VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

KONSTRUKCE EFEKTIVNÍCH AUTOMATŮ PRO ROZPOZNÁVÁNÍ REGULÁRNÍCH VÝRAZŮ V HW

CONSTRUCTION OF EFFECTIVE AUTOMATA FOR REGEX MATCHING IN HW

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **JAKUB FREJLACH**
AUTHOR

VEDOUCÍ PRÁCE — **RNDr. MILAN ČEŠKA, Ph.D.**
SUPERVISOR

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2019/2020

Zadání bakalářské práce

Student: **Frejlach Jakub**
Program: Informační technologie
Název: **Konstrukce efektivních automatů pro rozpoznávání regulárních výrazů v HW**
Construction of Effective Automata for Regex Matching in HW
Kategorie: Teoretická informatika

Zadání:

1. Seznamte se s problematikou rozpoznávání regulárních výrazů v HW.
2. Seznamte se s technikami aproximace konečných automatů, které využívají charakteristiku vstupních dat.
3. Navrhněte a implementujte vlastní modifikace uvedených technik dovolující konstrukci aproximovaných automatů, které splňují dané omezení na jejich strukturu (počet stavů/přechodů či počet tříd znaků) a které mají co nejmenší chybu vzhledem k daným vstupním datům.
4. Proveďte podrobné experimentální ověření implementovaných technik a diskutujte dosažené výsledky a možnosti jejich dalšího rozvoje.

Literatura:

- M. Češka, V. Havlena, L. Holík, O. Lengál, and T. Vojnar. Approximate reduction of finite automata for high-speed network intrusion detection. International Journal on Software Tools for Technology Transfer, 2019.
- M. Češka, V. Havlena, L. Holík, J. Kořenek, et al. Deep packet inspection in FPGAs via approximate nondeterministic automata. In FCCM'19, p. 109-117, IEEE 2019.
- S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In SIGCOMM'06, p. 339-350, ACM 2006.

Pro udělení zápočtu za první semestr je požadováno:

- První dva body zadání a začátek prací na bodě třetím.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Češka Milan, RNDr., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 28. května 2020
Datum schválení: 7. dubna 2020

Abstrakt

Motivací této bakalářské práce je užití rozpoznávání regulárních výrazů v aplikačních doménách, kde je vyžadováno rychlé rozpoznávání jako například v hloubkové kontrole paketů. Během akcelerace jsou regulární výrazy ve formě nedeterministických konečných automatů syntetizovány na FPGA. Ačkoliv hardwarová akcelerace řeší rychlostní problémy, tak trpí zvýšenou spotřebou FPGA součástek, konkrétně LUT. Tato práce se zabývá návrhem, implementací a experimentálním vyhodnocením heuristické metody pro aproximaci konečných automatů pro rozpoznávání regulárních výrazů v hardware. Účelem této aproximace je snížení spotřeby LUT součástek při syntéze na FPGA. Princip redukční metody je založen na přidávání nových přechodů, čímž je zajištěna tvorba menšího počtu znakových tříd a je tak dosaženo zredukování spotřeby LUT při implementaci přechodů. Zavedená nepřesnost je minimalizována modifikací pouze méně významných částí automatu. Navržená metoda i s testovacím prostředím je implementována v nástroji TOFA. Technika byla vyhodnocena na syntetických i reálných datech. Výsledky experimentů ukázaly, že přechodová aproximace zvláště dobře funguje na automatech, kde se vyskytuje velký počet znakových tříd.

Abstract

This thesis is motivated by the application of REs in domains requiring fast matching such has deep packet inspections. To ensure sufficient speed a HW acceleration is typically employed. During the acceleration, REs are in the form of NFA synthesized on FPGA. Although HW acceleration solves the speed problems, it suffers from increased consumption of the FPGA components, specifically LUT. The goal of this thesis is to design, implement and experimentally evaluate heuristic method for approximation of FA in context of HW accelerated RE matching. The purpose of this approximation is to lower consumption of LUT components during FPGA synthesis. The key idea of the method is to add some transitions allowing to construct a smaller number of character classes and thus to reduce the number of LUT implementing the transition relation while reducing the error by modifying only less significant parts of FA. Proposed method together with evaluation pipeline is implemented in TOFA tool. Technique was evaluated on both synthetic and real data. Results of experiments shows, that transitional approximation works especially well on automatas with large number of equivalence character classes.

Klíčová slova

konečný automat, redukce a aproximace automatů, regulární výraz, hardwarová akcelerace

Keywords

finite automata, automata reduction and approximation, regular expression, hardware acceleration

Citace

FREJLACH, Jakub. *Konstrukce efektivních automatů pro rozpoznávání regulárních výrazů v HW*. Brno, 2020. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce RNDr. Milan Češka, Ph.D.

Konstrukce efektivních automatů pro rozpoznávání regulárních výrazů v HW

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana RNDr. Milana Češky, Ph.D. Další informace mi poskytli Ing. Jiří Matoušek, Ph.D. a Ing. Vlastimil Košař. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

. . . . . .
Jakub Frejlach
3. června 2020

Poděkování

Velice rád bych poděkoval svému vedoucímu Milanu Češkovi za jeho skvělé vedení práce, cenné rady, které mi byl kdykoliv ochoten poskytnout, a hlavně za jeho vždy pohotové a rychlé reakce při nutnosti konzultování nových věcí. Dále bych chtěl poděkovat Jiřímu Matouškovi a Vlastimilu Košařovi za jejich pomoc s proniknutím do hardware problematiky. Všem třem výše zmíněným bych chtěl nakonec poděkovat za příjemnou a přátelskou atmosféru, která vždy na osobních setkáních s nimi panovala.

