České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra řídící techniky

Aplikace pro převod časovaných automatů do FPGA

Jan Breuer

Vedoucí práce: Ing. Jan Krákora

Bakalářská práce
srpen 2007

Abstrakt

Cílem této bakalářské práce je vytvořit převodník jedné reprezentace časovaného automatu na jinou. Předloha časovaného automatu je vytvořena a otestována v nástroji UPPAAL. Výstup tohoto programu je načítán a konvertován do jiného jazyka, který je vhodný pro další práci a následné naprogramování do FPGA. Za výsledný jazyk byl zvolen jazyk MATLABu. V MATLABu se pomocí Xilinx System Generatoru vytváří kód pro výslednou platformu FPGA.

Abstract

The purpose of this Bachelor's thesis is to create a converter from one representation of a timed automata to another. The model of the timed automata is created and tested in a UPPAAL tool. The output of the tool is loaded and converted to another language, which is suitable for other work and a subsequent programming to FPGA. As a final language was selected a language of MATLAB. In MATLAB, with the aid of Xilinx System Generator, is created a code for the final platform FPGA.

Poděkování

Chtěl bych poděkovat především Ing. Janu Krákorovi, vedoucímu mé bakalářské práce, za podnětné rady, připomínky a poskytnutí vzorových příkladů a dalších potřebných materiálů. Dále bych chtěl poděkovat své rodině za podporu a finální korektury.

České vysoké učení technické v Praze, fakulta elektrotechnická

Katedra řídicí techniky | Školní rok: 2006/2007

Zadání bakalářské práce

Student: Jan Breuer

Obor: Kybernetika a měření

Název tématu: Aplikace pro převod časovaných automatů do FPGA

Zásady pro vypracování:

1. Seznamte se s problematikou programování FPGA.
2. Seznamte se s pojmem časovaných automatů (TA).
3. Navrhněte možné řešení aplikace převádějící struktury z TA do jazyků FPGA (VHDL, HandelC, Xilinx System Generator).
4. Naprogramujte aplikaci převodníku.
5. Proveďte funkční testy a vše řádně zdokumentujte.

Seznam odborné literatury:

- Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on uppaal, formal methods for the design of real-time systems. In SFM-RT 2004, pages 200–236. Springer-Verlag, 2004. http://www.uppaal.com
- Gerd Behrmann. Uppaal Timed Automata Parser - UTAP, 2005
- Sudhakar Yalamanchili. VHDL Starter's Guide. Prentice Hall, 1998. 269 pages
- http://micro.feld.cvut.cz/home/hazdra/prs/
- Xilinx system generator v7.1 user guide, 2005. Xilinx ltd. http://www.xilinx.com

Vedoucí bakalářské práce: Ing. Jan Krákora

Datum zadání bakalářské práce: zimní semestr 2006/07

Termín odevzdání bakalářské práce: 15. 8. 2007

Prof. Ing. Michael Šebek, DrSc.
vedoucí katedry

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE Fakulta elektrotechnická -4- L.S.

Prof. Ing. Zbyněk Škvor, CSc.
děkan

V Praze, dne 28. 2. 2007

Prohlášení

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

V Praze dne ....

podpis

Obsah

1 Úvod

Tato bakalářská práce se zabývá převodem časovaných automatů vytvářených pomocí nástroje UPPAAL [13] do podoby použitelné pro naprogramování FPGA. UPPAAL je nástroj na vytváření, simulaci a testování časovaných automatů. Nástroj obsahuje několik dílčích částí. V grafickém uživatelském rozhraní lze pohodlně navrhovat čaasované automaty a dále je pomocí podpůrných programů simulovat a verifikovat. Navržený automat lze uložit do souboru, který je klíčový pro další použití a převod. UPPAAL ukládá časované automaty do XML souboru ve kterém jsou všechny logické výrazy a inicializace systému psány v jeho vlastním jazyku.

Hlavní použití tohoto nástroje je vytvoření automatu pro testování řídících systémů [12]. Konkurenční nástroje využívají operační systém reálného času [11], například RTLinux. Tento převodník má za cíl platformu FPGA. Hlavní výhodou konstrukce s FPGA je mnohonásobně kratší doba odezvy. Například vzorkovací perioda je u zařízení s FPGA zhruba 10 μs oproti 100 ns u operačního systému. Mnohem důležitější je navíc fakt, že nestabilita vzorkovací frekvence (jitter) je u FPGA daleko menší, protože se jedná o synchronní logický obvod.

V první části je popsán teoretický úvod do časovaných automatů [6, 2, 1], jejich definice a základní rozdělení. Dále je popsán nástroj UPPAAL a je vysvětleno jeho možné použití v oblasti návrhu, simulace a verifikace časovaných automatů [3, 15, 5]. Následující sekce popisuje realizaci časovaného automatu v Xilinx System Generator Toolboxu [9]. Dále je popsán použitý nástroj na generování parseru gramatiky UPPAALu – ANTLR [14] a ANTLRWorks [7] a je popsán převod. Tím je dvoufázová operace, ve které se xml soubor převede na textovou reprezentaci, která je předána parseru. Parser je vytvořen pomocí parser generátoru ANTLR a jeho výstupem je stromová struktura v paměti. Ta se dále prochází a generují se příslušné stavy, podmínky přechodu a hlavičky funkcí. Při převodu byl brán důraz na optimalizaci počtu vstupně/výstupních proměnných.

2 Časovaný automat – teoretický úvod

Nejprve je třeba definovat základní pojmy pro další práci. Uvádím zde definici přechodového systému, který slouží k obecnému popisu. Následný časovaný automat je už jen konkrétní případ přechodového systému. Přechodový systém představuje ještě větší abstrakci než *konečný stavový automat* a na rozdíl od něj nemusí mít konečný počet stavů a přechodů. Zde uvedené definice nejsou úplně matematicky korektní a slouží hlavně pro přiblížení vlastností časovaných automatů.

2.1 Přechodový systém

Přechodový systém je abstraktní stroj pro modelování systémů a jejich chování [1]. Je složen ze stavů a z přechodů mezi nimi. Přechodový systém S lze popsat uspořádanou n-ticí (Q, Q_0, Σ, T), kde

- Q je množina stavů
- $Q_0 \subseteq Q$ je množina počátečních stavů
- Σ je množina označení nebo událostí
- $T \subseteq Q \times \Sigma \times Q$ je množina přechodů

Pro přechod (q, α, q') z T lze psát $q \xrightarrow{\alpha} q'$. Systém začne ve svém počátečním stavu a pokud $q \xrightarrow{\alpha} q'$, může změnit svůj stav z q do q' při události α. Událost α může znamenat například vstup do systému, podmínku, která musí platit, nebo akci, která se má vykonat. Stav q je dosažitelným stavem přechodového systému, když je dosažitelný z některého počátečního stavu. Z jednoho stavu do druhého je možné přejít jedním z pojmenovaných přechodů, kterých může být obecně více.

2.2 Komplexní přechodový systém

Komplexní systém může být popsán jako výsledek spojení přechodových systémů. Například $S_1 = (Q_1, Q_1^0, \Sigma_1, T_1)$ a $S_2 = (Q_2, Q_2^0, \Sigma_2, T_2)$ jsou dva přechodové systémy. Výsledný systém, který vznikne jejich spojením, je $(Q_1 \times Q_2, Q_1^0 \times Q_2^0, \Sigma_1 \cup \Sigma_2, T)$. Symboly, které patří oběma automatům, slouží k jejich synchronizaci. Přechod $(q_1, q_2) \xrightarrow{\alpha} (q_1', q_2')$ proběhne právě tehdy, když α náleží do označení obou automatů $\alpha \in \Sigma_1 \cap \Sigma_2$ a $q_1 \xrightarrow{\alpha} q_1'$ a $q_2 \xrightarrow{\alpha} q_2'$. V tomto příkladu je synchronizace definována jako blokující.

2.3 Časovaný automat

Časovaný automat je uspořádaná n-tice (Q, q_0, I, G, C, T), kde navíc

- q_0 je jediný počáteční stav (vychází z požadavků UPPAALu)
- I je množina invariantů
- G je množina guard (podmínek přechodu)
- C je množina hodin
- F_G: $T \to G$ je funkce, která na vstupní přechod vrátí všechny jeho podmínky přechodu
- F_I: $Q \to I$ je funkce, která na vstupní stav vrátí všechny jeho invarianty

- F_T: $Q \rightarrow T$ je funkce, která na vstupní stav vrátí všechny přechody, které z něj vystupují

V kontextu časovaných automatů je *invariant* podmínka, která musí být splněna, aby bylo možné v daném stavu setrvat, nebo do něho vstoupit. Podmínka přechodu (*guard*) slouží k otevření daného přechodu, pokud je splněna.

2.4 Nedeterministický časovaný automat (NTA)

Je uspořádaná n-tice (A, C, V), kde

- $A = \{a_1, a_2, a_3, ..., a_n\}$ je množina časovaných automatů
- $C = C_1 \cup C_2 \cup ... \cup C_n$ je množina hodin v systému
- V je množina proměnných v systému
- Q_C je množina aktuálních stavů všech automatů

2.4.1 Možnosti přechodu do jiného stavu

Přechod do nového stavu $(Q_C, C, V) \rightarrow (Q'_C, C', V')$ může mít několik možností.

