovacích sekvencí pro jeden postfix v závislosti na jeho délce k a počtu vstupů m.

	k=2	k=3	k=4	k=5	k=6	k=7	...
m=2	14	44	94	168	270	404	...
m=3	21	68	149	272	445	676	...
m=4	28	92	204	376	620	948	...
m=5	35	116	259	480	795	1220	...

Tabulka 5.3: Závislost délky testu synchronizace postfixu v závislosti na počtu vstupů m a délce postfixu k.

5.2.4 Složitost předchozích testovacích metod

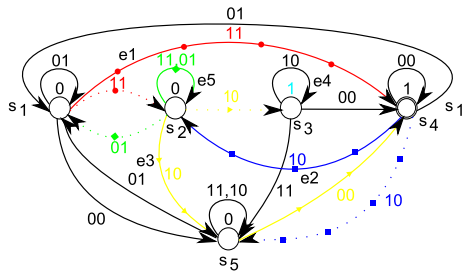
Výstupem předchozích algoritmů jsou posloupnosti vstupních podnětů a odezev na ně. Složitost a náročnost testovací metody je vyjádřena délkou těchto posloupností. Časová náročnost testovacího experimentu závisí na rychlosti aplikace testovacích sekvencí na testovaný objekt. Závisí tedy na použitých hardwarových prvcích, ze kterých je testovaný objekt realizován a na rychlosti generátoru testovacích sekvencí. Časové vyjádření délky testu je $T = d \cdot \mu[s]$, kde μ je doba v sekundách potřebná pro vygenerování znaku a jeho přečtení studentskou úlohou a d vyjadřuje počet znaků v testovací posloupnosti. Následující tabulka ukazuje časovou celého navrženého testovacího experimentu pro $\mu = 20ms$.

	k=2	k=3	k=4	k=5	k=6	k=7	...
m=2	1.50	4.78	12.58	30.46	70.80	161.08	...
m=3	5.36	23.86	93.28	344.34	1227.00	4276.00	...
m=4	14.36	85.34	449.98	2232.00	10682	49771.00	...
m=5	32.30	239.82	1587.00	9887.00	59234.00	345260.00	...

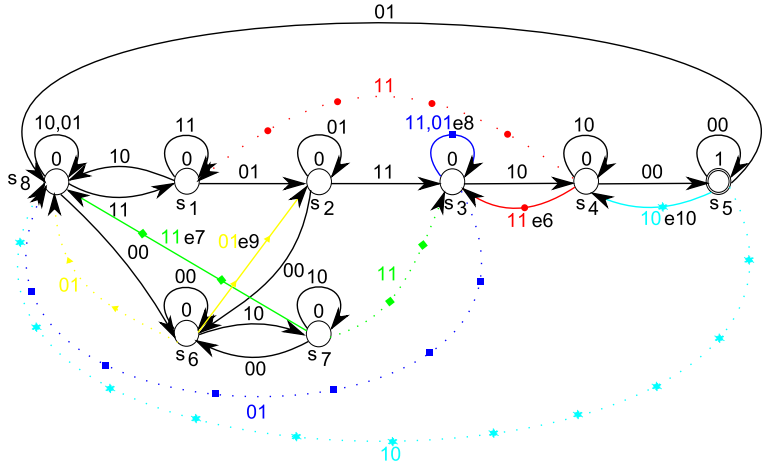
Tabulka 5.4: Délka (v sekundách) celého testu (bez náhodného faktoru) pro jeden postfix v závislosti na počtu vstupů m a délce postfixu k.

5.2.5 Výsledky testovacích experimentů

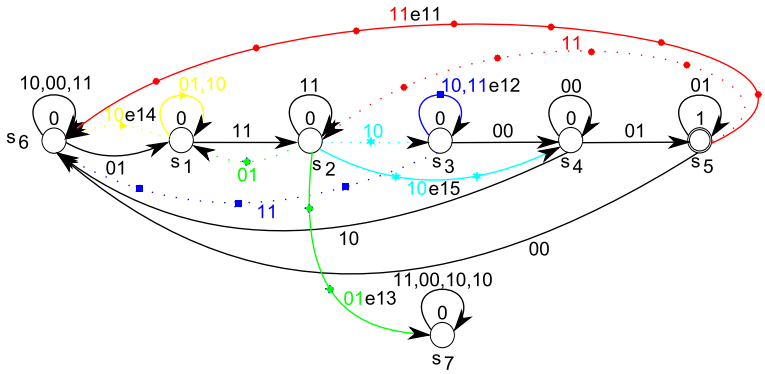
Pro prezenteci funkčnosti navržených experimentů jsem provedl simulace s Matlabovskými modely studentských úloh. V těchto modelech jsou úmyslně zaneseny různé chyby (voleny náhodně), aby byla vidět robustnost experimentů. Přechodové diagramy chybových konečných automatů jsou uvedeny na obr. 5.6, obr. 5.7, obr. 5.8 a 5.9. Tečkovanou čarou jsou značeny správné přechody, tučnou barevnou (značenou) čarou jsou chybně vedené přechody. Výsledky experimentů jsou v tabulce 5.5. Významy písmen v tabulkách této kapitoly jsou V-variační test, S-synchronizační test, P-test na vnoření posftixů.



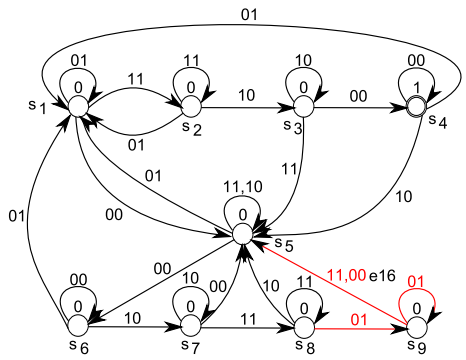
Obrázek 5.6: Asynchronní kódový zámek s jedním klíčem $\{01, 11, 10, 00\}$



Obrázek 5.7: Asynchronní kódový zámek s klíči $\{11, 01, 11, 10, 00\}$ a $\{00, 10, 11, 10, 00\}$



Obrázek 5.8: Asynchronní synchronizátor s klíčem $\{01, 11, 10, 00, 01\}$



Obrázek 5.9: Asynchronní kódový zámek s jedním klíčem $\{01, 11, 10, 00\}$ nepřijímající prefix $\{00, 10, 11\}$

Chyba č.	Odhalení/Délka testu	Test	Chyba č.	Odhalení/Délka testu	Test
e1	7/561	V	e9	140/2829	V
e2	98/594	S	e10	16/2829	V
e3	10/594	S	e11	152/1407	S
e4	16/594	S	e12	49/1407	S
e5	12/574	V	e13	65/1418	V
e6	35/2829	V	e14	41/1417	V
e7	14/2827	V	e15	10/1419	V
e8	23/2829	V	e16	291/586	P

Tabulka 5.5: Výsledky testovacách experimentů pro chybné implementace úloh asynchronních detektorů 5.1.

Z výsledků testovacích experimentů lze vidět, že nejvíce chyb je odhaleno variačním testem. Proto je výhodné provést test touto metodou jako první. Po variačním testu následuje test na synchronizaci a pak až test na vnoření postfixů. Chyba č.16 byla odhalena až testem na vnoření postfixů, proto jsem provedl 25 opakování tohoto testu pro ukázku vlivu náhodného faktoru na nalezení této chyby. Výsledky jsou uvedeny v tab. 5.6.

Chyba č.	Pst. odhalení [%]/Průměrná délka testu	Náhodný faktor
e16	88/354	2
e16	72/275	3
e16	62/247	4
e16	43/218	5
e16	35/196	6
e16	25/182	7

Tabulka 5.6: Pravděpodobnost nalezení chyby č. 16 testem na vnoření postfixu pro 25 opakování tohoto testu.