Obsah

Kapitola 1

Úvod

Téma této práce je úzce spjato s dvojicí, v informatice, velmi mocných nástrojů. Prvním z této dvojice je konečný automat, výpočetní model, který se rozličnými způsoby dotýká širokého spektra informatiky, ať už jde o softwarové inženýrství, číslicové systémy nebo třeba síťové protokoly. Druhým z dvojice je regulární výraz, nepostradatelný pomocník při vyhledávání různých vzorů v textu. Oba mají společné jedno a to, že jsou možnou reprezentací regulárního jazyka.

Využít regulární výrazy je možné k řešení rozličné škály problémů. Některé problémy mají společnou vlastnost a to jest potřebu řešit je rychle a v reálném čase. Řeč je například o uplatnění regulárních výrazů při vyhledávání v databázích a nebo filtraci síťového provozu, kde jsou systémy pro odhalení průniku postaveny na metodě hloubkové inspekce paketů. Ta funguje na principu prohledávání paketů pomocí regulárních výrazů, které popisují nejrůznější podezřelou síťovou aktivitu a útoky. Mezi takové systémy patří například i moderní nástroj Snort [13] nyní vyvíjený společností Cisco.

Inspirace pro předkládanou práci pramení hlavně z výzkumu na němž se podílely výzkumné skupiny VeriFIT a ANF@FIT z Fakulty informačních technologií Vysokého učení technického v Brně. Právě filtrace síťového provozu byla motivací pro tento výzkum. Prvotní problém spočíval v tom, že softwarové provedení hloubkové inspekce paketů nedosahovalo dostatečných rychlostí. Vyřešení rychlosti bylo představeno skupinou ANT@FIT využitím hardwarové akcelerace [5]. V rámci tohoto procesu jsou regulární výrazy ve formě konečných automatů syntetizovány na programovatelná hradlová pole, nebo-li anglicky Field Programmable Gate Array (FPGA).

Ačkoliv akcelerace jeden problém vyřešila, objevil se díky ní problém druhý. Syntetizované automaty měly příliš velkou spotřebu FPGA zdrojů. Hlavním FPGA zdrojem je komponenta nazývající se vyhledávací tabulka, anglicky Lookup Table (LUT). S částečným řešením problému velké spotřeby zdrojů přišla skupina VeriFIT. Jejich výzkum přináší několik technik, jenž s využitím aproximace stavů redukují automat a tím pádem snižují spotřeby FPGA zdrojů [16].

Tato práce se bude dále zabývat návrhem dalších redukčních technik, které by měly umožnit ještě dodatečně snížit spotřebu zdrojů. Jak tyto nové, tak již existující techniky, o kterých zde bude řeč, jsou založené na principu aproximace. Aproximace se vyznačuje hlavně tím, že nezachovává přijímaný jazyk, ale za to naopak umožňuje dosáhnout daleko větších redukcí, než metody, které jazyk zachovávají. Jinými slovy při aplikaci takových technik není zachována původní přesnost automatů, ale výměnou za to je možno dosáhnout větších úspor. Na rozdíl od implementovaných redukcí jenž byly popsány v článku [16], kde hlavním fenoménem byly počty stavů, bude zde brán zřetel hlavně na přechody a abecedu

automatu. Výsledný efekt redukčních metod je nutné dále experimentálně vyhodnotit na příslušné sadě dat, v tomto případě se samozřejmě bude jednat o automaty různých rozměrů, aby bylo možné ověřit, že navržené techniky přinesou přijatelnou úsporu FPGA zdrojů výměnou za únosnou míru nepřesnosti.

K návrhu a implementaci redukčních technik byly velmi nápomocny redukce navržené při výzkumu, jenž jsou popsány ve vědeckém článku [16]. Poskytnuté byly ve formě sofistikovaného redukčního nástroje. V pochopení hardwarové stránky pak značně pomohla výzkumná skupina ANT@FIT a jejich nástroj schopný spočítat informace o spotřebě FPGA zdrojů.

Přínosem této práce je nově navržená heuristická redukční technika, založená na nadaproximaci přechodů automatu. Otestována byla na sadě synteticky vygenerovaných automatů a dále pak na automatech získaných z regulárních výrazů využívaných k filtraci síťového provozu. Bylo pomocí ní dosaženo zhruba 36% úspor na zdrojích za cenu 2% nepřesnosti na deterministických konečných automatech, dále ušetření cca 14 % zdrojů na nedeterministických konečných automatech za zavedení 7% chyby a nakonec asi 3% úspor na automatech pro filtraci síťového provozu při zanedbatelné chybě zdaleka necelého 1 %. Technika byla implementována jako jednoduchý redukční nástroj v programovacím jazyce Python 3. Ve stejném jazyce je napsán i testovací skript. Výsledky testování i redukční nástroj mohou být využity k redukování automatů, kde se vyskytuje exploze přechodů v menších i větších mírách, a dále k návrhu a vylepšení dalších metod z tohoto odvětví.

V práci jsou na začátku v kapitole 2 napřed jasně definovány základní pojmy důležité k pochopení problematiky. Dále je v kapitole 3 poskytnut náhled na již existující techniky ve srovnání s nově navrhovanými a ukázky dalších možných přístupů k řešení těchto problémů. Rovněž jsou zde detailněji vysvětleny spojitosti mezi redukcemi a hardwarovou částí a s ní pak dále související nástroj. Kapitola 4 o aproximaci přechodů pak definuje navrženou heuristiku a detailně popisuje celé její fungování. Heuristika a nástroje potřebné k jejímu otestování byly dále implementovány ve formě několika spolupracujících skriptů, více o nich prozradí kapitola 5. Navržené řešení bylo podrobeno několika fázím experimentování, o jehož průběhu a výsledcích pojednává kapitola 6. Celkové shrnutí práce a jejích výsledků, možnosti pro další rozšíření a nedostatky navrženého řešení lze nalézt v závěrečné kapitole 7.