- Setrvání v aktuálním stavu $q \xrightarrow{d} q$ a posun všech hodin o d, což je celé kladné číslo $C' = C + d$, pokud je pro každý stav $q \in Q_C$ splněn invariant $F_I(q)$ i pro čas C'
- Přeskočení do nového stavu $q \xrightarrow{\alpha} q'$ muže nastat v jednom z případů
    1. Některý přechod $t = F_T(q)$ splňuje podmínky přechodu a pokud jsou pro stav $q' \in Q'_C$ splněny invarianty $F_I(q')$.
    2. Některé dva přechody $t_i, t_j \in T$ se synchronizačním kanálem c ze stavů $q_j, q_i \in Q_C$ ve dvou automatech a_i, a_j splňují podmínky $F_I(q'_i), F_I(q'_j)$ a zároveň podmínky přechodu $F_G(q_i), F_G(q_j)$, pak proběhne přechod do dalšího stavu $q_i \xrightarrow{c!,\alpha_i} q'_i$ a $q_j \xrightarrow{c?,\alpha_j} q'_j$ s vykonáním α_i, α_j.
    3. Několik přechodů $t_0, ..., t_n \in T$ z několika automatů $a_0, ..., a_n$, pro které díky kanálu c nastal broadcast $q \xrightarrow{c!,\alpha} q'$, přejde do nového stavu $q_0 \xrightarrow{c?,\alpha_0} q'_0, ..., q_n \xrightarrow{c?,\alpha_n} q'_n$, pokud jsou splněny jejich podmínky přechodu $F_G(q_0), ..., F_G(q_n)$ a zároveň invarianty nových stavů $F_I(q'_0), ..., F_I(q'_n)$.

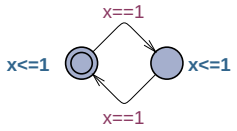
2.5 Deterministický časovaný automat (DTA)

Výše popsané systémy mají pro praktickou realizaci několik nevýhod. Jedna z těch nejvíce omezujících je, že jsou nedeterministické. Účelem převodníku je implementovat časovaný automat v nějakém formálním jazyce. Pro tento účel je velice důležité převést automat na deterministický.

Základní vlastnosti deterministického automatu jsou následující. Má pouze jeden počáteční stav. Pokud z jednoho stavu vychází více přechodů, musí být definovány jednoznačně. Nesmí být otevřen více než jeden přechod v jednom časovém okamžiku. [2] Nedeterministický automat může setrvat ve stejném stavu tak dlouho, jak mu to dovolí jeho invariant.

Deterministický automat je možné definovat obdobně jako nedeterministický, pouze místo proměnné nezáporné doby d zvolíme konstantní časovou jednotku δ a při libovolném splnění podmínky přechodu a invariantu následujícího stavu, dojde k přechodu do tohoto stavu.

Příklad jednoho z problémů, který může nastat v časovaném automatu je takzvaný *Zeno-timelock* [10]. V takovém případě je potřeba vykonat nekonečné množství akcí v konečném čase, což je nerealizovatelné.



Obrázek 1: Zeno-timelock

Pokud stav definujeme jako trojici (Q_C, C, V) a cestu v automatu jako posloupnost po sobě jdoucích stavů, pak NTA nemá pouze jednu cestu, ale množinu cest. DTA je redukce na jednu z těchto cest. [11]

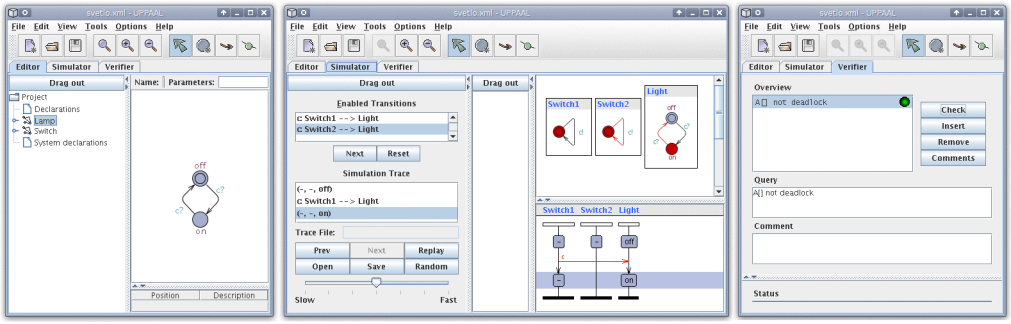
Pro přechod t do jiného stavu q' stačí u NTA, aby byly splněny podmínky $F_G(t)$ pro (C, V) a $F_I(q')$ pro (C, V'), protože při přechodu neplyne čas. U DTA je potřeba, aby invarianty platily pro následující časovou jednotku, $F_G(t)$ pro (C, V) a $F_I(q')$ pro (C', V'), protože při každém přechodu se posune čas minimálně o δ.

3 Nástroj UPPAAL

UPPAAL je vývojový nástroj pro vytváření, simulaci a verifikaci systémů modelovaných jako síť časovaných automatů. První verze byla vydána v roce 1995 a od té doby je UPPAAL stále intenzivně vyvíjen. Program je výsledkem spolupráce Uppsala University a Aalborg University. Automaty, které lze v UPPAALu popsat, jsou z principu nedeterministické.

Pokud je potřeba simulovat deterministické chování, je nutné zajistit, aby byly všechny podmínky přechodu určeny jednoznačně. To znamená, aby v jednu časovou jednotku byl otevřen vždy jen jeden přechod. Dále je potřeba se vyhnout

cestám, které mají vykonat více přechodů v nulovém čase. Je sice možné v implementaci takového automatu do FPGA nasimulovat rychlý sled přechodů v jedné časové jednotce, ale jen omezené délky, která musí být předem známa.



Obrázek 2: Nástroj UPPAAL, editace, simulace, verifikace

3.1 Reprezentace časovaného automatu v UPPAALu

3.1.1 Šablona (Template)

Základním stavebním prvkem jsou šablony automatů. Z těchto šablon lze vytvořit instance konkrétních automatů a ty vzájemně propojit.

3.1.2 Synchronizace (Binary synchronisation)

Synchronizace se děje pomocí proměnných typu `chan`. Například hrana s označením *c!* se synchronizuje s přechodem, který obsahuje *c?*.

3.1.3 Broadcast synchronizace (Broadcast channel)

Neblokující synchronizace `broadcast chan`, která synchronizuje všechny čekající *c?*. I v případě, že není žádné *c?* připraveno, je přechod otevřen.

3.1.4 Urgentní synchronizace (Urgent synchronisation)

Kanál je definován s prefixem `urgent`. Přechody, které používají urgentní kanály nemohou v podmínkách přechodu (`guard`) obsahovat proměnné s časem. Urgentní kanál musí synchronizovat okamžitě bez jakéhokoli zpoždění.

3.1.5 Urgentní stav (Urgent location)

V tomto stavu nesmí systém setrvat a musí ho hned opustit, žádný časový posun se v takovém případě neprovede.

3.1.6 Commited location

V tomto stavu podobně jako v urgentním nesmí systém setrvat a musí ho hned opustit, navíc předbíhá všechny urgentní stavy, které musejí neprodleně přejít do jiného stavu a je proveden před nimi.

3.1.7 Podmínka přechodu (Guard)

Podmínka, která musí být splněna, aby mohla být opuštěna aktuální pozice právě daným přechodem. Je to podmínka nutná, ne dostačující.

3.1.8 Přiřazení (Assign)

Při průchodu přechodem je možné přiřadit určené hodnoty lokálním a globálním proměnným.

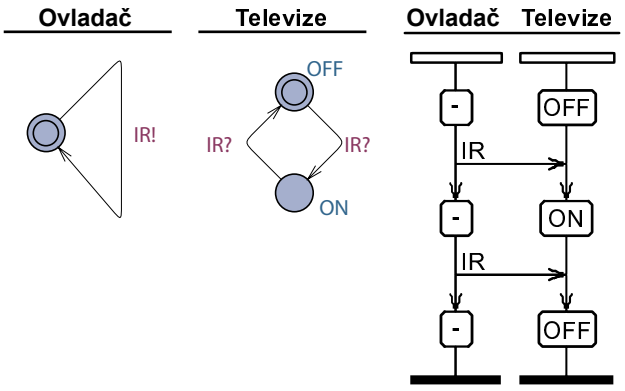
3.1.9 Invariant

Podmínka, za které může systém setrvávat v aktuálním stavu, nebo při které může do nového stavu vstoupit.

Po nadefinování všech šablon lze provést simulaci celé sítě. Od každé šablony lze odvodit libovolné množství instancí s různými inicializačními podmínkami.

3.2 Simulace sítě automatů

Na obrázku je znázorněna jednoduchá simulace dálkového ovladače a televize. V libovolném čase může ovladač vyslat signál a tím televizi zapnout nebo vypnout. Proměnná IR je typu `broadcast chan` aby byla synchronizace neblokující a simulovala tak reálnou situaci, kdy televizní ovladač nečeká na odezvu od televize.



Obrázek 3: Jednoduchá simulace dálkového ovladače na televizi.

V okně simulace je možné krokovat celý systém a vybírat aktivní přechody systému, pokud jich je v jednom okamžiku více na jednou otevřených. Dále je v okně vidět celá simulovaná cesta se zvýrazněním všech synchronizací. Na modelu dálkového ovladače je například vidět, že byla zapnuta a po blíže nespecifikované době opět vypnuta. Celou simulovanou cestu je možné uložit a znovu kdykoli načíst z externího souboru.