V následující tabulce 5.7 uvádím výsledky testovacích experimentů při testu konečného automatu kódového zámku s klíčem $w = \{01, 11, 10, 00\}$. Testovací aplikaci však byly "podsunuty" automaty přijímající postfixy dle prvního sloupce tabulky 5.7.

Přijímané postfixy	Odhalení/Délka testu	Test
$\{(10 + 01), 11, 10, 00\}$	93/468	S
$\{01, 11, (10 + 01), 00\}$	61/465	S
$\{01, 11, 10, (00 + 11)\}$	44/471	V

Tabulka 5.7: Výsledky testovacích experimentů pro automaty, které přijímají jiné postfixy než $\{01, 11, 10, 00\}$.

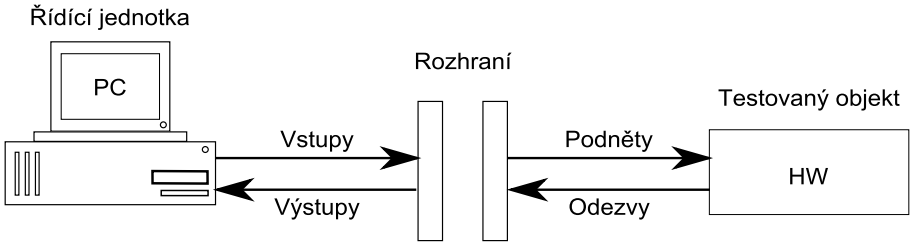
Při testu synchronizátoru klíče $\{00, 01, 00, 10, 00, 01\}$, místo kterého byl podsunut as. detektor tohoto klíče, byla chyba odhalena variačním testem (po 155 krocích) i synchronizačním testem (po 30 krocích). Variačním testem byla chyba odhalena proto, že odezva na testovací posloupnost je vypočtena až po vygenerování celého testu, tímpádem díky vložení mezislova mezi dvě vedlejší variace mohlo vzniknout slovo, které musí sychnronizátor synchronizovat, více viz 5.1.3.

5.3 Testovací soustava pro automatickou kontrolu studentských úloh

Pro provedení kontrolních experimentů jsem navrhl testovací soustavu dle 4.1 pro studentské úlohy popsané v kapitole 5.1. Testovací soustava pro automatické testování se skládá z řídícího členu a z rozhraní pro přístup ke konkrétnímu hardware. V době řešení této diplomové práce ještě nebyl znám typ hardware, na kterém budou studenti realizovat své úlohy, a proto jsem vytvořil pouze řídící jednotku této soustavy. Jako jistou náhradu jsem vytvořil rozhraní pro externí simulační software MATLAB.

Řídící jednotka je softwarová formulářová aplikace běžící na PC v prostředí Windows. Jejím úkolem je na základě zadání studentské úlohy vytvořit a provést testovací experiment, tzn. vypočíst vstupní testovací soubor, aplikovat jej na studentskou úlohu, zachycovat a porovnávat odezvu testované úlohy s požadovanou odezvou. Výstupem je rozhodnutí o správnosti implementace zadání.

Úkolem rozhraní je transformovat vstupní podněty z formátu, se kterým pracuje řídící jednotka (logické hodnoty v paměti PC), na logické úrovně schopné buzení hardware. Rozhraní musí také provádět zpětnou transformaci z logických úrovní na výstupech hardware do logických hodnot v paměti počítače. Schématický obrázek na obr. 5.10



Obrázek 5.10: Schématický popis testovací soustavy.

5.3.1 Řídící jednotka testovací soustavy

V řídící jednotce je ukryta veškerá logika a řízení experimentu. Roli řídící jednotky testovacího systému obr. 5.10 plní aplikace běžící na PC v operačním systému Windows napsaná v jazyce C#. Spuštění vyžaduje instalaci .NET Framework 2.0 na daném PC. Jedná se o Windows Form aplikaci s grafickým uživatelským rozhraním.

Aplikace je sestavená z několik assembly (projektů). Všechny projekty jsou součástí

jedné solution `StudentWorksTester` s nastavenými dependencies. Touto solution je vytvořen jednoduchý build systém. Rozdělením do jednotlivých assembly (knihoven) a objektů uvnitř těchto assembly je zajištěna modularita výsledného programu.

5.3.2 Struktura řídící jednotky

Struktura solution *StudentWorksTester* je :

Wrapper C++ knihovna, přístup k API funkcím karty Advantech (nepoužito)

External C++ sdílená knihovna pro komunikaci s externím simulačním SW

Utils C# knihovna s pomocnými funkcemi, zobrazovacími userControly a C# wrappery pro přístup k C++ unmanaged funkcím

TestGenerator C# knihovna obsahující testovací algoritmy a definice studentských úloh

Tester C# aplikace, která sestavuje jednotlivé moduly do finální podoby řídící jednotky

ExcerciseGenerator C# aplikace generátoru zadání studentských úloh

Test C# aplikace pro testovací a ladící účely

Všechny okna jsou tvořeny userControly (kontejner několika uživatelských prvků, který lze přepoužívat), aby bylo možné jednoduše změnit výsledné sestavení do jednoho celku. Projekt `Tester` sestavuje finální podobu řídící jednotky. Sám však neobsahuje skoro žádný výkonný kód. Slouží pouze jako kontejner (formulář), který v sobě obsahuje jednotlivé moduly z ostatních knihoven.

5.3.3 Vygenerování testovacích experimentů

Objekty imlementující jednotlivé studentské úlohy a generující pro ně testovací experimenty dle 5.2 jsou součástí projektu `TestGenerator`. Základní rodičovskou třídou je třída `RegExpMachine`, která obsahuje základní funkce a objekty pro práci s posloupnostmi logických signálů. Funkcionalitu této třídy dědí třída `Sequencer`, která obsahuje funkce potřebné k vygenerování testovacího experimentu. Některé funkce závisí na povaze studentské úlohy, proto jsou v této třídě definovány jako virtuální, aby mohly byt překryty v

podřízených třídách, ve kterých pro ně musí být uveden výkonný kód. Potomky této třídy jsou `AsynLock` a `SynLock`, které obsahují pouze funkce specifické (překrytí virtuálních) pro jednotlivé úlohy. Ukázka úseku kódu pro vygenerování testovacího experimentu je např. :

```
...
Signal[] key=new Signal[4];
//definice klíče

//vytvoření objektu as. kódového zámku s jedním klíčem
//a dvěma vstupy
AsynLock m_Lock = new AsynLock(key, 2);

//inicializace vnitřních proměnných
m_Lock.BeginTest();
//výpočet a přidání variačního testu
//do výsledné testovací posloupnosti
m_Lock.SetVariationTest();
//výpočet a přidání synchronizačního testu
//do výsledné testovací posloupnosti
m_Lock.SetSynTest();
//výpočet a přidání testu na vnoření
//do výsledné testovací posloupnosti
//s povolením náhodného faktoru 2
m_Lock.SetPrefixTest(2);
//nastavení vnitřních proměnných
m_Lock.EndTest();
...
```

Po tomto úseku kódu je objekt m_Lock naplněn daty pro provedení testovacího experimentu. Vstupní podněty jsou uloženy v listu `m_Lock.m_TestSignal` a výstupní odezva je v listu `m_Lock.m_TestResponse`.

5.3.4 Komunikace řídící jednotky s okolím

Pro komunikaci řídící jednotky s okolím slouží třída `External` v projektu `Utils`, která obsahuje funkce pro přenos testovacích podnětů a odezev do/z simulačního prostředí MATLAB. Ukázka inicializace komunikačního modulu :

```
...
//vytvoření komunikačního objektu
External m_External=new External();
//inicializace komunikačního objektu dle typu komunikace
m_External.Init(CommType.Matlab);
//start výměny dat
m_External.Start();
...
```

Pro zastavení komunikace pak slouží příkaz :

```
...
m_External.Stop();
...
```

Třída `External` má stejné rozhraní pro všechny typy komunikace. Pro doplnění o komunikaci s příslušným hardware stačí dopsat pouze podporu tohoto hardware (privátní funkce), ale z vnějšku se třída měnit nebude.