Kapitola 2

Základní pojmy

Tato práce bude hojně využívat některých základních termínů z oblastí teoretické informatiky a hardwaru a proto budou tyto pojmy nejprve vysvětleny a jasně definovány. V případě, že je čtenář již dostatečně obeznámen se všemi pojmy z této kapitoly, je možné tuto kapitolu vynechat. Následující definice a popisy vztahující se k teoretické informatice jsou převážně převzaty z článku a studijních materiálů [3, 8, 9, 18]. Nutný podklad pro síťový provoz pochází z knihy [6] a základy o FPGA jsou z článků [10, 4].

Představena bude napřed dvojice pojmů jazyk a řetězec, která je naprostým základem pro další souvislosti. Dále budou definovány pojmy konečný automat a regulární výraz, jenž jsou oba možnou reprezentací regulárního jazyka. Budou ukázány možné způsoby využití regulárních výrazů a konkrétněji bude představeno užití regulárních výrazů při filtraci síťového provozu. Závěrem kapitoly pak bude čtenář uveden do problematiky využití hardwarové akcelerace při syntéze automatů na integrovaný obvod FPGA.

2.1 Řetězce a jazyky

Definice 2.1 *Abeceda je konečná, neprázdná množina, kterou značíme* Σ. *Prvky abecedy se nazývají symboly. Řetězcem nad abecedou* Σ *označujeme každou konečnou posloupnost symbolů z abecedy* Σ.

Jako příklad si lze uvést řetězec $x = ababa$ nad abecedou $\Sigma = \{a, b\}$. Pro každý řetězec w lze dále definovat jeho délku jako $|w| = n$. Délka řetězce x by tedy byla $|x| = 5$. Nad abecedou Σ existuje ještě speciální prázdný řetězec, který značíme ϵ. Jeho délka je nulová, tedy $|\varepsilon| = 0$. Množinu všech řetězců nad abecedou Σ označujeme jako Σ^*.

Definice 2.2 *Formálním jazykem nad abecedou* Σ *nazýváme každou množinu, pro kterou platí, že* $L \subseteq \Sigma^*$.

Příkladem jazyka nad abecedou $\Sigma = \{a, b\}$ nechť je jazyk $L_1 = \{w : |w| = 2\}$. Jazyk L_1 obsahuje řetězce aa, ab, ba, bb. V případě, že jazyk L neobsahuje žádné řetězce, jedná se o prázdný jazyk, což značíme jako $L = \emptyset$. S jazyky lze provádět běžné množinové operace, jakožto průnik, sjednocení, rozdíl, doplněk, atd. Pro jazyky jsou dále definovány ještě další operace. Jedná se o součin (konkatenaci) a iteraci.

Definice 2.3 *Nechť* L_1 *a* L_2 *jsou jazyky nad abecedou* Σ. *Potom součinem (konkatenací) jazyků* L_1 *a* L_2 *nad abecedou* Σ *rozumíme jazyk* $L_1 \cdot L_2 = \{w_1 w_2 : w_1 \in \Sigma \wedge w_2 \in \Sigma\}$.

Konkrétní příklad lze uvést s jazyky $L_1 = \{a, bc\}$ a $L_2 = \{d, ef\}$ nad abecedou $\Sigma = \{a, b, c, d, e, f\}$. Výsledkem operace součin je jazyk $L_1 \cdot L_2 = \{ad, aef, bcd, bcef\}$.

Definice 2.4 *Nechť L je jazyk. Potom iteraci jazyka L, L^*, a pozitivní iteraci jazyka L, L^+ lze definovat následně:*

1) $L^0 = \varepsilon$

2) $L^n = L \cdot L^{n-1}, n \geq 1$

3) $L^* = \bigcup_{n \geq 0} L^n$

4) $L^+ = \bigcup_{n \geq 1} L^n$

Opět si lze uvést příklad. Mějmě jazyk $L = \{a, ab\}$ nad abecedou $\Sigma = \{a, b\}$. Pak $L^* = \{\varepsilon, a, ab, aa, aab, aba, abab, ...\}$ a $L^+ = \{a, ab, aa, aab, aba, abab, ...\}$.

2.2 Konečné automaty

V předchozí sekci byl definován pojem jazyk. Konkrétně se dále zaměříme na skupinu regulárních jazyků k jejichž reprezentaci lze použít konečného automatu nebo regulárního výrazu. Napřed definujeme pojem konečný automat.

Definice 2.5 *Konečným automatem (KA) rozumíme uspořádanou pětici $M = (Q, \Sigma, \delta, q_0, F)$, kde:*

- *Q je konečná množina stavů,*
- *Σ je konečná vstupní abeceda,*
- *δ je funkce přechodů (přechodová funkce) ve tvaru $\delta : Q \times \Sigma \to 2^Q$,*
- *$q_0 \in Q$ je počáteční stav,*
- *$F \subseteq Q$ je množina koncových stavů.*

Dále je potřeba definovat funkci $\hat{\delta}$, která poslouží k vysvětlení souvislosti konečného automatu a řetězce.