3.3 Verifikace

UPPAAL implementuje metodu pro formální verifikaci systému. V klasické verifikaci, prováděné symulací nebo testováním, není možné zaručit, že daná pomínka opravdu nenastane. U formální verifikace je model testován oproti formulím v temporální logice. Tím se zaručí ověření podmínky v celém rozsahu všech možných stavů. Pro zápis se používá Computation tree logic CTL* a její podmnožiny CTL a LTL [8]. CTL* se skládá z operátorů temporální logiky a kvantifikátorů cesty. Ve všech zápisech je možné používat logické operátory $\neg, \vee, \wedge, \rightarrow$.

CTL logika obsahuje pouze operátory

- E – Existuje
- A – Pro každé
- $\Diamond$ – Eventuálně (někde na cestě)
- $\Box$ – Globálně (všude na castě)
- $\circ$ – V dalším stavu

LTL logika obsahuje pouze operátory

- $\Diamond$ – Eventuálně (někde na cestě)
- $\Box$ – Globálně (všude na castě)
- $\circ$ – V dalším stavu
- $\rightsquigarrow$ – Předchází

V UPPAALu je obsažen verifikátor používající zápis ve formulích temporální logiky, který implementuje jinou podmnožinu CTL*. V jeho zápisu jsou povoleny pouze operátory $\Diamond$, $\Box$, A, E a $\rightsquigarrow$.

3.3.1 Dosažitelné vlastnosti (Reachability Properties)

Jsou nejjednodušší forma vlastností systému. Ukazují pouze na to, zda je možné v nějakém stavu systému, do kterého vede cesta z počátku, splnit danou vlastnost. Například při konstruování komunikačního protokolu od vysílače k přijímači je to dotaz typu, zda existuje možnost, že bude zpráva doručena. Tato verifikace neříká nic o správnosti protokolu, pouze informuje, že existuje možnost, že nějaká zpráva může být doručena.

Pro nějaký stav, ve kterém podmínka φ může být dosažitelná užitím výrazu $E\Diamond\varphi$. V UPPAALu se zapíše jako `E<>` φ.

3.3.2 Bezpečnostní vlastnosti (Safety Properties)

Říkají, že nic špatného se v systému nemůže stát. Například pro model jaderné elektrárny může být bezpečnostní vlastnost, že teplota nikdy nepřestoupí nad určitou mez. V UPPAALu jsou tyto vlastnosti definovány pozitivně ve smyslu, něco dobrého je vždycky pravda.

Pro stavovou formuli φ je zápis v podobě $A\Box\varphi$, kde φ je pravdivá ve všech dosažitelných stavech, nebo pro $E\Box\varphi$, kde φ je dosažitelné v nejdelší cestě. V UPPAALu se zapíše `A[]` φ resp. `E[]` φ.

3.3.3 Životnostní vlastnosti (Liveness Properties)

Říkají, že něco se eventuálně stane. Například při stisknutí tlačítka dálkového ovladače by se měla televize zapnout. Vlastnost říká, že formule může být splněna. $A\Diamond\varphi$, zápis v UPPAALu je pak `A<>` φ. Další vlastnost je *předchází* $\varphi \rightsquigarrow \psi$, která se v UPPAALu zapíše φ `-->` ψ. Tento zápis říká, že pokud nastane událost φ, mělo by následovat i splnění vlastnosti ψ. Například pokud eventuálně odešleme zprávu, měla by být doručena. [5]

4 Reprezentace časovaného automatu pomocí Xilinx System Generátotu a MATLABu

Za pomoci MATLABu, Simulinku a Xilinx System generátoru lze modelovat časované automaty. Výstup system generátoru je přímo použitelný pro programování FPGA.

4.1 Úvod do FPGA

FPGA (field-programmable gate arrays) jsou programovatelné logické obvody. To znamená, že je lze naprogramovat tak, aby vykonávaly takřka jakékoli logické operace (dokonce jakékoli číslicové operace). Jejich popis se vytváří na počítači v grafické nebo textové formě. Pro popis se nejčastěji využívají HDL (hardware

description language) jazyky jako je VHDL nebo verilog. Vzniklý popis se zkompiluje pomocí nástrojů výrobce a uloží do FPGA. Při změně funkce není třeba předělávat tištěný spoj, ani vyměňovat součástky, stačí natáhnout nová data do FPGA. Navrhovaný obvod může být rychlejší, než stejný, sestavený z dílčích součástek, protože jsou všechny prvky na jednom křemíkovém čipu. Obvod si musí po zapnutí načíst celý program z externí paměti do vnitřní RAM, aby mohl vůbec fungovat. U novějších modelů je program uložen ve vnitřní flash paměti, takže komplikace s načítáním odpadají.

4.2 Srovnání konkurenčních architektur

Oproti mikroprocesorům má FPGA mnoho rozdílů. V mikroprocesoru běží celý program postupně instrukce za instrukcí. V FPGA se vše děje paralelně. V FPGA lze vytvořit struktury, které budou provádět aritmetické operace. Takových struktur může být víc a všechny mohou běžet paralelně. Výsledkem je možnost dosažení větší rychlosti zpracování vstupních dat.

Na rozdíl od systémů sestavených z konstrukčních bloků, jako třeba integrované obvody řady 7400 a 4000, mají FPGA mnohem větší spolehlivost a navíc, díky malým rozměrům, jsou i rychlejší, protože se veškerá komunikace děje na jednom čipu a ne na desce s dlouhými cestami sběrnic.

Na rozdíl od ASIC (application-specific integrated circuit) zákaznických obvodů mají naopak výhodu v rychlejším vývoji, protože do FPGA stačí nahrát nové rozvržení a obvod je připraven k použití.

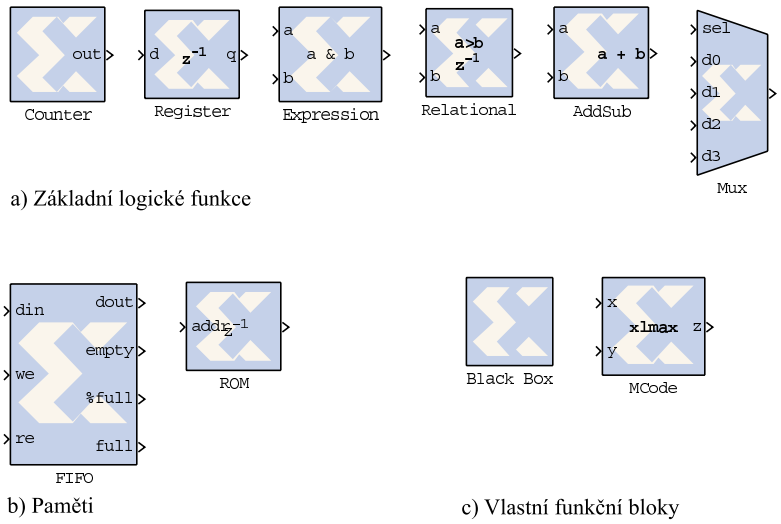
Nevýhodou FPGA oproti ASIC je zejména velká spotřeba. Ta plyne z toho, že ASIC jsou optimalizované pro jeden úkol a neobsahují zbytečné součástky. Navíc není zanedbatelná spotřeba statické paměti RAM, která u mnohých obvodů slouží pro uchování programu. Další nevýhodou mnoha typů je již zmíněná nutnost bootování po každem zapnutí přístroje.

4.3 Hlavní výrobci FPGA

Výrobci FPGA jsou hlavně Xilinx, Altera, Lattice, Actel a Atmel. Většina výrobců vyrábí již typy, které obsahují paměť flash, takže není třeba načítat data a obvod je již plně připraven hned po zapnutí napájení.

4.4 MATLAB a Xilinx System Generator

Převodník, který má za cíl tato práce, načítá vstupní data z výstupu UPPAALu a převádí je do jiné vhodné reprezentace. Pro tuto reprezentaci byl zvolen Xilinx System Generator (XSG). Pro každý automat vygeneruje převodník kód, který je přímo přiřazen funkčním blokům `MCode` z XSG Toolboxu. Bloky jsou použity v simulinkovém schématu a pomocí dalších částí z XSG jsou pospojovány. Z takto vytvořeného schematu se již přímo generuje výstup pro programování FPGA.



Obrázek 4: Výběr z nabídky funkčních bloků toolboxu System Generator

Tento převodník, který je cílem této práce, byl vytvořen pro převod automatů, které testují parametry řídících systémů. V následujícím textu se pokusím ukázat, jak vypadá realizace časovaného automatu v MATLABu a jak lze vytvářet komunikující sítě automatů.

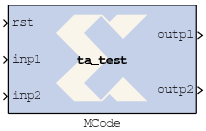
4.5 Základní struktury

Pro generování výsledných struktur se používá XSG toolbox. Schémata je možné tvořit v prostředí simulinku. Toolbox rozšiřuje simulink o řadu funkčních bloků. Na obrázku 4 jich je několik uvedeno.