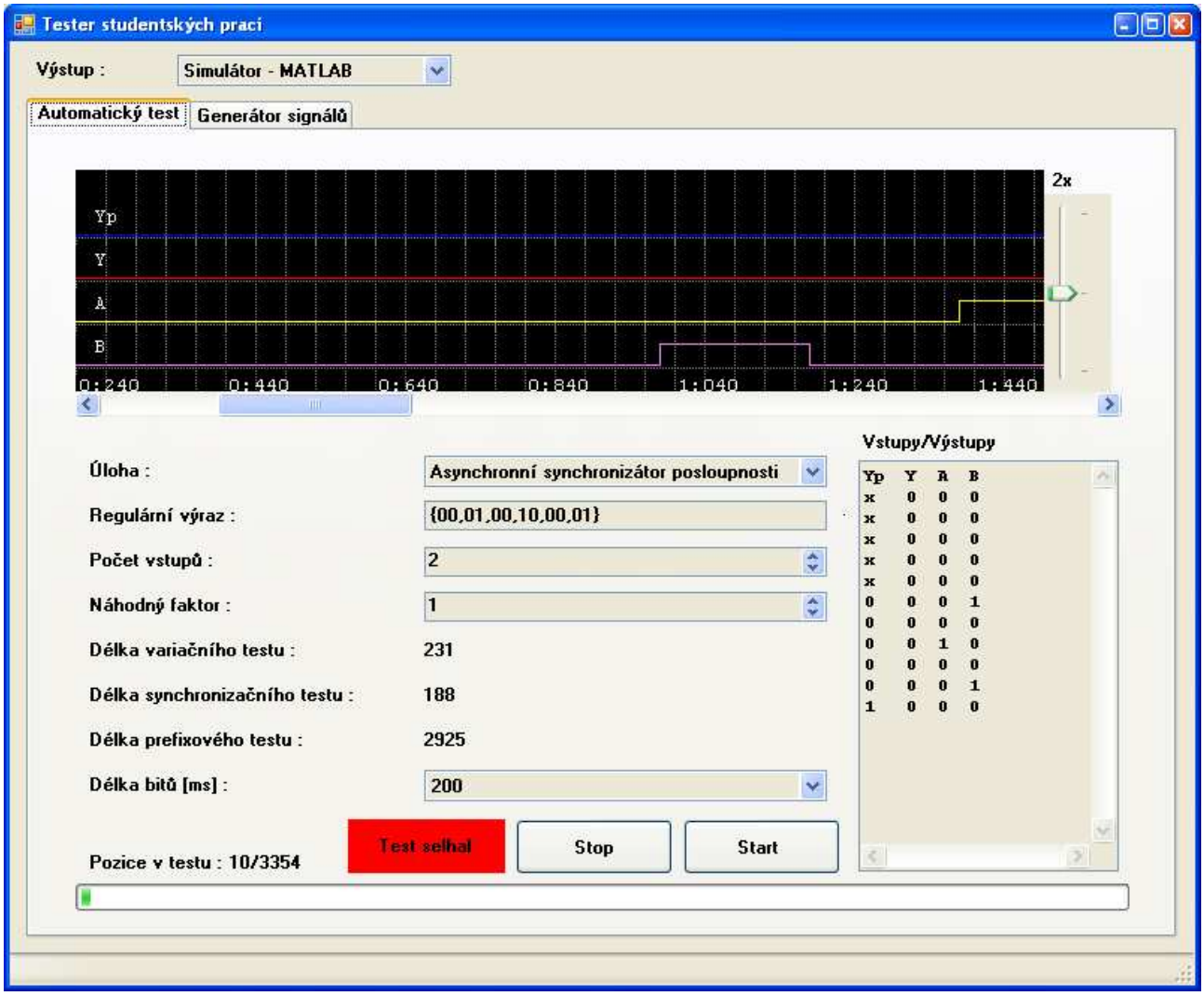
5.3.5 Modul pro provedení testovacího experimentu

UserControl `TestGenerator` slouží pro aplikaci vytvořeného testovacího souboru na testovanou studentskou úlohu a pro komunikaci s obsluhou řídící jednotky. Pro aplikaci vygenerované testovací posloupnosti a čtení odezvy na ní slouží jednoduchá smyčka :

```
...
for (int i=0;i<m_Lock.m_TestSignal.Count;i++)
{
```

```
    m_External.WriteInput(m_Lock.m_TestSignal[i].ToInt());
    Sleep(...)
    int o=m_External.ReadOutput();
    if (!m_External.AnalyzeResponse(o,i))
    {
        //test neproběhl v pořádku
        break;
    }
  }
  ...
```

Vstupem pro generátor testovacího experimentu je typ úlohy, regulární výraz, který úloha zpracovává, počet vstupů a nastavení náhodného faktoru pro test na vnoření postfixů. Tlačítkem Start se test spustí. V průběhu testu se zobrazuje aktuální pozice v testu, testovací podněty, odezva na ně a požadovaná odezva. Vstupní a výstupní průběhy jsou zobrazeny ve formě jednoduchého osciloskopu a v textové podobě. Úspěšný test je signalizován textem Test OK v zeleném rámečku. Pokud test není splněn, je zobrazen text Test selhal v červeném rámečku. V případě úspěšného dokončení i nesplnění testu je testovací smyčka ukončena. Pro komunikaci s obsluhou jsem vytvořil jednoduché GUI viz obr. 5.11.



Obrázek 5.11: GUI pro provedení testovacího experimentu.

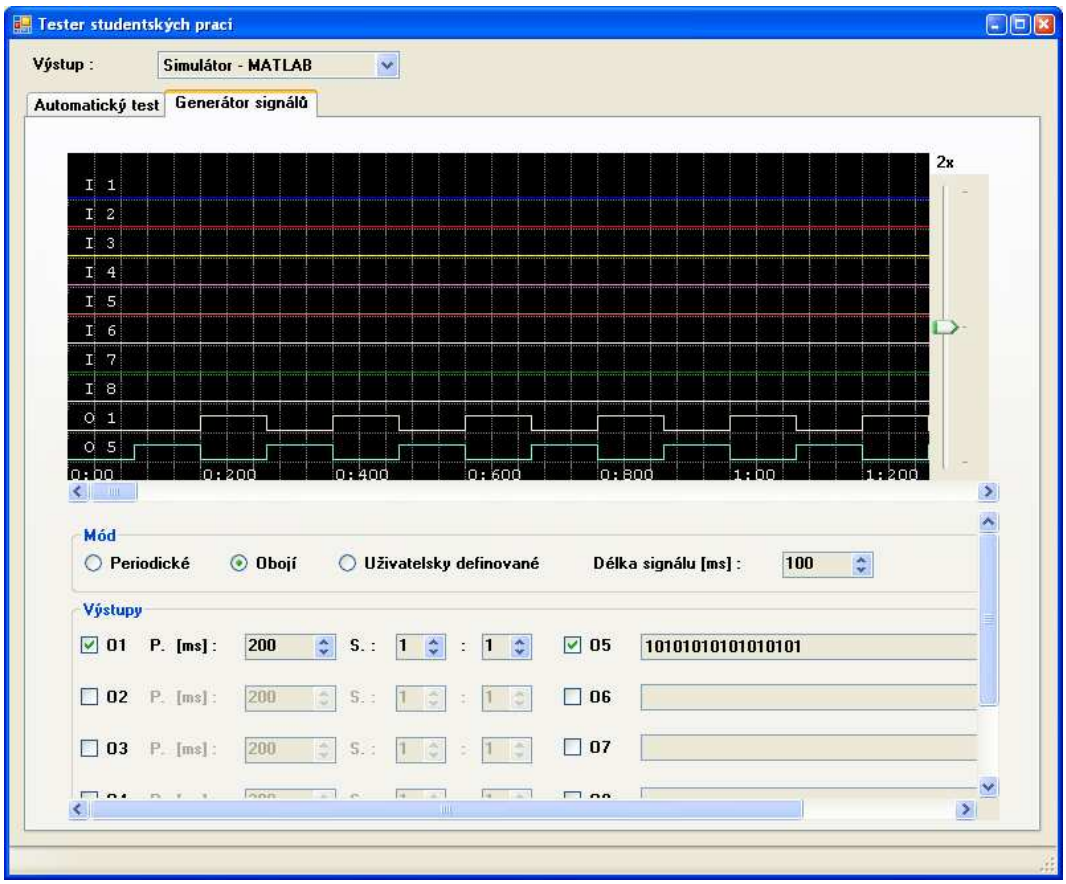
Délka bitů [ms] slouží pro nastavení komunikační rychlosti s implementací úlohy a musí být kompromisem mezi rychlostí a správnou synchronizací. Pro komunikaci se simulačním prostředím MATLAB - Simulink byla minimální délka bitu pokusy zvolena na 200 ms.