Definice 2.6 *Funkce $\hat{\delta} : Q \times \Sigma^* \to 2^Q$ je definována rekurzivně z funkce δ jako:*

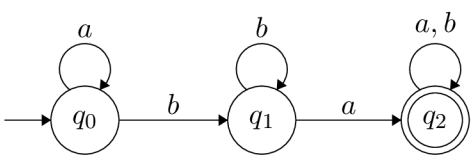
- $\hat{\delta}(q, \varepsilon) = q$,
- $\hat{\delta}(q, wa) = \delta(\hat{\delta}(q, w), a)$.

V podstatě tato funkce mapuje stav $q \in Q$ a řetězec w do nové množiny stavů $\hat{\delta}(q, w)$. O jazyku L pak lze prohlásit, že je přijímán (akceptován) konečným automatem $M = (Q, \Sigma, \delta, q_0, F)$, když pro každý řetězec $w \in L$ platí, že $\hat{\delta}(q_0, w) \subseteq F$. Skutečnost, že jazyk L je přijímán konečným automatem M, značíme $L(M)$.

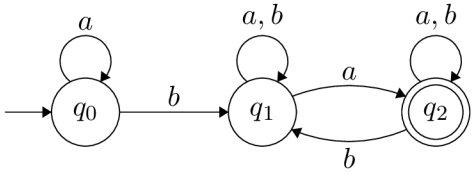
Existují dva typy konečných automatů a to deterministický konečný automat (DKA) a nedeterministický konečný automat (NKA). Aby by mohl být konečný automat označován jako deterministický (DKA), tak musí platit, že $\forall q \in Q, \forall a \in \Sigma |\delta(q, a)| \leq 1$. Analogicky

jestliže platí, že $\exists q \in Q, \exists a \in \Sigma |\delta(q,a)| > 1$, pak je konečný automat nazýván nedeterministický (NKA).

Každý NKA lze převést na DKA, tomuto procesu se říká determinizace. Pro každý regulární jazyk existuje minimální DKA, tedy automat s minimálním počtem stavů, který ho přijímá. Pro každý regulární jazyk existuje taktéž nějaký NKA, který ho přijímá. Ačkoliv výpočetní síla NKA a DKA je totožná, tak DKA může mít exponenciálně větší počet stavů než jemu odpovídající NKA. Rozdíl mezi NKA a DKA ilustruje obrázek 2.1.



(a) Deterministický konečný automat M



(b) Nedeterministický konečný automat M'

Obrázek 2.1: DKA a k němu odpovídající NKA, $L(M) = L(M')$.

2.3 Regulární výrazy

Regulární výrazy jsou druhou možnou reprezentací regulárního jazyka.

Definice 2.7 *Nechť Σ je abeceda. Potom regulární výrazy nad abecedou Σ a jazyky, které značí, jsou definovány následně:*

- *$\emptyset$ je regulární výraz značící prázdnou množinu,*
- *ε je regulární výraz značící jazyk $\{\varepsilon\}$,*
- *a, kde $a \in \Sigma$, je regulární výraz značící jazyk $\{a\}$,*
- *dále nechť r a s jsou regulární výrazy, které značí po řadě jazyky L_r a L_s, potom:*
  - *$(r \cdot s)$ je regulární výraz značící jazyk $L = L_r \cdot L_s$,*
  - *$(r + s)$ je regulární výraz značící jazyk $L = L_r \cup L_s$,*
  - *(r^*) je regulární výraz značící jazyk $L = L_r^*$*

Jako příklad si lze uvést regulární výraz $a^*bb^*a(a + b)^*$, jenž značí stejný regulární jazyk, jako oba automaty z obrázku 2.1. Velmi častou operací s regulárním výrazem je jeho transformace na konečný automat. Tato reprezentace je vhodnější k realizaci regulárního výrazu. Ve většině případů se regulární výraz transformuje na NKA. Volba nedeterminismu nad determinismem je ovlivněna již známým faktem, že DKA může být exponenciálně větší co do počtu stavů než jemu odpovídající NKA.

2.3.1 Využití regulárních výrazů

Regulární výrazy a jejich rozpoznávání nalézají v informatice velmi široké uplatnění, jelikož i velmi krátký regulární výraz může popsat velmi obsáhlý jazyk. Jejich schopnost vyhledávání různých textových řetězců lze využít například při:

- Validaci dat – hesla, e-mailové adresy, atd.

- Získávání dat z webu
- Analýze a transformaci dat
- Vyhledávání v databázích
- **Filtraci síťového provozu**

Požadavky na rozpoznávání se různí od případu užití. V některých případech je kladen důraz hlavně na rychlost a v jiných zase na co největší přesnost. S kritériem rychlosti se pak ještě může pojit nutnost rozpoznávání v reálném čase. Právě při realizaci filtrace síťového provozu se potkávají požadavky na rychlost a zpracování v reálném čase společně a vznikají zde problémy již naznačené v úvodu. Pro lepší pochopení bude tedy nutno napřed si lehce přiblížit filtraci síťového provozu.

Rozpoznávání regulárních výrazů ve filtraci síťového provozu
Filtrace síťového provozu stojí na bezpečnostních systémech. Tyto systémy jsou nasazovány pro monitorování nejrůznějších síťových objektů. Jednou z jejich hlavních součástí jsou systémy pro odhalení průniku, anglicky Intrusion Detection System (IDS). Tyto systémy sbírají data, skenují síťové porty, analyzují pakety, atd. V případě, že zaznamenají podezřelou aktivitu či nějaký síťový útok, tak zareagují. Reagovat mohou jak pouhým hlášením, tak aktivním zásahem [6]. Mezi systémy pro odhalení průniku patří například Snort [13] nebo Zeek (dříve Bro) [11].