Toolbox obsahuje základní logické funkce, bloky pro aritmetické operace, porovnávání a složitější bloky, jako jsou paměti ROM a RAM. Zajímavé jsou zejména bloky určené pro vytvožení vlastní funkce. `Blackbox` je určen pro HDL (Verilog/VHDL) a `MCode` je určen k popisu pomocí m-file. M-file se musí skládat z jedné funkce, jejíž hlavička vypadá podobně jako následující.

```
1 function [ outp1, outp2] = ta_test ( rst, intp1, inp2);
```

Ze vstupních proměnných se vytvoří vstupy bloku a z výstupních proměnných se vytvoří výstupy bloku.



Obrázek 5: Ukázkový blok m-code

Na konci funkce je ještě možné všechny výstupní proměnné přetypovat na vhodné typy pomocí funkce `xfix`, aby bylo možné napojit další funkční bloky.

```
state_new = xfix({xlUnsigned, 32, 0, xlTruncate, xlSaturate}, state_new);
sfd_new = xfix({xlBoolean}, sfd_new);
```

Celý automat je dále realizován jako blok `switch-case`.

```
% States
switch double( state )
        case 1 % first_state
                state_new = 2;
        case 2 % second_state
                state_new = 3;
        case 3 % last_state
                state_new = 3;
        otherwise % an error occured
                state_new = 1;
end
```

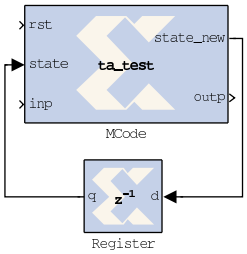
Důležitá část celého automatu je reset, který je realizován následujícím kódem. Vnitřní část je shodná s tělem `otherwise` u `switch-case`.

```
% Reset
if( rst )
        state_new = 1;
end
```

Proměnné, které se mají v automatu uchovat, je možné uchovávat dvojím způsobem. První z nich je pomocí persistantních stavových proměnných.

```
persistent state_i, state_i = xl_state(1, {xlUnsigned, 16, 0});
```

Druhou možností reprezentace je přes vstupně výstupní proměnné a blok registru z^{-1}. Při každém hodinovém impulzu se funkce spouští a na její vstup je přiváděna hodnota, která byla na výstupu při předchozím tiku hodin.

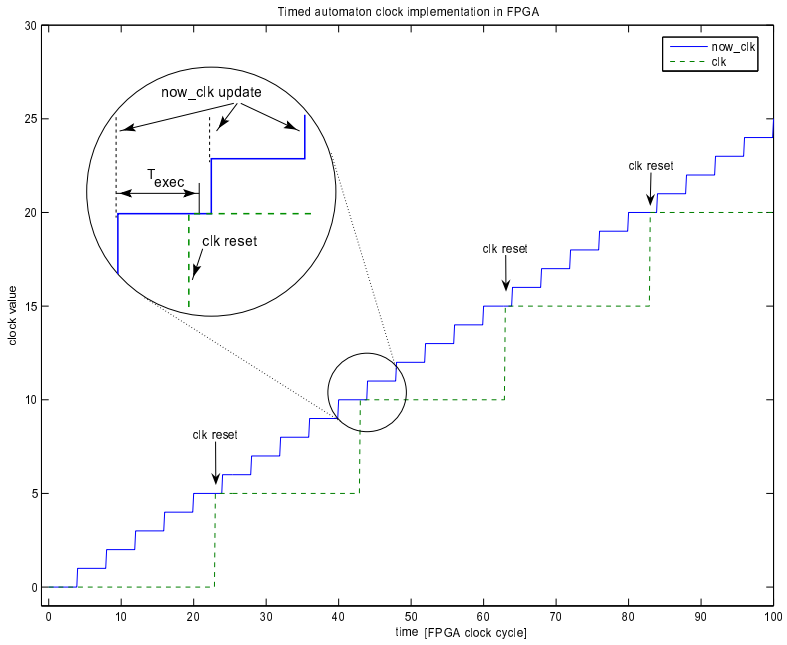


Obrázek 6: Uchování vnitřních proměnných automatu pomocí vnějšího registru

4.6 Časové proměnné

Ve výsledné realizaci je použito vnitřních hodin, které jsou dále děleny děličkou kmitočtu. Teprve tato vydělená frekvence je brána jako hodiny pro časované automaty. Tímto způsobem je umožněno vykonat více přechodů najednou v jednom tiku vydělených hodin. Vzniká tu ale možnost, že dělení nebude dostatečné a systém bude požadovat vykonání více přechodů, než na které byla dělička nastavena. V rámci vydělené frekvence se přechody chovají, jako by proběhly v jeden

okamžik, protože celý systém je synchronizován frekvencí vyšší. Pokud ale nastane situace většího požadavku, než je možné zpracovat, budou další přechody vykonány v dalším tiku hodin.



Obrázek 7: Implementace hodin v časovaném automatu

Globální hodiny, ze kterých jsou všechny automaty synchronizovány, běží nepřetržitě. Kvůli správné funkci porovnávání času jsou hodiny uloženy v 64bit proměnné. Při zjišťování časového zpoždění se postupuje podle následujících pravidel. Před začátkem měření se vynulují lokální hodiny tím, že se do nich nastaví aktuální hodnota globálních hodin. Ve větvi, která má hodiny využívat, se testuje rozdíl globálních hodin a lokální časové proměnné. Pokud tento rozdíl překročí danou mez, je vyhodnoceno překročení určitého času.

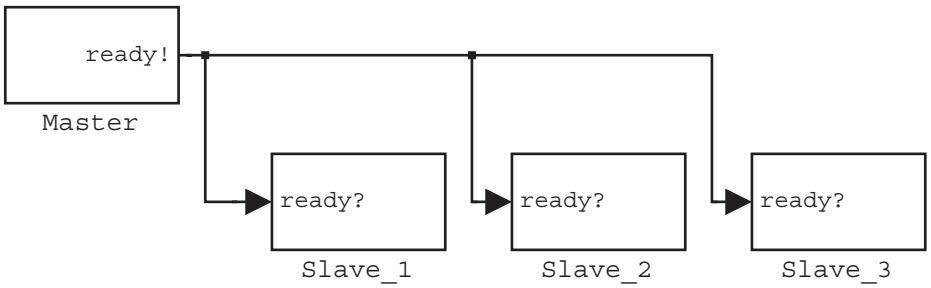
Porovnávání dlouhých čísel trvá dlouho i v FPGA a trvá to dokonce déle, než jedna časová jednotka automatu. Je proto důležité před stavy porovnávající čas vložit předřadný stav, který má stejné tělo jako vyhodnocovací podmínka, přes kterou se do aktuálního stavu vstupuje. Tento stav prodlouží aktuální stav. Tím se eliminuje vliv dlouhého porovnávání.

4.7 Synchronizace

Jednou z důležitých vlastností časovaných automatů, kterou je nutno implementovat, jsou synchronizační kanály. Ty se podle definice časovaného automatu dělí na dva základní druhy – obyčejný a broadcast. V této implementaci se všechny

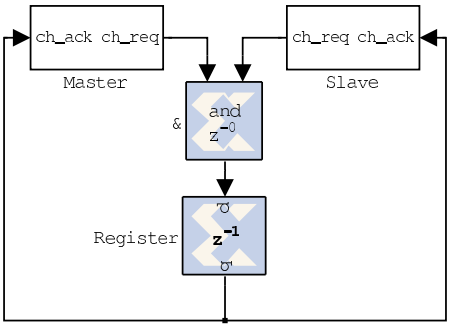
přechody chovají jako urgentní, protože pokud jsou splněny všechny podmínky přechodu, přejde systém okamžitě do dalšího stavu.

Pokud je signál typu broadcast, je automat, který vyšle po tomto kanále aktivační impulz, připraven k další činnosti a na nic nečeká. U broadcast synchronizací není třeba posluchače a přesto proběhne vyslání synchronizace korektně. Implementace takového kanálu je pouhé rozvedení synchronizačního pulzu do všech bloků, které to vyžadují.



Obrázek 8: Bloková ukázka broadcast synchronizace v simulinku

Obyčejný synchronizační kanál se chová odlišně. Pro jeho správný průchod je potřeba posluchač a to v této implementaci právě jeden. Není možné spustit více posluchačů najednou. Zavedením náhodnosti do jejich výběru by se zavedl další nedeterminismus a automat by se choval nepředvídatelně, což není žádoucí.



Obrázek 9: Synchronizace s logickým AND

Obyčejný synchronizační kanál lze realizovat hned několika zapojeními. Pro vyslání synchronizace se používá výstupní pin bloku se jménem končícím na `_req`, pokud je tento pin přímo propojen přes registr na svůj vlastní odpovídající pin `_ack`, jde o broadcast kanál a z výstupu se odebírá signál pro další automaty. Při zapojení jednoduchých logických funkcí a nebo přímým připojením na posluchače s mírnou úpravou vnitřního bloku příslušného `case` lze vytvořit blokující synchronizační kanál, který čeká na odpověď. Dokud vysílač nepřijme odpověď, nemůže pokračovat dál a dokud posluchač nepřijme výzvu, nemůže také pokračovat dál.

Použitý útržek kódu pro synchronizace: vysílač synchronizace

```
case 1 % dec_bellow
        if ( impact_chAck )
                state_new = 2;
                impact_chReq = false;
        else
                state_new = state;
                impact_chReq = true;
        end
```

a přijímač broadcast synchronizace.

```
case 1 % check_sensor
        if ( impact_chAck )
                state_new = 4;
                timer_new = gtimer;
        else
                state_new = state;
                timer_new = timer;
        end
```

Přijímač obyčejné synchronizace vypadá stejně jako vysílač. Jde o blokující zapojení, kde přes logický součin je vyhodnocováno, jestli oba přechody žádají o synchronizaci. Pokud ano, je výsledkem logického součinu jednička, která je zavedena na vstupy `_ack`.