5.3.6 Modul generování a zobrazování signálů

Jako ladící pomůcku při návrhu a implementaci studentských úloh jsem vytvořil modul pro ruční generování vstupních podnětů pro studentské úlohy, který obsahuje generátor logických signálů a osciloskop pro zobrazovaní odezev.

Logický signál může být definován buď přímo logickými hodnotami (např. posloupnost 10101), nebo jako periodický signál periodou a střídou. Periodický signál ($f(t+T) = f(t)$) je takový signál, který pravidelně opakuje svůj průběh po určitém čase (periodě T). Digitální signál během jedné periody přechází mezi log. úrovněmi "1" a "0". Střída udává poměr časů, ve kterých je signál v jednotlivých úrovních. Pokud se uvádí střída ve tvaru např *X:Y*, je tím myšlen poměr trvání stavů "zapnuto":"vypnuto". Pokud je střída udána v procentech, myslí se tím obvykle doba trvání úrovně "zapnuto" vůči celkové periodě signálu.

Tester obsahuje generátor obou typů signálů. Periodický signál může být použit např. pro generování hodinových signálů a signál definovaný posloupností pro generaci testovacích podnětů. Modul obsahuje generátor 8 nezávislých logických signálů a osciloskop pro zobrazení 8 logických vstupů. Jediným nastavením osciloskopu je vzorkovací frekvence (perioda) v rozsahu 10ms - 1000ms, která udává rychlost čtení vstupů. Jelikož jsou zobrazovány i výstupy generátoru, tak je tato perioda zároveň minimální možnou šířkou generované urovně signálu (aby nedocházelo k aliasingu). Generované výstupy je možné definovat pomocí posloupnosti logických hodnot a jejich šířkou, nebo peridou a střídou. Perioda udává délku jednoho opakování v mili sekundách, střída pak poměr urovní *"0":"1"*. Pro ovládání generátoru signálů slouží GUI viz obr. 5.12.



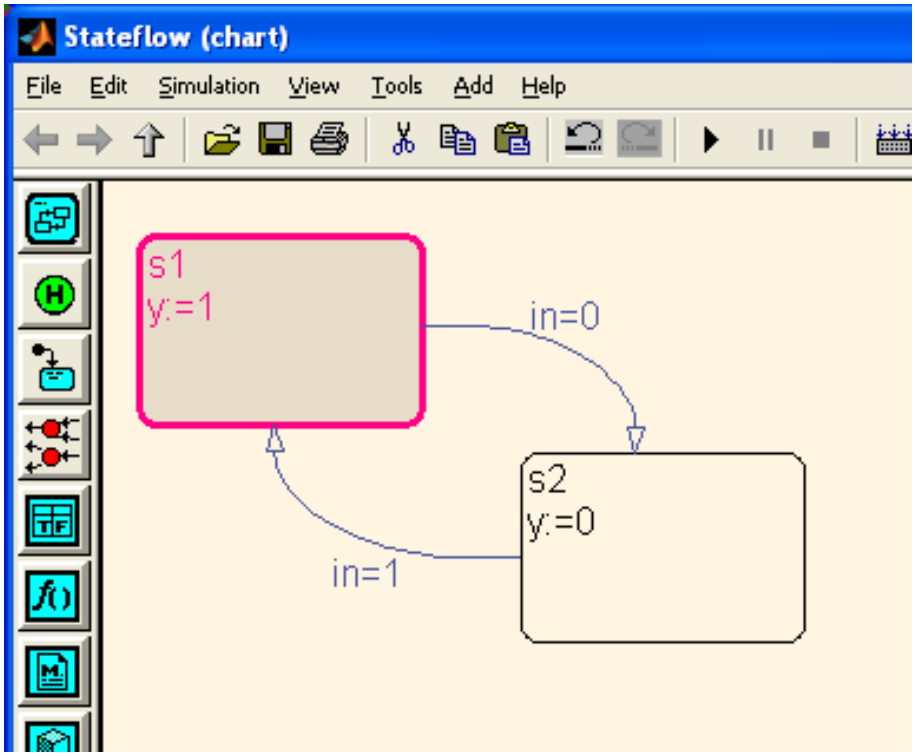
Obrázek 5.12: GUI modulu pro generování a zobrazování logických signálů.

5.4 Propojení s externím simulačním SW

Před každou hardwarovou realizací úlohy je vhodné nejdříve provést její simulaci, odhalit chyby v řešení, opravit je a končnou realizaci provést až když "funguje" model. Pro simulaci konečných automatů slouží mnoho software. Jedním z nich je MATLAB - Simulink, který obsahuje toolbox Stateflow pro tvorbu přechodových diagramů. V řídící jednotce testeru jsem proto vytvořil komunikační rozhraní s modelem v MATLAB Simulinku přes sdílenou dynamickou knihovnu. Od testování hardware se testovací experiment až na způsob komunikace nijak neliší.

Program Simulink je zaměřen na tvorbu modelů dynamických systémů. Obsahuje velké množství knihoven s moduly (bloky), ze kterých se tyto modely staví. Jedním z

bloků je Stateflow, který realizuje dynamický systém popsaný přechodovým diagramem (pomocí stavů a přechodů mezi nimi). Tento blok je možné propojit s dalšími moduly simulinku (např. volání zdrojových kódů matlabu) a zpřístupnit tak model externí testovací aplikaci.



Obrázek 5.13: Způsob tvorby přechodových diagramů pomocí toolboxu Stateflow.

MATLAB od verze 6.5.1 obsahuje podporu pro volání dynamicky linkovaných knihoven (*.dll ve Windows a *.so v Linuxu). Pomocí funkce *loadlibrary* je možné zavést knihovnu do MATLABu a funkcí *calllib* volat funkce z této knihovny (MATLAB se sám postará o konverzi mezi typy). Podrobnosti uvedeny v [6]. Zdrojový kód MATLABu sloužící pro dynamické zavedení knihovny a přečtení vstupů generovaných z testeru je

```
%zavedeni knihovny
loadlibrary External.dll External.h alias lib
B=calllib('lib','WReadInput');
%zruseni odkazu na knihovnu
unloadlibrary lib
```

Funkce *loadlibrary* jako parametry přebírá cestu na knihovnu, její hlavičkový soubor (*.h) a jméno aliasu pro tuto knihovnu. Funkce *calllib* vyžaduje název aliasu, název volané funkce a vstupní parametry této funkce. Návratovým parametrem je pak návratová hodnota volané knihovní funkce. Po ukončení práce s knihovnou se na ni smaže vazba funkcí *unloadlibrary*. Simulinkový model v každém simulačním kroku volá knihovní funkce *WReadInput*, *WReadOutput* pro čtení/zápis aktuálních vstupů/výstupů generovaných/čtených testerem.