IDS může být realizován několika způsoby, ovšem tím stěžejním pro tuto práci je realizace pomocí hloubkové inspekce paketů, anglicky Deep packet inspection (DPI). Ta je založena na tom, že podezřelá síťová aktivita a různé útoky jsou popsány pomocí regulárních výrazů, čímž je tak vytvořena sada pravidel. IDS pak přímo prohledává pakety a hledá shodu mezi jejich obsahem a nějakým pravidlem, které je popsáno regulárním výrazem. V případě, že je shoda nalezena, IDS může zareagovat.

Hloubková inspekce paketů bohužel trpí velkým omezením dosažitelné rychlosti. Pro dnešní vysokorychlostní sítě jsou softwarová provedení DPI velmi pomalá a není tedy možné zůstat u samotného softwarového provedení. Konkrétně Zeek může na serverech s vhodnou distribucí provozu dosáhnout až 100 Gbps propustnosti, což není ani zdaleka dost při existenci 400 Gbps sítí [16]. Pro odstranění problému rychlosti je třeba sáhnout k hardwarové akceleraci, díky které celý proces rozpoznávání dosáhne požadovaných rychlostí [5].

Ačkoliv pro uvedení do problematiky a celkovou motivaci posloužila filtrace síťového provozu, tak je nutné se od tohoto odvětví částečně oprostit. Jak již bylo řečeno, síťové odvětví totiž samozřejmě není ani zdaleka jediné, kde rychlost rozpoznávání hraje důležitou roli. A i když bude filtrace síťového provozu ještě několikrát zmíněna, tak je nutné další sekce a kapitoly brát z obecného hlediska, kdy rychlost a přesnost rozpoznávání se neomezuje pouze na síťovou problematiku.

2.4 Hardwarová akcelerace

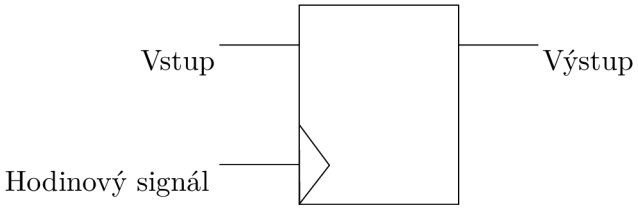
Nezbytnou součástí akcelerace jsou bezesporu vhodné komponenty, na kterých celá akcelerace poběží. Dnes se k tomuto účelu hojně využívá programovatelných hradlových polí, která jsou ovšem nejčastěji známá pod zkratkou FPGA z anglického Field Programmable Gate Array. FPGA je typ integrovaného obvodu mezi jehož výhody patří velká flexibilita, která spočívá v možnosti programování požadované aplikace přímo až cílovým zákazníkem.

Není tak tedy dopředu přesně určeno, k čemu FPGA bude využito, což s sebou přináší řadu výhod, jako například snadný a rychlý vývoj aplikací a možnost jednoduché rekonfigurace [4].

2.4.1 Syntéza na FPGA

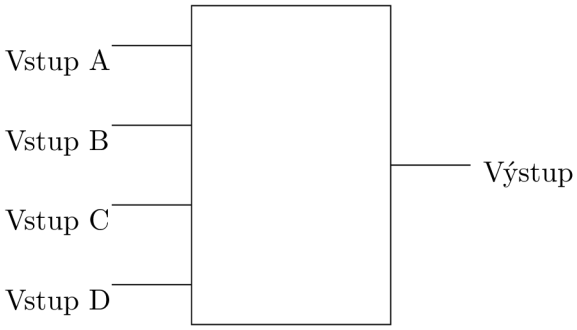
Pro syntézu na FPGA je nutné regulární výraz převést na jeho hardwarovou reprezentaci, protože sám o sobě není regulární výraz pro tuto oblast vhodný. Naopak reprezentace ve formě konečného automatu je pro hardware velice vyhovující. Konkrétně NKA, kvůli jeho výhodám, které spočívají v jednodušším mapování na FPGA a obecně vyšších dosažitelných frekvencí při zpracovávání, jak je popsáno v článku [5]. Každý takový automat samozřejmě po syntéze na FPGA zabere určité množství zdrojů. V FPGA jsou to převážně součástky dvojího typu.

Registr
Registr nebo anglicky též Flip–Flop (FF) je binární registr, který slouží pro zachování logického stavu. Při každém hodinovém cyklu FPGA produkuje registr signál s hodnotou 0 nebo 1. Hodnota, kterou uchovává, může být rovněž modifikována [10]. Při syntéze konečného automatu na FPGA jsou registry zodpovědné za uchování informace o aktuálním stavu automatu.



Obrázek 2.2: Schéma registru (převzato z[10]).

Vyhledávací tabulka
Vyhledávací tabulka, pro kterou se ovšem ve světě používá zkratky LUT z anglického názvu Lookup Table, je součástka, jenž je schopná realizace libovolné funkce s N proměnnými vstupními hodnotami a s 1–bitovým výstupem. Počet vstupních proměnných se může lišit podle technologie, ovšem nejčastější jsou 4-vstupé a 6-vstupé LUT. Pro všechny kombinace vstupních hodnot je v LUT již předdefinována pravdivostní tabulka s příslušnými výstupy [4, 10]. LUT při syntéze konečného automatu realizují přechody a abecedu automatu.



Obrázek 2.3: Schéma 4-vstupé LUT (převzato z [4]).