5 ANTLR

Pro celou práci je klíčový převod z jedné textové reprezentace na jinou. Parsování výstupního souboru UPPAALu přímo by bylo náročné, proto byl zvolen generátor ANTLR, který po předložení gramatiky vytvoří příslušný parser automaticky.

5.1 Překladače

Nejčastěji bývá rozdělen do několika částí, z nich první dvě jsou lexikální analyzátor a parser.

5.1.1 Lexikální analyzátor

Načítá vstupní soubor a podle zadaných pravidel vytváří tokeny. V gramatice pro lexer je například uvedeno, jak vypadá identifikátor, jak číslo, jak bílá mezera a pod. Tyto tokeny dále postupují do další jednotky.

5.1.2 Syntaktický analyzátor

Přebírá tokeny ze vstupu lexeru a porovnává je s vlastní gramatikou. V té je již možné vytvořit složitější pravidla. Parser navíc potřebuje informace o dalším minimálně jednom tokenu, aby správně vyhodnotil ten stávající. Nejznámější druhy parsovacích algoritmů jsou LL(k) a LR(k), kde k udává počet tokenů, které musí načíst napřed.

LL – Pro správně navržený jazyk by měla jít napsat LL(1) gramatika. O to se zajímal například Niklaus E. Wirth, který navrhl například jazyk pascal.

LL parser je typu shora-dolů (top-down). Nejprve se snaží vyhodnocovat obecnější pravidla, postupně přichází až na úroveň jednotlivých tokenů. LL parser je docela jednoduché napsat ručně, protože je práce pro člověka logická.

LR – LR parser je pro člověka méně intuitivní. Používá přesně opačné techniky parsování, tedy odspoda nahoru (bottom-up). Nejprve vyhodnocuje nejjednodušší pravidla, a pak teprve hledá složitější, ve kterých je to jednodušší obsaženo. Příkladem je například program yacc nebo GNU bison, který používá modifikovanou verzi LR algoritmu (LALR). Parsery generované těmito nástroji jsou pro člověka těžko čitelné, protože obsahují velké množství parsovacích tabulek.

5.2 Výběr parser generátoru

Nabízelo se několik variant, jak vytvořit parser. Nejznámnější nástroje jsou:

- ručně LL(K)
- yacc (GNU bison) – LALR
- JavaCC – LL(k)
- ANTLR – LL(k)

Člověk je většinou schopen napsat parser typu LL(k), protože je jeho tvorba intuitivní. Nevýhodou ručně generovaného parseru je zdlouhavé psaní a navíc, toto řešení není vůbec odolné proti drobným změnám gramatiky vstupních dat. Pokud nastane změna, je třeba přepsat větší část parseru. Naopak je možné touto cestou vybudovat parser, který je rychlý a optimální.

Nevýhoda bisonu je, že generuje kód nečitelný pro člověka. Jeho hlášení chyb je mnohem méně výřečné a věcné než u ostatních produktů. Pro mě nejzajímavější je projekt parser generátoru ANTLR.

5.3 ANTLR (ANother Tool for Language Recognition)

Jde o LL(k) parser. Za celým tímto systémem stojí profesor z university v San Francisku Terence Parr. Vytváří tento program již od roku 1989.

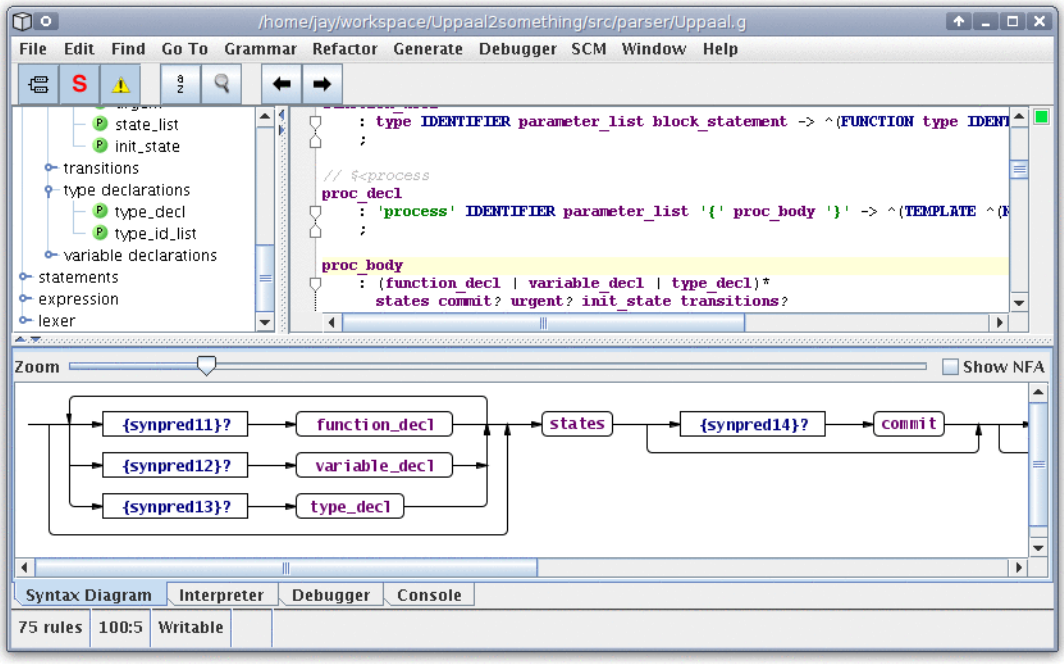
Vygenerovaný kód je přehledný a není příliš obtížne pro něj psát gramatiku. Pro zápis gramatiky se využívá rozšířené BNF syntaxe EBNF, která se v různých verzích trochu liší a zpřehledňuje. (Mimochodem na syntaxi podobné EBNF je založen i soubor DTD pro popisování značek XML.) Při vytváření gramatiky je potřeba ctít některá pravidla a nevytvářet pravou rekurzi v zápisech. Momentálně stabilní verze ANTLR je 2.7.x, ale v blízké budoucnosti by měla vyjít i finální

verze ANTLR 3. Ten má přehlednější zápis a je k němu vytvořeno i poměrně pěkné vývojové prostředí ANTLRWorks.

5.4 Nástroje použitelné s ANTLR

5.4.1 ANTLRWorks

Hlavní důvod použití ANTLR byl vývojový nástroj pro gramatiku ANTLRWorks. Obsahuje v sobě přehled všech pravidel, která jsou v gramatice použita. Pravidla jde dále shlukovat do skupin, takže se tím velice usnadní orientace v kódu. V hlavní části okna je vidět zdrojový kód gramatiky, který je samozřejmě barevně zvýrazněn. Ve spodní části se podle potřeby ukazují syntax diagram a debuger.



Obrázek 10: ANTLRWorks – nástroj pro vývoj gramatiky

5.4.2 StringTemplate

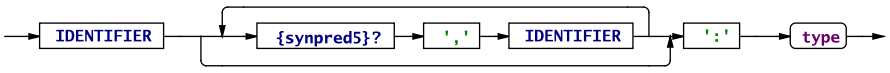
Jde o knihovnu, kterou lze přímo napojit na výstup ANTLR a generovat s její pomocí textový výstup. Je možné napsat několik gramatik pro StringTemplate, které budou moci například převádět vstupní text do více formátů, bez většího zásahu do kódu.

5.5 Ukázková gramatika

V následujícím listingu je ukázáno jedno pravidlo pro parser.

```
1 variable : IDENTIFIER ( ',' IDENTIFIER )* ':' type
2          ;
```

Kde `variable` je název pravidla, `IDENTIFIER` je lexikální pravidlo, protože je psáno velkými písmeny a je definováno v části pro lexer a `type` je pravidlo, které vyhodnocuje typ proměnné a je definováno v části pro parser. Dále jsou v pravidle použity standardní operátory pro definování počtu opakování jako `*`, `+`, `?`. ANTLR částečně kombinuje lexikální a syntaktickou analýzu. V pravidle se například vyskytují tokeny jako `':'`, což je určitě lexikální pravidlo. Je tu proto, že je tento token nevýznamný a pouze určuje, jaké pravidlo se má použít. Žádnou další užitečnou hodnotu nenese.

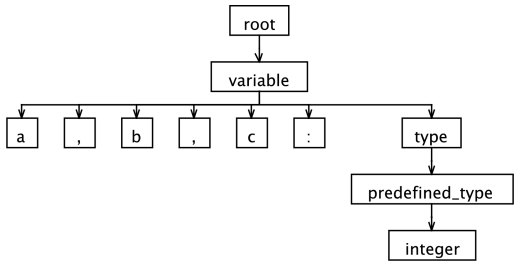


Obrázek 11: Grafické znázornění pravidla v ANTLRWorks

Tomuto pravidlu například vyhoví vstup

```
1 a, b, c : integer
```

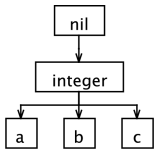
Výstup parseru lze zvolit v hlavičce gramatiky. Pro zpracovávání velkých souborů je optimální brát z parseru token po tokenu a rovnou tento výsledek zpracovávat. Bohužel je k této metodě potřeba více představivosti. Další možností je, aby ANTLR rovnou vygeneroval parsovací strom v hierarchii, v jaké jsou pravidla zapsána. Po zavolání parseru je pak k dispozici kořen tohoto stromu a celkově je ve zpracování lepší orientace.