Každá aplikace, která dynamicky linkuje knihovnu si ve svém paměťovém prostoru vytváří její obraz (od Win32). Data nesdílí, a proto je nutné explicitně vytvořit blok sdílené paměti. Překladač Microsoft Visual C++ obsahuje direktivu překladače *#pragma(dataseg)*, kterou je možné definovat začátek bloku sdílené paměti. Pro komunikaci testeru s MATLTABem jsou použity 3 sdílené proměnné (vstup, výstup a příznak aktivity komunikace).

```
#pragma data_seg(".shared")
  volatile int Input=0;
  volatile int Output=0;
  volatile bool start=false;
#pragma data_seg()
```

V *.def souboru projektu je poté nutno uvést význam názvu *.shared*.

```
SECTIONS
  .shared   READ WRITE SHARED
```

Pro vzájemnou komunikaci pak slouží funkce pro zahájení/zrušení komunikace, zápis a čtení vstupů/výstupů. Hlavičkový souboru knihovny External.dll je

```
// External.h
#define DLLWIN32_EXPORTS
#ifdef  DLLWIN32_EXPORTS
    #define DLLWIN32_API extern "C" __declspec(dllexport)
#else
```

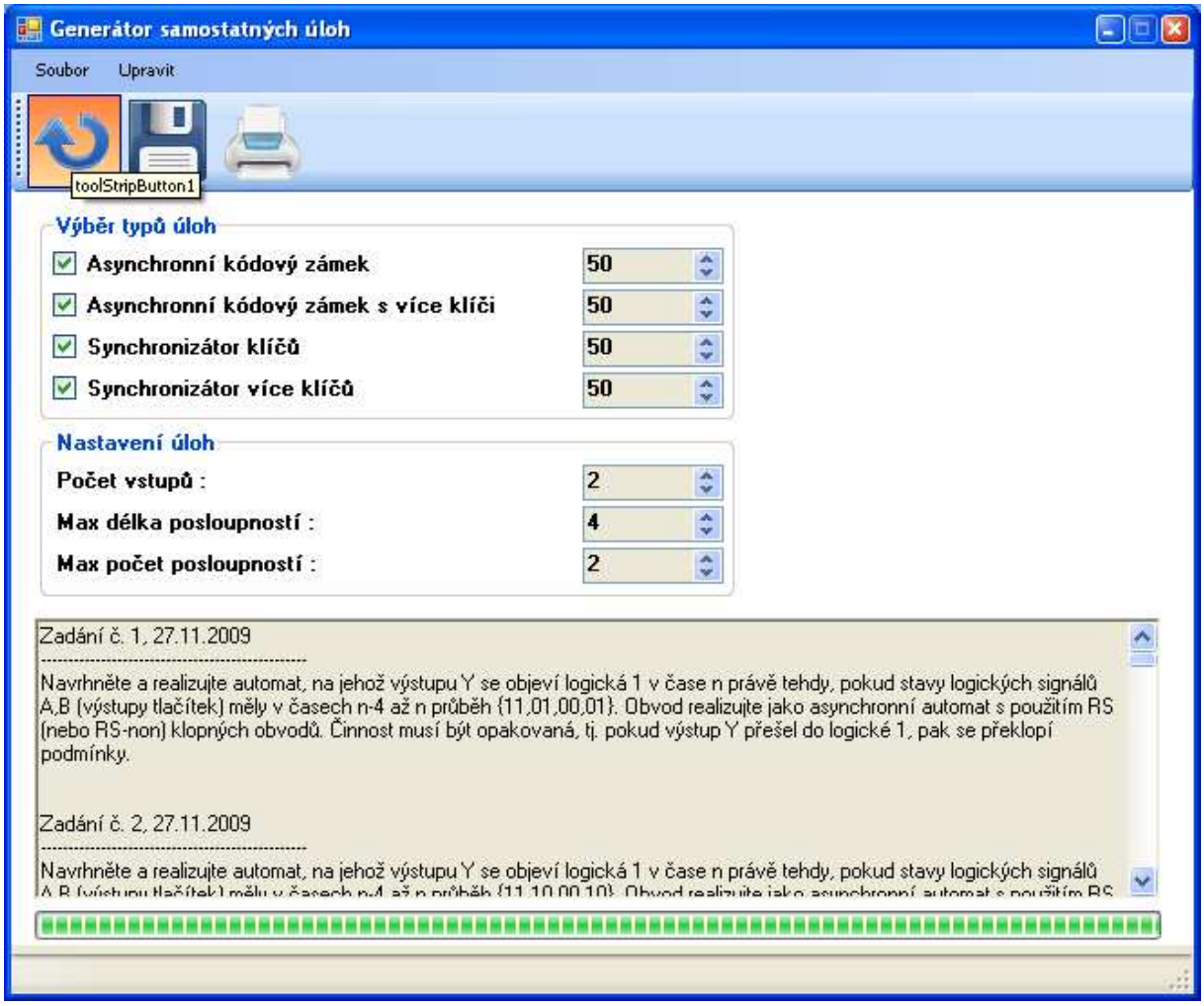
```
    #define DLLWIN32_API __declspec(dllimport)
#endif

DLLWIN32_API "C" __declspec(dllexport) void __stdcall WInitCommunication();
DLLWIN32_API "C" __declspec(dllexport) void __stdcall WStartCommunication();
DLLWIN32_API "C" __declspec(dllexport) void __stdcall WStopCommunication();
DLLWIN32_API "C" __declspec(dllexport) void __stdcall WWriteInput(int I);
DLLWIN32_API "C" __declspec(dllexport) int  __stdcall WReadInput();
DLLWIN32_API "C" __declspec(dllexport) int  __stdcall WReadOutput();
DLLWIN32_API "C" __declspec(dllexport) void __stdcall WWriteOutput(int O);
DLLWIN32_API "C" __declspec(dllexport) void __stdcall WPause(int Ms);
```

Pro správnou synchronizaci simulinkového modelu s testovací aplikací je nezbytné, aby Matlabovská simulace běžela v diskrétních krocích v hodnotách desítek ms, čehož se dosáhne voláním funkce `WPause(Ms)`. Tato funkce zajistí "uspání" simulace na dobu zadanou v parametru Ms v každém kroku simulace. Pokusy byla hodnota kroku nastavena na 20ms, což odpovídá vzorkovací periodě testovací aplikace.

5.5 Generátor studentských úloh

Pro ulehčení a zrychlení procesu vymýšlení-vytváření zadání studentských úloh jsem vytvořil jejich generátor. Generátorem jsou vytvářeny zadání všech úloh popsaných v kapitole 5.1. Pro ovládání generátoru slouží jednoduché GUI viz obr. 5.14.



Obrázek 5.14: GUI generátoru zadání studentských úloh.

Generátor zadání studentských úloh umožňuje ovlivnit množinu zadání pomocí nastavení :

Typ úloh ovlivňuje které úlohy budou obsaženy ve výstupní množině zadání.

Počet vstupů (znaků) generovaných zadání úloh.

Délka postfixů generovaných zadání úloh.

Max. počet postfixů maximální počet postfixů úloh s více klíči viz 5.1.

Výstupem generátoru je textový soubor ve formátu `*.rtf`, který je možné uložit na

disk, nebo vytisknout na tiskárně. Textové formulace zadání úloh nejsou konstantní, ale je možné je upravit pomocí jejich editoru volbou položky menu Upravit/Upravit text zadání.