2.4.2 Akcelerační architektury

Pro hardwarovou akceleraci existuje mnoho různě výkonných architektur. Zvláště efektivní architektury pro rozpoznávání regulárních výrazů představuje článek [5], jedná se především o architektury s paralelními automaty a se zřetězenými automaty. Pro kontext této bakalářské práce není tak důležitý princip těchto architektur jako spíše jejich výsledný efekt na rychlost rozpoznávání regulárních výrazů. Obě zmíněné architektury měly velmi kladný dopad na celkovou rychlost rozpoznávání. Toto zvýšení rychlosti je ovšem draze vykoupeno prudkým nárůstem spotřeby FPGA zdrojů, kde pro různé automaty v závislosti na jejich rozsáhlosti vystoupala spotřeba na desetitisíce a v některých případech dokonce až na statisíce LUT. Odstranit problém zdrojů se částečně podařilo rozličným aproximačním redukčním technikám [16].

Kapitola 3

Aproximace automatu pro rozpoznávání regulárních výrazů v hardware

V této bude kapitole důkladněji představena aproximace automatů, jenž se jeví jako jedno z možných řešení exploze spotřeby FPGA zdrojů. Důkladněji budou ukázány aproximační techniky z výzkumu, jenž byl popsán v článku a práci [16, 14]. Jako kontrast s aproximačními technikami budou zmíněny i jiné redukční techniky. Závěrem této kapitoly bude pak poskytnut náhled na celkový dopad zmíněných redukčních zlepšení a na základě nových znalostí budou navrženy další možné způsoby, jak dosavadní řešení ještě více vylepšit.

3.1 Redukce automatu

Redukce automatů je možné rozdělit do dvou základních skupin. Ty, které jazyk zachovávají, a ty které jazyk nezachovávají. Z hlediska přesnosti by se zprvu mohlo zdát, že nejvhodnější budou právě ty, které jazyk zachovávají, jelikož pro regulární výraz je důležitá přesnost rozpoznávaného jazyka. Například v již zmíněné filtraci síťového provozu by mohl být nějaký závažný útok přehlédnut v případě, že jazyk nebude zachován. Mezi způsoby, které jazyk zachovávají, lze zařadit třeba obyčejnou minimalizaci NKA, která, ačkoliv je sama o sobě velmi náročným problémem, může být alespoň částečně provedena. Jedním z existujících nástrojů, které jsou takových minimalizací schopny, je například Reduce [7]. Bohužel metody zachovávající jazyk jsou značně omezeny a jimi poskytnuté redukce nejsou velmi atraktivní.

Pro dosažení větších redukcí jsou zapotřebí techniky, jenž jazyk nezachovávají. Obecně se jim říká aproximační. Aproximační techniky mají obecně tu vlastnost, že výměnou za větší redukce, než jaké poskytují jazyk zachovávající techniky, automat znepřesňují. Tento fakt sám o sobě nemusí být nevýhodou, pokud je automat znepřesněn pouze na určitou míru a při tom je dosaženo dostatečných úspor. Aproximaci lze dále rozdělit na podaproximaci a nadaproximaci. Blíže budou oba dva pojmy specifikovány v kapitole 4, tudíž zde jen pro vysvětlení nezbytně nutného základu pro další souvislosti. Oba přístupy mohou přinést značné redukce, nicméně s podaproximací se pojí jistá rizika. Příkladem si lze uvést již dobře známou filtraci síťového provozu. Ačkoliv původní jazyk nezachová ani jeden typ aproximace, tak u podaproximace hrozí, že nějaký typ útoku může IDS odignorovat, což

je v praxi velmi nebezpečné. Naproti tomu nadaproximace pouze zvýší počet případů, kdy bude hlášen planý poplach.

Mezi aproximační techniky patří například přidávání vlastních smyček přes každý symbol nad vybrané stavy automatu (nadaproximace) nebo odebírání stavů automatu (podaproximace), obě techniky zmiňuje článek [17]. Dále se jedná o metody prořezávání a spojování stavů (nadaproximace), které představuje výzkum popsaný v článku [16] a práci [14]. Z těchto posledních dvou metod má práce vychází a tedy je vhodné je blíže popsat.

3.2 Aproximace stavů automatu

Informace pro tuto sekci jsou převážně převzaty z článku [16] a práce [14], kde je představeno částečné řešení neúnosné spotřeby FPGA zdrojů, jenž byla nastíněna v sekci 2.4. Jedná se o redukční techniky automatů, které jsou postavené na nadaproximaci stavů automatu. Konkrétně se nazývají prořezávání a spojování stavů. Obě techniky na sebe částečně navazují.

3.2.1 Aproximace založená na datech

Základním principem dvou výše zmíněných technik je odstraňování stavů a následná úprava přechodů, počátečního stavu a stavů koncových. Způsob odstranění a dalších úprav se v obou technikách výrazně liší, ovšem to, co mají obě společné, jsou řídící metriky. Metriky a tím pádem i celá aproximace jsou v tomto případě založeny na konkrétních datech, které automaty, potažmo regulární výrazy běžně zpracovávají. Jedná se o textové řetězce. Tyto data pomohou určit méně relevantní části automatu, jenž jsou vhodné pro modifikaci. Tento přístup se tak liší od jiných přístupů, které mohou být založené třeba čistě na vlastnostech automatu nebo jiných skutečnostech.

V ideálním případě by metrika existovala pouze jedna, která by dokázala přesně určit, jakým způsobem automat aproximovat a zároveň by znázorňovala chybu takto provedenou aproximací. Jinak řečeno to znamená, že metrika ukáže například na určitý stav, který vyhodnotí jako aktuálně nejvhodnější pro odstranění a zároveň řekne, že automat se touto aproximací znepřesní o dané procento.