Obrázek 12: Parsovací strom aplikování popisovaného pravidla na příklad

Je to něco podobného jako SAX parser a DOM parser pro XML soubory. ANTLR ale na tomto místě nekončí a nabízí ještě možnost přepisovacích pravidel přímo v gramatice. Následující příklad demonstruje stejné pravidlo s přidáním přepisovací části. Ta se zaslouží o redukci parsovacího stromu podle našich potřeb.

```
1 variable : IDENTIFIER ( ',' IDENTIFIER )* ':' type
2            -> ^(type IDENTIFIER+)
3          ;
```



Obrázek 13: AST po aplikování popisovaného přepisovacího pravidla na příklad

Výsledkem tohoto pravidla je strom AST (Abstract Syntax Tree), se kterým se pracuje nejlépe, protože už je předpřipraven k dalšímu použití.

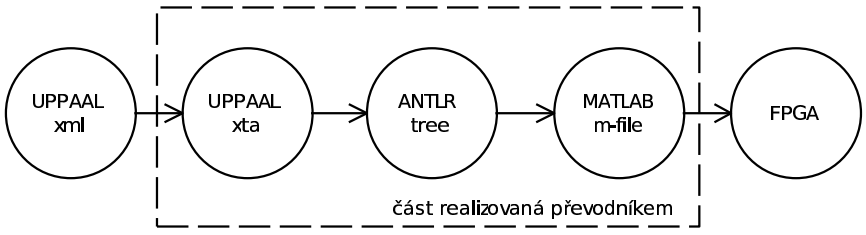
Veškeré tyto kroky by se těžko zjišťovaly naslepo, proto je velice výhodné používat ANTLRWorks. Integrovaný debuger obsahuje mnoho možností zvýraznění pravidel, výsledných stromů, krokování gramatiky a pod.

5.6 Výsledky generování

Po celém procesu odlaďování gramatiky je k dispozici několik možných výstupů. ANTLR umí generovat kód pro několik jazyků jako například Java, C++, Python a C#. Kvůli přenositelnosti kódu byl zvolen jako výstupní jazyk Java. Dále kvůli jednoduchosti vývoje byl zvolen jako výstup AST i přesto, že je více paměťově náročný.

Výsledný parser má několik zajímavých vlastností. Jak již bylo popsáno výš, jedná se o parser LL(k), dokonce o LL(*), což znamená, že čte tokeny dopředu tak dlouho dokud si není jistý, že je to právě a jen jedno konkrétní pravidlo. Dále v sobě obsahuje generátor chybějících tokenů, takže pokud v parsovaném textu chybí nějaký znak, který nemůže ovlivnit rozhodování o tom, jaké pravidlo se má na daný úsek aplikovat, je tento token doplněn. Je to výhodné například, když ve zdrojovém textu chybí středník. V takovém případě se pouze nahlásí warning a pokračuje se dál se správným výsledným stromem.

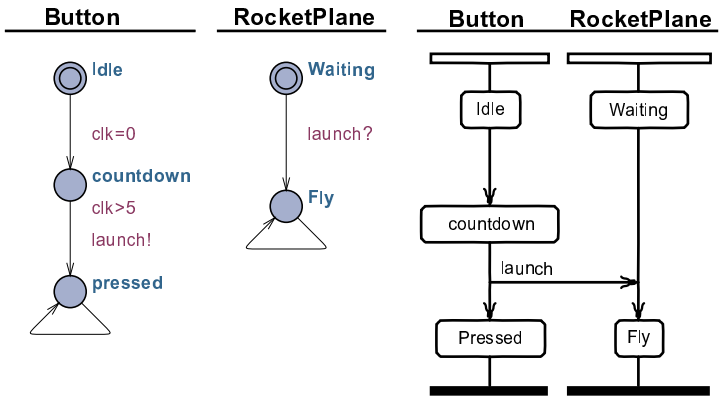
6 Převod UPPAAL → FPGA



Obrázek 14: Schéma kroků převodníku

6.1 Grafická reprezentace v UPPAALu

Na schématu je znázorněno jednoduché schéma dvou automatů. Jsou synchronizovány synchronizačním kanálem `launch`. Automat, který představuje tlačítko navíc obsahuje podmínku, že není možné synchronizaci provést dřív než za 5 tiků hodin.



Obrázek 15: Schéma časovaného automatu a schéma běhu automatu

6.2 Formát UPPAALu (XML-XTA)

Schéma z UPPAALu je uloženo v XML souboru, který je definován vlastním `DTD Flat System 1.1`. Příklad obsahu takového souboru je uveden v listingu. Obsah souboru se skládá z několika částí, které odpovídají logickému členění. První část jsou deklarace globálních proměnných, vlastních typů a funkcí. Kvůli přílišné složitosti není v převodníku implementována možnost překladu vlastních funkcí a typů.

Hlavní část je oblast `template`, která obsahuje definice jednotlivých stavových automatů. Důležité jsou hlavně části `location`, reprezentující stav a `transition`, reprezentující přechod. V souboru jsou i informace o poloze na schématu, které ale nejsou pro převod důležité.

Poslední důležitou částí je sekce `system`, ve které se provede vlastní propojení instancí templateů. Tato část je také ignorována a je potřeba ji ručně vytvořit v MATLABu jako simulinkové schéma. Převodník sám při běhu vypisuje na standardní chybový výstup, co nepodporuje, aby mohl tvůrce automat příslušně upravit pro potřeby převodu.

```
1 <?xml version='1.0' encoding='utf-8'?>
2 <!DOCTYPE nta PUBLIC '-//Uppaal Team//DTD Flat System 1.1//EN'
3  'http://www.it.uu.se/research/group/darts/uppaal/flat-1_1.dtd'>
4 <nta>
5     <declaration>// Place global declarations here.
6         chan launch;
```

```
7           clock clk;
8       </declaration>
9       <template>
10          <name>RocketPlane</name>
11          <location id="id3" x="-32" y="32">
12              <name x="-16" y="16">Fly</name>
13          </location>
14          <location id="id4" x="-32" y="-128">
15              <name x="-16" y="-144">Waiting</name>
16          </location>
17          <init ref="id4"/>
18          <transition>
19              <source ref="id3"/>
20              <target ref="id3"/>
21              <nail x="0" y="64"/>
22              <nail x="-64" y="64"/>
23          </transition>
24          <transition>
25              <source ref="id4"/>
26              <target ref="id3"/>
27              <label kind="synchronisation" x="-16" y="-64">launch?</label>
28          </transition>
29      </template>
30      <system>
31          // Place template instantiations here.
32          // List one or more processes to be composed into a system.
33          system Button, RocketPlane;
34      </system>
35  </nta>
```

6.3 Přestupní formát UPPAALu (XTA)

Aby mohl celý převod fungovat jednodušeji, provede se nejprve transformace vstupního souboru do formátu XTA. Ten obsahuje pouze informace o samotném automatu a je očištěn od přebytečných informací o poloze jednotlivých přechodů a stavů v grafické reprezentaci. Po převodu předchozího zkráceného příkladu vznikne následující text. Tento formát je odvozen z dokumentace o Uppalu [4].

```
1  // Place global declarations here.
2  chan launch;
3  clock clk;
4
5  process RocketPlane () {
6          // declarations
7
8          // states
9          state id3 { name Fly; } , id4 { name Waiting; };
10
11         // initial state
12         init id4;
13
14         // transitions
15         trans id3 -> id3 { }, id4 -> id3 { sync launch?; };
16 }
17
18 // Place template instantiations here.
19 // List one or more processes to be composed into a system.
20 system Button, RocketPlane;
```

Transformace byla původně řešena technikou „by hand“. To se ukázalo jako nešťastné a málo přizpůsobivé. Druhá verze byla vytvořena pomocí XSL transformace. Obě třídy (`Xml2XtaByHand`, `Xml2XtaXslt`) jsou zděděné ze stejného předka `Xml2Xta`, takže jsou v kódu vzájemně zaměnitelné.

```
1 Xml2Xta xta = new Xml2XtaByHand(fileName);
```

nebo

```
1 Xml2Xta xta = new Xml2XtaXslt(fileName);
```

Protože byla na parsování XML použita knihovna Xerces, bylo zapotřebí definovat lokální zdroje DTD pro vstupní soubory. Lokální kopie jsou důležité ve chvíli, kdy je program offline, protože jinak by skončil chybou. O určení, od kterých DTD existuje lokální kopie, se stará třída `UppaalEntityResolver`,

```
1 public class UppaalEntityResolver implements EntityResolver {
2     public InputSource resolveEntity(String publicId, String SystemId) {
3       if ("-//Uppaal Team//DTD Flat System 1.1//EN".equals(publicId))
4         return new InputSource("flat-1_1.dtd");
5       return null;
6     }
7 }
```

která je nastavena při vytváření objektu `DOMParser`.