Kapitola 6

Závěr

Praktické studentské úlohy jsou nezbytnou součástí výuky většiny předmětů na vysoké škole, což platí i pro předměty s výukou počítačových systémů, konkrétně logického řízení. Studentské úlohy slouží pro ověření teoretických znalostí a praktických dovedností studentů. Nestačí však aby studenti úlohy pouze vyřešili, musí je i odevzdat a nechat projít kontrolou. Kontrola je prováděna většinou vyučujícím konkrétního předmětu na cvičení, což vede k jeho zbytečné časové alokaci. Podstatou této práce bylo proces kontroly zautomatizovat, tzn. navrhnout testovací experimenty a vytořit soustavu, pomocí níž kontrolu provede sám student během cvičení.

Nahrazení člověka v procesu kontroly automatickými prostředky znamenalo sestrojit kontrolní experiment 2.3 pro zvolenou třídu studentských úloh, což jsou v tomto případě hardwarové implementace konečných automatů viz 2.2, konkrétně asynchronní detektory posloupností popsané v kapitole 5.1. Výsledkem kontrolního experimentu je rozhodnutí o správnosti implementace zadané studentské úlohy. Toto rozhodnutí je možné učinit, pouze pokud byly úlohy podrobeny testovacím experimentům viz 4.1. Testovací experiment je proces stimulace systému vstupními podněty a současného pozorování odezvy systému na tyto podněty. Z tohoto V/V chování je možné vypozorovat správnost implementace a rozhodnout o ni.

Testovací metody popsané v kapitole 4.2.1 jsem nepoužil, protože úlohy asynchronních detektorů nevyhovují podmínkám, které jsou kladeny na testované systémy. Proto jsem pro tyto úlohy navrhl vlastní testovací experimenty, které jsou popsány v kapitole 5.2. Asynchronní detektory posloupností jsou stavové systémy, jejihž konečné automaty přechází do koncového stavu po přijmutí některého ze slov z jazyka, který přijímají. Podstatou experimentů které jsem navrhl je ověřit, jestli studentské implementace přijímají pouze tento jazyk. Ověření funkčnosti navržených experimentů jsem provedl

simulacemi na matlabovských modelech úloh předpokládaných složitostí, do kterých jsem záměrně zanesl různé chyby. Závěry těchto simulací uvádím v kapitole 5.2.5. Předpokládaná časová náročnost testů při testování reálných hardware je uvedena v 5.4. Tato tabulka je také důkazem, že testovací experiment pro předpokládané složitosti studentských úloh (posloupnosti délky max. $k = 6$ se dvěmi vstupy $m = 2$) je možné rychle provést během cvičení. Pro složitejší úlohy se délka testu výrazně redukuje zavedením náhodného faktoru do prefixového testu. Ze závěrů uvedených v kapitole 5.2.5 lze vidět, že většina chyb je odhalena variačními a synchronizačními testy, takže redukce množiny testovacích prefixů výrazně nevadí.

Pro provedení kontrolního experimentu jsem vytvořil pouze řídící jednotku testovací soustavy, protože v době kdy jsem na diplomové práci pracoval ještě nebyl znám typ hardware, na kterém budou studenti své úlohy realizovat. Úkolem řídící jednotky je dle zadání úlohy kontrolní experiment vytvořit a provést. Vstupem pro řídící jednotku je typ úlohy a rozeznávané postfixy, výstupem je pak rozhodnutí o správnosti implementace úlohy. Podrobný popis řídící jednotky je uveden v kapitole 5.3.1. Pro podporu fáze návrhu a ladění úloh jsem v řídící jednotce vytvořil ruční generátor logických signálů viz 5.3.6. Pro tyto účely obsahuje řídící jednotka komunikační mód MATLAB - Simulink. Pomocí paměťově sdílené dynamické knihovny může řídící jednotka komunikovat s modely studentských úloh vytvořených v toolboxu Stateflow simulačního prostředí MATLAB - Simulink. Student si tak může otestovat své řešení ještě před jeho hardwarovou implementací. Popis komunikace uvádím v kapitole 5.4. Návod na vytvoření simulinkového modelu a nastavení komunikace s řídící jednotkou je uveden v příloze B.

Pro zrychlení vytváření zadání studentských úloh jsem vytvořil jejich generátor. V kapitole 5.14 popisuji jakým způsobem pomocí generátoru vytvořit množinu zadání pro studenty.

V této diplomové práci jsem vytvořil prostředky pro zautomatizování celého procesu práce se studentskými úlohami dle definice 2.2. Učitelé nebudou muset vymýšlet zadání úloh a složitě kontrolovat hardwarové implementace těchto úloh. Studenti ve fázi návrhu mohou použít simulační prostředí a generátor signáů pro ladící účely. Veškeré zdrojové kódy a sestavené assembly jsou přiloženy na CD.

Pro plnou funkčnosí testovací soustavy je však nutné vytvořit komunikační rozhraní s konkrétními typy hardware. Další možností rozšíření řídící jednotky je vytvoření testovacích experimentů pro jiné typy úloh než detektory posloupností. Celá řídící jednotka je navřena tak, aby rozšíření o další moduly úloh nepřineslo velké úsilí a komplikace. Modul generování a provádění experimentů je striktně oddělen od komunikačního mod-

ulu, takže nebude problém komunikaci s příslušným hardware doplnit. Programátorská dokumentace zdrojového kódu je taktéž přiložena na CD.

Literatura

[1] Friedman A.D., Menon P. R. : *Fault Detection in Digital Circuits.*
Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1971

[2] Lee D., Yannakakis M. : *Principles and Methods of Testing Finite State Machines-A survey.*
Proceedings of the IEEE, 1996

[3] Abramovici M., Breuer M.A., Friedman A.D. : *Digital Systems Testing and Testable Design.*
IEEE PRESS, 1990

[4] Lee D., Yannakakis M. : *Testing Finite State Machines : state identification and verification.*
IEEE Trans. Computers, vol. 43. no. 3, pp. 360-320, 1994

[5] Kohavi Z. *Switching and Finite Automa Theory, 2nd. ed..*
McGraw-Hill, New York, 1978

[6] *Calling Functions in Shared Libraries.*
`http://www.mathworks.com`

[7] Černá I., Křetínský M., Kučera A. *Automaty a formální jazyky I.*
FI MU, 2002

[8] Barták R. *Automaty a gramatiky, přednášky.*
KTIML `http://ktiml.mff.cuni.cz/~bartak`

[9] Bayer J. *Logické řízení.*
DCE, FEL ČVUT `http://dce.felk.cvut.cz/lor/prednasky/`

Příloha A

Obsah přiloženého CD

Obsah přiloženého CD :

- dp_2009_Lukas Dibelka.pdf : tato práce ve formátu pdf
- stateflow_test_manual.pdf : návod na vytvoření simulinkového modelu asynchronního detektoru posloupností ve formátu pdf
- source : všechny zdrojové kódy řídící jednotky testovací soustavy
- bin : všechny přeložené assembly řídící jednotky testovací soustavy
- template : prázdný simulinkový model asynchronního detektoru posloupností
- html_doc : kompletní HTML dokumentace všech zdrojových kódů

Příloha B

Návod na vytvoření matlabovského modelu asynchronního detektoru posloupností

1 Úvod

Tento dokument slouží jako návod pro vytvoření modelu konečného automatu v toolboxu Stateflow programu Matlab - Simulink. Dále je zde uveden popis propojení modelu asynchronního detektoru posloupností s testovací aplikací `Tester`.

2 Minimální požadavky

Matlab - Simulink obsahující toolbox Stateflow pro vytvoření modelu konečného automatu.