Naneštěstí taková metrika zatím neexistuje a proto je nutné zapojit do aproximace metrik vícero. Můžeme je rozdělit na metriky řídící a metriky vyhodnocovací. Řídící metrika plní první ze dvou funkcí výše popsané „univerzální“ metriky, tedy určuje průběh aproximace, což znamená, že pomáhá identifikovat, jaké úpravy na automatu provést, aby byla výsledná chyba co nejmenší. Bohužel o tom, jak velká chyba nastane, už není schopná prozradit nic. K tomu je zapotřebí metrik vyhodnocovacích, jenž na základě původního automatu, aproximovaného automatu a dalších potřebných dat určí co nejpřesněji, jak velká je zavedená chyba.

Mezi metriky řídící lze zařadit frekvenci stavu, důležitou metriku pro prořezávání a spojování stavů, jenž bude popsána v další sekci. Další řídící metriky pak budou zavedeny pro nové redukční techniky v kapitole 4. Vyhodnocovací metriky budou detailněji vysvětleny v kapitole 6.

3.2.2 Frekvence stavu

Frekvence stavu[1] je tedy velmi důležitá metrika pro obě stavové aproximační techniky, která jednoduše řečeno určuje, které stavy jsou více a méně relevantní v rámci automatu jakožto celku. Stavy s menší relevancí jsou pak velmi vhodné pro aplikování redukcí, obecně modifikací.

Každý stav musí být před začátkem samotné aproximace označen hodnotou frekvence k němu příslušné. K výpočtu frekvence je zapotřebí původní automat určený k redukci a dále tzv. trénovací data. Obecně můžeme trénovacími daty nazvat množinu řetězců. Tyto trénovací řetězce jsou jeden po druhém zpracovány automatem určeným pro redukci. Při zpracovávání jednoho konkrétního řetězce je vždy navýšena o 1 hodnota frekvence všech takových stavů, kterých je při zpracovávání řetězce dosaženo. Pokud existuje více cest, jak s jedním řetězcem dosáhnout určitého stavu, tak se tyto cesty nerozlišují a frekvence je zvýšena taktéž pouze o 1. Jedinou výjimku tvoří počáteční stav automatu, pro něj se v každém případě hodnota frekvence zvýší o 1, jelikož počátečního stavu je v automatu dosaženo vždy. V případě, že je potom počátečního stavu dosaženo ještě nějakou další cestou, je hodnota frekvence zvýšena ještě o 1. Tedy pro počáteční stav se za každý trénovací řetězec může hodnota frekvence zvýšit o 1 až 2 a pro všechny ostatní stavy se hodnota frekvence může zvýšit o 0 až 1 za každý trénovací řetězec.

Výpočet frekvence byl napřed obecně představen na pouhých řetězcích, protože tento obecný popis bude využit dále v práci. Pro určení frekvence v článku [16] bylo ovšem použito nikoliv obecných trénovacích řetězců, ale konkrétně, jelikož hlavním cílem výzkumu bylo užití regulárních výrazů ve filtraci síťového provozu, trénovacího síťového provozu. Trénovací provoz není nic jiného, než množina paketů. Místo obyčejných řetězců je pak frekvence stavů určena přímo z obsahu těchto paketů.

3.2.3 Stavové aproximační techniky

Nyní, když je hlavní řídící metrika popsána, lze přiblížit obě aproximační techniky.

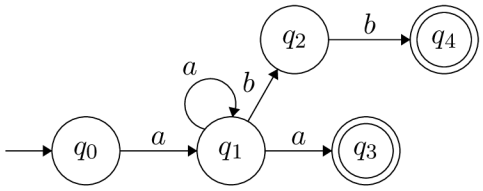
Prořezávání stavů

Prořezávání stavů z automatu odstraňuje množinu stavů R, které jsou na základě hodnoty frekvence pro automat méně důležité. Rovněž jsou z automatu odstraněny všechny přechody původně jdoucí z nebo do stavů z množiny R. Samotné odstranění by ovšem nevyhovovalo nadaproximaci, tudíž je nutné dále všechny stavy, které nepatří do množiny R a mají přechod jdoucí do stavu z množiny R, změnit na stavy koncové. Rozsah redukce pak musí být přesně určen mírou redukce $\theta \in (0, 1]$ a nový automat by tak měl mít $m = \lceil \theta \cdot |Q| \rceil$ stavů, kde Q je množina všech stavů automatu. Množina R potom obsahuje $|Q| - m$ stavů s nejmenší hodnotou frekvence, tedy stavy nejméně významné. Příklad prořezávání stavů si lze prohlédnout na obrázku 3.1.

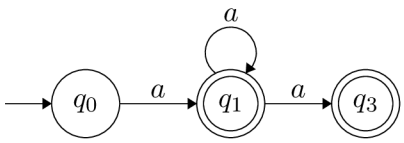
Spojování stavů

Spojováním stavů se rozumí spojování stavů s určitou vzdáleností. Vzdálenost stavů je další metrika, která je určena z frekvence stavů. Pro jakékoliv dva sousední stavy q a r je vzdálenost $d(q, r) = max(\frac{f_q}{f_r}, \frac{f_r}{f_q})$, kde f_q a f_r jsou frekvence stavů q a r. Pro jakékoliv dva nesousední stavy se pak vzdálenost určí jako $d(q, r) = \infty$. Dále ve spojování stavů figuruje

[1] Frekvence stavu je v článku [16] označována jako signifikance stavu, ale pro tuto bakalářskou práci je pojem signifikance stavu definován v podsekci 4.3.1. Pozor na případnou záměnu pojmů.