6.4 Parsování XTA

Výsledný zdrojový text z předchozích kroků je použit jako vstup parseru. Pro parser byla napsána gramatika inspirovaná gramatikou pro BISON z knihovny UTAP. Jako parser generátor byl zvolen program ANTLR. Samozřejmě bylo možné vytvořit parser ručně od základu, ale to by bylo velice náročné i při malé změně gramatiky by bylo potřeba dělat velké zásahy do programu. V případě generovaného parseru stačí upravit gramatiku a funkce, které s výstupem parseru pracují, nemusejí vůbec nic poznat.

ANTLR se inicializuje následovně: nejprve se vytvoří instance lexeru, která provede lexikální analýzu. Poté je výstup z lexikální analýzy předán parseru, který ze vstupních dat vytvoří v paměti stromovou strukturu AST (Abstract Syntax Tree). Z paměťové reprezentace je dále jednoduše možné generovat libovolný výstup.

```
1 lexer = new UppaalLexer(stream);
2 tokens = new CommonTokenStream(lexer);
3 parser = new UppaalParser(tokens);
```

6.5 Generování m-file

Generování výstupního souboru se děje ve dvou průchodech stromem. Při prvním průchodu se spočítají a označí veškeré použité proměnné. Tyto informace následně slouží k optimalizaci generovaných funkcí. Ve výsledné funkci nemusí být tolik parametrů a celkově je výstupní kód přehlednější.

Druhá fáze převodu by se dala zapsat následujícím algoritmem.

while existuje další *template* **do**
  zjištění globálních a lokálních *proměnných*
  vytvoření *hlavičku* funkce
  inicializace proměnných
  vytvoření *switch*
  while existuje další stav **do**
    vytvořen jeden *case*
    while existuje další podmínka přechodu **do**
      vygeneruj *if*
    end while
    seřazení podmínek podle jejich konkrétnosti
  end while
  vypsání podmínky pro reset
  přetypování výstupních proměnných
end while

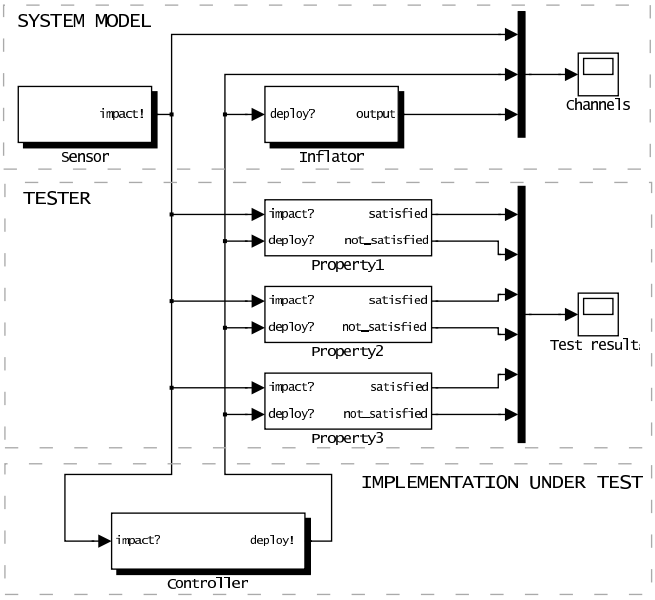
U stavů, které potřebují k vyhodnocení podmínky čas, je třeba vkládat předřadný stav, který zaručí správnost výsledku. Při samotném převodu probíhá několik optimalizací kódu. Jedna z nich je již jmenovaná eliminace nepotřebných proměnných. Dále probíhá převod z nedeterministické verze automatu na deterministickou. Všechny náhodnosti jsou předem vyloučeny a nedostatečně určené přechody mohou vypadnout z výstupu stavového automatu, protože jsou nedosažitelné. Je proto potřeba již při návrhu dbát na dostatečné podmínění všech přechodů, aby tato situace nenastala.

V následujícím listingu je vidět příklad výstupu překladače do m-file.

```
1 % Matlab M-file for timed automaton "RocketPlane"
2 % Automaton variables:
3 % global variables
4 %       clk
5 %       launch
6 % local variables
7 %       state
8 function [ state_new, launch_chReq] = RocketPlane (...
9 rst, state, launch);
10
11 % variables init
12 state_new = state;
13 launch_chReq = launch;
14
15 % States
16 switch double( state )
17         case 1 % Waiting
18                 if ( launch )
19                         state_new = 2;
20                         launch_chReq = true;
21                 else
22                         state_new = state;
23                         launch_chReq = launch;
24                 end
25         case 2 % Fly
26                 state_new = 2;
```

```
27                launch_chReq = launch;
28        otherwise % an error occured
29                state_new = 1;
30                launch_chReq = false;
31 end
32
33 % Reset
34 if( rst )
35        state_new = 1;
36        launch_chReq = false;
37 end
38
39 % variables casts
40 state_new = xfix({xlUnsigned, 32, 0, xlTruncate, xlSaturate}, state_new);
41 launch_chReq = xfix({xlBoolean}, launch_chReq);
```

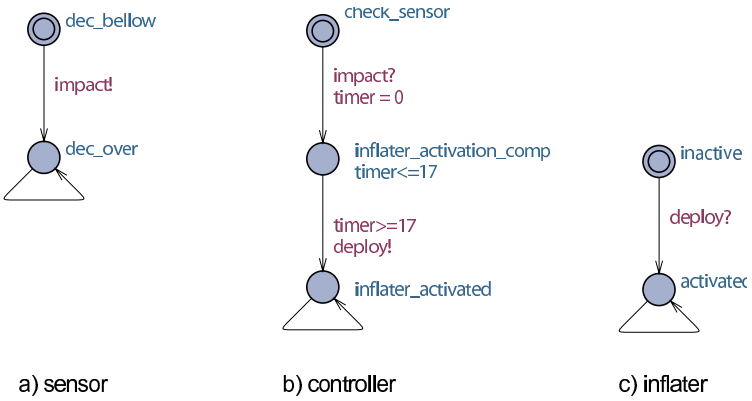
7 Popis konkrétního použití – bezpečný systém



Obrázek 16: Schéma celkové koncepce

7.1 Celková koncepce

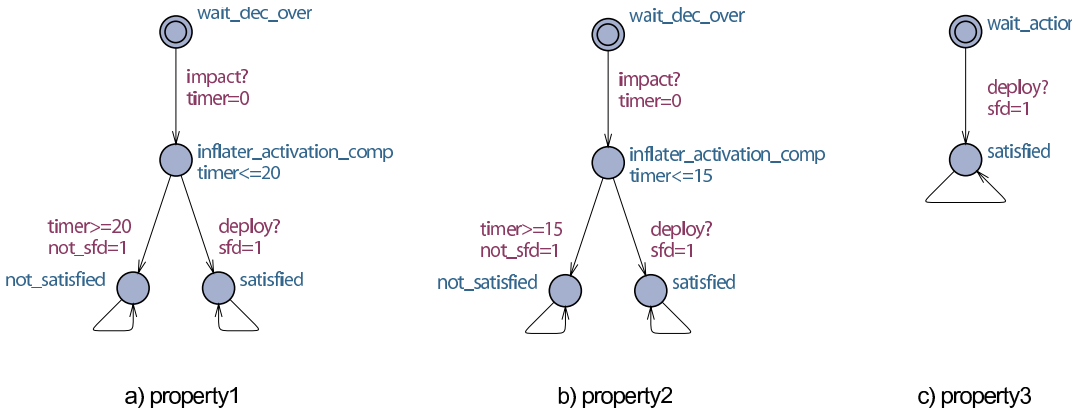
Schéma je rozděleno na tři části. První část je model systému, který obsahuje senzor a akční člen. Druhá část je tester sestavený z časovaných automatů, který sledují chod řídící jednotky. Poslední část je testovaná řídící jednotka. Výsledek celého testování se odebírá z prostřední části. Vyhodnocuje se, které podmínky uspěly a které ne.



Obrázek 17: Model soustavy

7.2 Testovaná řídící jednotka

Řídící jednotka je implementována pomocí časovaného automatu a jeho struktura je znázorněna na obrázku b). Ve chvíli, kdy je senzor aktivován, vyšle po synchronizačním kanalu *impact* signál, že došlo k aktivaci. Tento kanál způsobí otevření přechodu **check_sensor** → **inflater_activation_comp**. V tomto stavu je vypočítávána akce, simulovaná časovým zpožděním. Ve chvíli, když je akce spočítána, je vyslán po synchronizačním kanálu *deploy* příkaz akčnímu členu, kde je aktivován přechod **inactive** → **activated**.