Matlab 6.5.1 min. verze pro komunikaci simulinkového modelu s testovací aplikací.

Tester.exe testovací aplikace pro testování asynchronních detektorů posloupností.

External.dll,External.h sdílená dynamická knihovna pro výměnu dat mezi Matlabem a testovací aplikací.

3 Vytvoření modelu konečného automatu

Prvním krokem pro vytvoření modelu konečného automatu je založení nového Simulinkového modelu **.mdl*, např. příkazem `simulink` v příkazové řádce Matlabu a poté volbou menu File/New/Model. Do takto vytvořeného prázdného modelu se pak vloží blok Chart, a to volbou menu View/Library Browser/Stateflow/Chart a přetažením ikony Chart do okna modelu. Dvojklikem na blok Chart se otevře editační okno toolboxu Stateflow sloužící pro vytvoření přechodového diagramu konečného automatu. Detailní popis jak vytvořit přechodový diagram je např. na `http://www.mathworks.com/access/helpdesk/help/pdf_doc/stateflow/sf_ug.pdf`. Ukázka viz obr. 1.

Pro komunikaci přechodového diagramu v Stateflow se Simulinkem je nutné v Stateflow modelu definovat vstupy a výstupy, což se provede v menu Add/Data/Inputs from Simulink a Add/Data/Outputs to Simulink, ukázka nastavení je na obr. 2. Na takto pojmenované vstupy/výstupu (typ `double`) je poté možno odkazovat v přechodových funkcích přechodového diagramu. Stateflow model potřebuje pro svou funkci ještě hodinový signál, který určuje okamžiky přechodů. Po vytvoření přechodového diagramu je nutné propojit tento blok se zbytkem Simulinkového modelu.

4 Propojení vstupů a výstupů

Pro propojení modelu s testovací aplikací `Tester` je třeba zajistit správnou konverzi vstupních a výstupních dat. Sdílenou pamětí mezi testovací aplikací a modelem jsou dva `integery`, (jeden vstupní a jeden výstupní), tzn. každý vstup/výstup odpovídá jednomu bitu v masce vstupní/výstupní hodnoty (max. 32). Proto je třeba pro přenos správných vstupních a výstupních hodnot zajistit správné maskování (logický součin s maskou). Příklad propojení vstupních a výstupních bloků s maskováním je na obr. 3.

5 Nastavení komunikace s testovací aplikací

Komunikace testovací aplikace a Matlabu je ralizována přes sdílený blok paměti v dynamické knihovně `External.dll`, tzn. oba programy musí linkovat stejnou instanci knihovny. V Matlabu je proto nutné před spuštěním simulace tuto knihovnu (a její header **.h*) načíst do paměti příkazem :

```
loadlibrary C:\Tester\External.dll C:\Tester\External.h alias lib
```

Tímto příkazem Matlab do svého paměťovém prostoru načte funkce a sdílený blok paměti této knihovny.

6 Čtení vstupů a zápis výstupů

Pro čtení sdílených vstupů a zápis na výstupy jsou v knihovně `External.dll` funkce :

```
DLLWIN32_API "C" __declspec(dllexport) void __stdcall WWriteInput(int I);
DLLWIN32_API "C" __declspec(dllexport) int  __stdcall WReadInput();
DLLWIN32_API "C" __declspec(dllexport) int  __stdcall WReadOutput();
DLLWIN32_API "C" __declspec(dllexport) void __stdcall WWriteOutput(int O);
DLLWIN32_API "C" __declspec(dllexport) void __stdcall WPause(int Ms);
```

Funkce `WPause()` slouží pro pozastavení simulace na čas v ms, kvůli zajištění synchronizace s testovací aplikací. Z Matlabu se knihovní funkce volají přes matlab funkci `calllib`. Funkce pro čtení vstupů má tedy tvar :

```
function [I]=loadInputs(x)
  I=calllib('lib','WReadInput');
end
```

A pro zápis výstupů :

```
function saveOuts(Y)
  calllib('lib','WWriteOutput',Y);
  calllib('lib','WPause',20);
end
```

Pro volání těchto funkcí ze simulinkového modelu slouží v modelu bloky MATLAB Function viz (*reading inputs/writing outputs*) na obr. 3. Dvojklikem na tyto bloky se otevře okno s jeho vlastnostma, ve kterých se v políčku MATLAB Function nastaví příslušná funkce (čtení pro vstupy a zápis pro výstupy). Tyto funkce je nutné mít v samostatných souborech pojmenovaných stejně jako funkce v adresáři, na který ma Matlab nastavenou referenci (nejlépe *Works*). Nastavení reference na jiný adresář se v Matlabu provede volbou menu File/SetPath/Add folder.

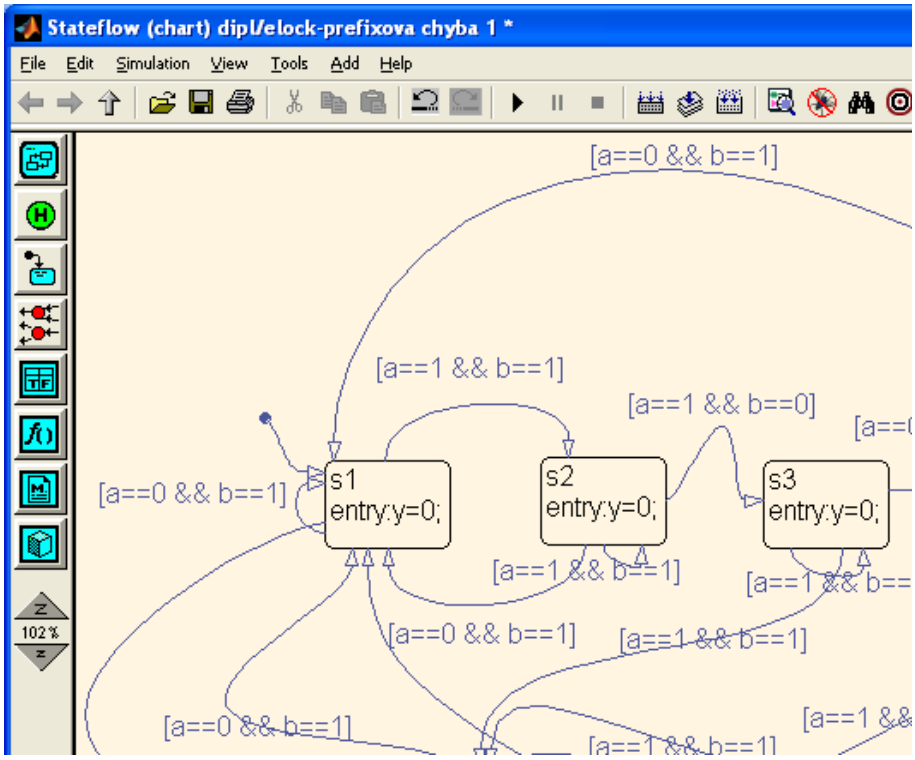
7 Spuštění simulace

Před spuštěním simulace je vždy potřeba načíst knihovnu `External.dll`. Simulaci je tedy výhodné spustit skriptem, který provede načtení knihovny i start simulace :

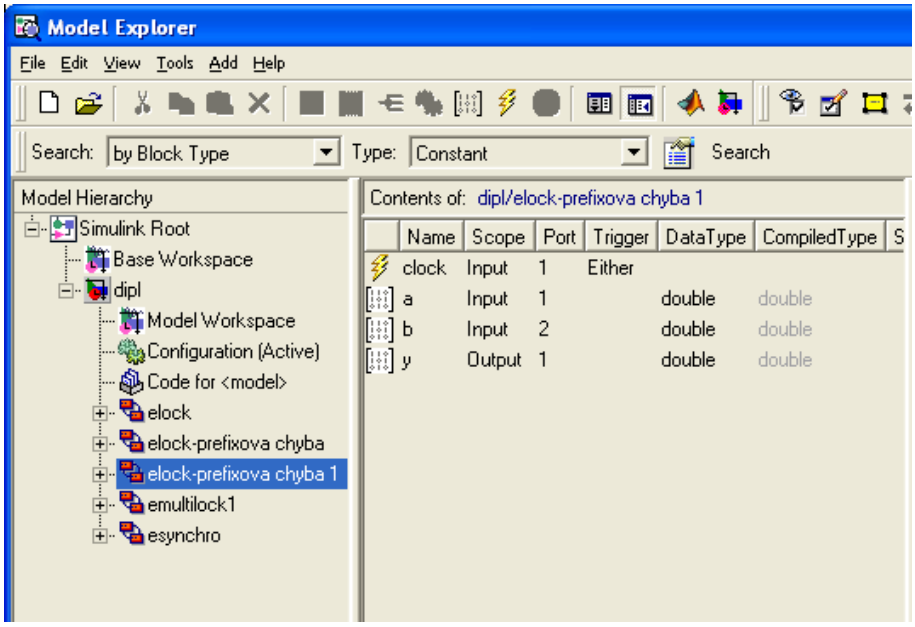
```
loadlibrary C:\Tester\External.dll C:\Tester\External.h alias lib
set_param('nazev_modelu','SimulationCommand','Start')
```

8 Ovládání testovací aplikace

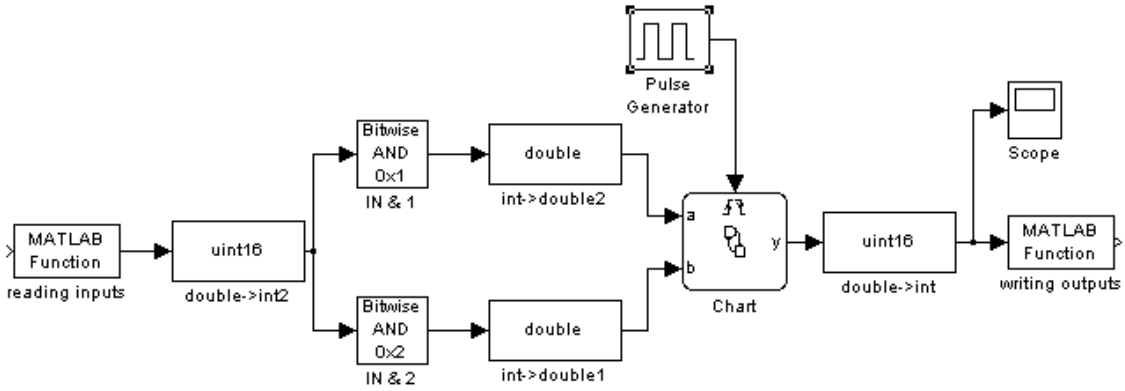
Testovací aplikace slouží pro otestování navrženého a implementovaného konečného automatu asynchronního detektoru posloupností. Vstupem pro aplikaci jsou typ detektoru (zámek nebo synchronizátor), přijímaný postfix/y, počet vstupů a náhodný faktor pro testovací algoritmus. Dále je třeba nastavit způsob komunikace, což je v tomto případě Simuátor - Matlab a délků bitů generovaných signálů v ms. Délka bitů ovlivňuje komunikační rychlost, v případě špatné synchronizace se simulinkovým modelem je třeba ji pokusně zvýšit. Uživatelské rozhraní aplikace je na obr. 4.



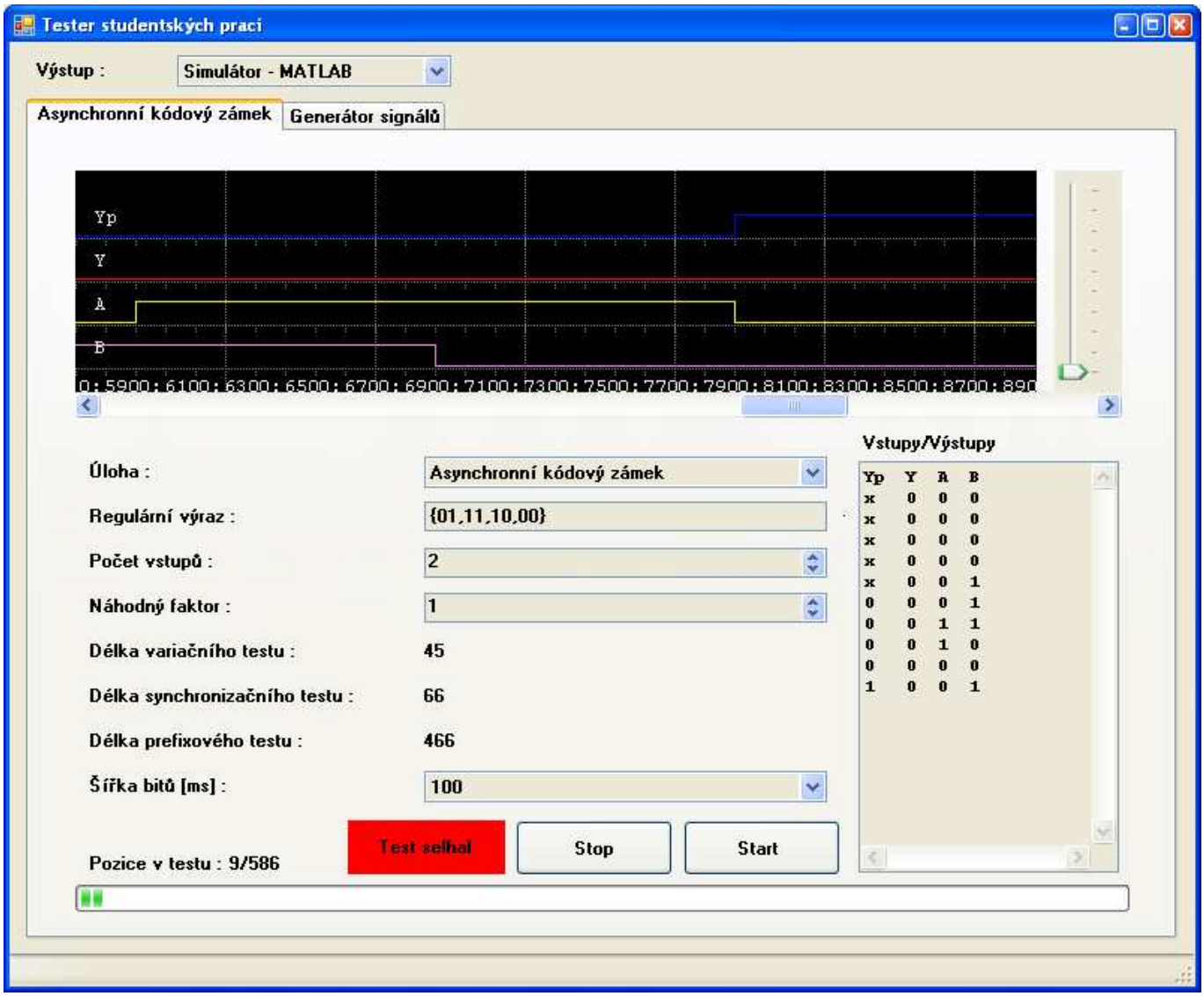
Obrázek 1: Ukázka přechodového diagramu modelu konečného automatu v toolboxu Stateflow.



Obrázek 2: Nastavení V/V proměnných a hodinového signálu.



Obrázek 3: Příklad propojení bloků simulinkového modelu asynchronního konečného automatu.



Obrázek 4: GUI testovací aplikace asynchronních detektorů posloupností.