Obrázek 18: Testovací podmínky

7.3 Testovací podmínky

Pro tento konkrétní případ odpovídá každá časová jednotka 1 μs. Tester se skládá ze tří testovaných podmínek, které jsou popsány následovně:

Property 1 kontrolér stihne odpálit airbag do 20 μs od okamžiku, kdy senzor vyšle signál. Ve formuli pro časový automat je tato podmínka zapsána jako

$A\Box t <= 20$ a znázorněna na obrázku a)

Property 2 kontrolér stihne odpálit airbag do 15 μs od okamžiku, kdy senzor vyšle signál. Ve formuli je tato podmínka zapsána jako $A\Box t <= 15$ a znázorněna na obrázku b)

Property 3 Existuje stav, ve kterém je airbag odpálen? $E\Diamond activated$. Je znázornět na obrázku c)

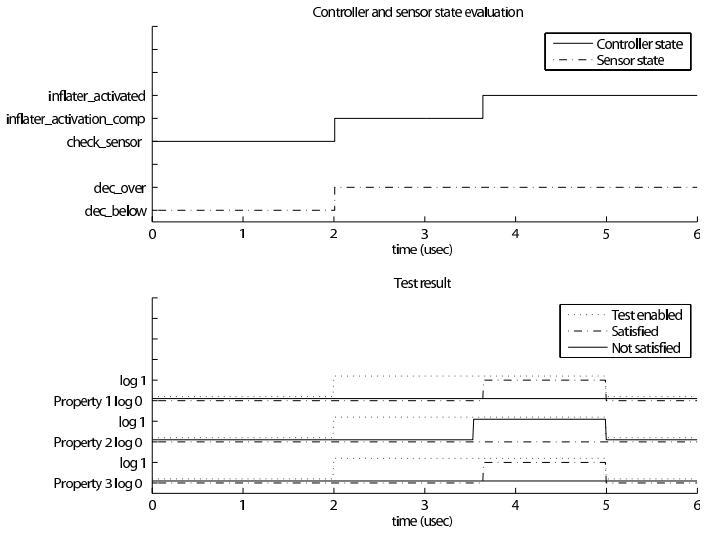
7.4 Výsledek

Pomocí testovacích podmínek bylo vyhodnoceno:

Property 1 je splněna, protože kontrolér stihne zareagovat do 20 μs

Property 2 není splněna, protože kontrolér nestihne zareagovat do 15 μs

Property 3 je splněna, protože kontrolér zareaguje.



Obrázek 19: Časové průběhy stavů a výstupních podmínek

7.5 Zhodnocení testu

Jeden možný příklad sestavený pomocí časovaných automatů je ověření bezpečnosti řídící jednotky airbagu. Výsledkem testu je, jestli řídící jednotka stihne, nebo nestihne vypustit airbag včas. V tomto případě stihne jednotka vypustit airbag nejpozději do 20 μs.

8 Další možnosti a nedořešené problémy

Program je připraven pro další rozšíření, která by zlepšila jeho funkčnost, nebo zvýšila schopnost odolávat špatně navrženým automatům. Momentálně nejsou tyto vlastnosti implementovány.

8.1 Generování do jiných jazyků

Díky použitému způsobu generování parseru pomocí parsergenerátoru ANTLR je možné výstup přímo předávat další knihovně, která jde ruku v ruce s ANTLR a to StringTemplate. Nynější verze ANTLR jsou na StringTemplate postaveny. Při správném použití by stačilo příkazem změnit pouze vzor (template) a program by generoval výsledek například přímo v jazyce pro popis hardware jako je VHDL nebo Verilog. Výstupem by také mohl být formát, který je výsledkem nástroje pro navrhování stavových automatů a který je přímo součástí Xilinx ISE.

8.2 Lepší kontrola

V konkurenčních projektech je kladen mnohem větší důraz na přesné chování výsledného automatu. Z toho důvodu jsou striktně kontrolována nedeterministická chování. Například pokud ve dvou větvích z jednoho uzlu může dojít k přechodu při stejné podmínce, je tato skutečnost vyhodnocena jako chyba a automat není vytvořen, dokud uživatel nedodefinuje striktnější pravidla pro přechod. Bohužel je nepraktické dodefinovávat všechny možné stavy a v nástroji UPPAAL chybí něco jako „else“, aby se daný přechod splnil jen tehdy, když ostatní jsou nesplněny.

8.3 Nedeterministické chování

Oproti předešlému požadavku je naopak možné zavést do systému ještě více nedeterministického chování. V nynější podobě převodník nepodporuje obyčejnou synchronizaci více posluchačů, protože není žádný mechanizmus, jak povolit přechod jen jednomu z nich. Tomu by právě mohlo pomoci náhodné chování. Generátor náhodných čísel by podle požadavku vybral jeden z několika připravených přechodů. Podobná situace nastává, pokud je na výstupu z uzlu povoleno více přechodů najednou. Stejný generátor náhodných čísel by mohl pomoci vybrat, jakým z těchto přechodů se má pokračovat dále. Momentálně se vybere v podstatě náhodně jeden z přechodů již při převodu formátů a ostatní možnosti jsou předem vyloučeny.

8.4 Optimalizace

Aby bylo generování co nejjednodušší, nekontroluje se reálné využití proměnných v automatu. Z toho plyne, že proměnné jsou využívány neoptimálně. Například v automatu, ve kterém běží čas, ale je často resetován, není potřeba pro běh časové proměnná tak velkého rozsahu, jako jsou momentálně používány, ale řádově mnohem menší. V mnoha případech je použita proměnná typu 64 bitové číslo. Pokud například v reálném systému dosahuje časová proměnná hodnoty nejvýše 15,

použijí se na její reprezentaci jen první 4 bity a 60 jich je stále nulových. Program s nimi ale nadále počítá a bere je v úvahu.

Pokud by se proměnné optimalizovaly, napravily by se tím některé problémy, které nastaly při implementaci v základním rozsahu. Například by odpadla nutnost generování přechodného stavu, protože porovnání s časem by bylo stejně rychlé, jako ostatní logické operace.

Další problém nedeterministického chování je počet přeskoků které je automat schopen udělat v „nulovém“ čase. Definovat toto číslo není dost dobře možné, navíc může být v systému timelock, ve kterém proběhne nekonečné množství přechodů za nulový čas (teoreticky), prakticky to není realizovatelné.

9 Závěr

Výsledkem práce je program, který umí načítat soubory generované pomocí UPPAALu a dále je převádí na reprezentace časovaných automatů pomocí m-file. Výsledný program byl testován na několika časovaných automatech vytvořených v UPPAALu a výsledek převodu byl simulován v MATLABu. Výsledky simulací v MATLABu byly podle očekávání stejné jako simulace přímo v UPPAALu. V převodníku není implementována žádná náhodnost. Nedostatečně definované podmínky přechodu mohou způsobit, že do dané větve automat nikdy nepřejde, nebo v rámci optimalizací bude větev úplně odstraněna. Pomocí toolboxu Xilinx System Generátor jsou vytvořena zdrojová data pro FPGA, která dále slouží pro testování a kontrolu chování řídících systémů, které byly testovány postupně v UPPAAL a v MATLABu.

Vedlejším efektem této práce bylo vytvoření převodníku *Texy!* → LaTeX kvůli snadnější orientaci v psaném textu a automatizování některých rutinních postupů, jako je například psaní nedělitelných mezer před jednopísmenné předložky. Texy syntaxe byla pro tento účel rozšířena o zápis vzorců.

Literatura

[1] Alur, R. Timed automata. In *Computer Aided Verification* (1999).

[2] Alur, R., a Dill, D. L. A theory of timed automata. *Theoretical Computer Science 126*, 2 (1994), 183–235.

[3] Alur, R., a Dill, D. L. Automata-theoretic verification of real-time systems, 1996.

[4] Behrmann, G. Utap - uppaal timed automata parser library. 2007 [cit. 2007-06-06]. URL: ⟨http://www.cs.auc.dk/~behrmann/utap/⟩.

[5] Behrmann, G., David, A., a Larsen, K. G. A tutorial on uppaal. M. Bernardo and F. Corradini, Eds., no. 3185 in LNCS, Springer–Verlag, pp. 200–236.

[6] Bengtsson, J., a Yi, W. *Timed Automata: Semantics, Algorithms and Tools*. PhD thesis, Uppsala University, 2004.

[7] Bevot, J. *ANTLRWorks* [online]. 2007 [cit. 2007-06-06]. URL: ⟨http://antlr.org/works⟩.

[8] Clarke, E. M., Grumberg, O., a Peled, D. A. Model checking. 1999.

[9] Golson, S. State machine design techniques for verilog and vhdl. Tech. rep., Trilobyte Systems, Trilobyte Systems, 33 Sunset Road, Carlisle MA 01741, 2004.

[10] Gomez, R., a Bowman, H. Compositional detection of Zeno behaviour in Timed Automata. Tech. Rep. 12-06, Computing Laboratory, University of Kent, CT2 7NF Canterbury, Kent, UK, December 2006.

[11] Kristensen, J., Mejlholm, A., a Pedersen, S. Automatic translation from uppaal to c.

[12] Krákora, J., a Hanzálek, Z. Testing of hybrid real-time systems using fpga platform.

[13] Larsen, K. G., a Yi, W. *UPPAAL* [online]. 2007 [cit. 2007-06-06]. URL: ⟨http://www.uppaal.com/⟩.

[14] Parr, T. *ANTLR* [online]. 2007 [cit. 2007-06-06]. URL: ⟨http://antlr.org⟩.

[15] Tasiran, S., Alur, R., Kurshan, R. P., a Brayton, R. K. Verifying abstractions of timed systems. In *International Conference on Concurrency Theory* (1996), pp. 546–562.

Seznam obrázků

Obsah CD

BP-Jan_Breuer

Adresář `BP-Jan_Breuer` obsahuje tuto bakalářskou práci ve formátu PDF.

Uppaal2XSG

Adresář `Uppaal2XSG` obsahuje zdrojové kódy k aplikaci pro převod a samotný přeložený program

- `src` – zdrojové kódy k aplikaci napsané v Jave a projekt pro Eclipse
- `bin` – výsledá aplikace pro převod spolu s potřebnými knihovnami Xerces a Antlr3

Examples

Adresář `Examples` obsahuje několik příkladů převodu.