(a) Původní automat

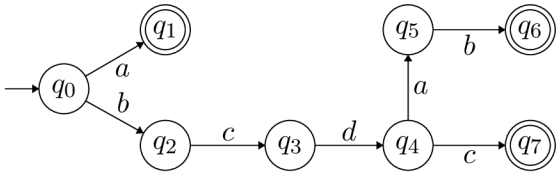


(b) Automat získaný prořezáváním stavů

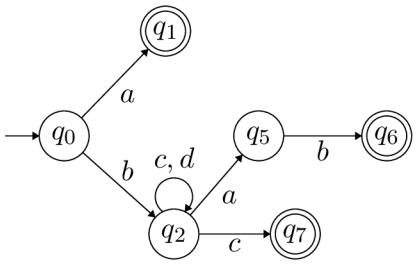
Obrázek 3.1: Prořezávání stavů (převzato z [16]).

hranice spojování D, což znamená, že jsou spojeny pouze stavy, kde $d(q, r) < D$. Pro větší přesnost je zavedena ještě maximální frekvence $F_{max} \in (0, 1]$, která zabraňuje, aby byly spojeny stavy, jejichž poměr frekvence k celkovému počtu trénovacích paketů, potažmo řetězců S, je větší než zmíněna maximální frekvence. Tedy stav q nelze spojit s žádným jiným stavem, pokud platí, že $\frac{f_q}{|S|} > F_{max}$.

Pokud jsou dva stavy vyhodnoceny jako vyhovující pro spojení, pak spojování probíhá následně. Mějme konečný automat $M = (Q, \Sigma, \delta, q_0, F)$, ve kterém chceme spojit stavy q a r, potom napřed r bude vyměněno za q na levé straně každé přechodové funkce $\delta(r, a) \to q_{'}$. Dále všechny přechody jdoucí do stavu r budou přesměrovány do stavu q. Pro zachování přesnosti musí být q učiněno stavem počátečním a/nebo stavem koncovým, pokud stav r byl stavem počátečním a/nebo stavem koncovým. Na závěr je nutné stav r vyjmout z množiny stavů Q a z množiny koncových stavů F. Příklad spojování ilustruje obrázek 3.2.



(a) Původní automat



(b) Automat získaný spojováním stavů

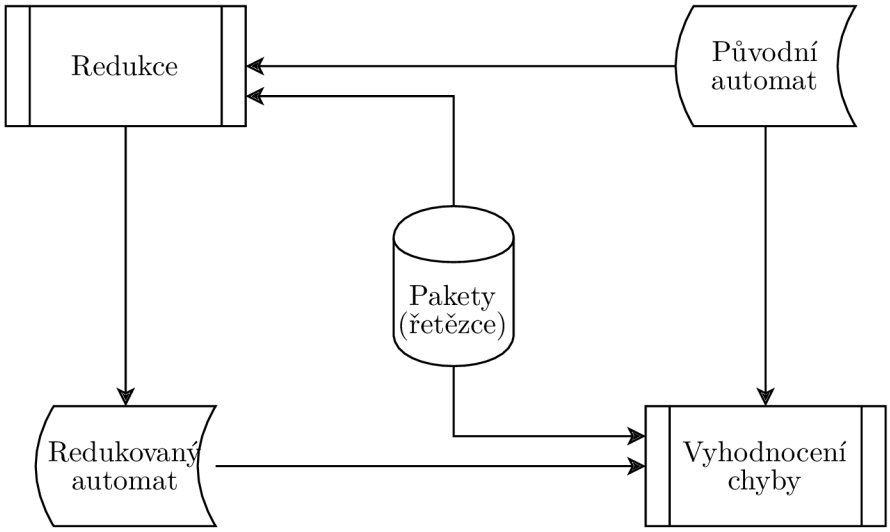
Obrázek 3.2: Spojování stavů (převzato z [16]).

3.2.4 Aproximační nástroj AHOFA

Vše související se stavovými aproximacemi je zapouzdřeno v aproximačním nástroji AHOFA (Approximate Handling of Finite Automata) [15]. AHOFA je napsán v programovacích jazycích C++ a Python 3 a skládá se ze tří hlavních částí. Všechny části spolu komunikují podle zjednodušeného diagramu na obrázku 3.3.

1) **Redukce automatů (prořezávání a spojování stavů)** – tato část provádí redukce prořezávání a spojování, vstupem je míra redukce a automat určený k redukci v textovém formátu `.fa` (více o něm v 3.2.4), výstupem je redukovaný automat ve stejném formátu.

2) **Výpočet frekvence stavů** – tato část není uživatelsky přístupná, nicméně ze vstupních trénovacích paketů, které jí jsou předány ve formě binárních souborů PCAP, a ze vstupního automatu určí frekvenci pro každý stav automatu.



Obrázek 3.3: Zjednodušené schéma nástroje AHOFA (převzato z [14]).

3) **Vyhodnocení chyby redukovaného automatu** – v této části je vypočtena chyba redukovaného automatu na základě testovacích paketů/řetězců (více o nich prozradí kapitola 6). Vstupem je redukovaný automat, původní automat a seznam testovacích paketů opět ve formě binárních souborů PCAP. Výstupem je několik hodnot, týkajících se přesnosti redukovaného automatu, blíže budou specifikovány rovněž v kapitole 6.

Textová reprezentace automatu – fa
Celý nástroj AHOFA pracuje s automaty v textovém formátu s koncovkou `fa`, který byl vytvořen speciálně pro tento nástroj. Jeho zápis je velmi intuitivní.

```
<stav> // počáteční stav
<stav> <stav> <symbol> // přechody
...
<stav> // koncové stavy
...
```

Všechny značky `<stav>` jsou nezáporná celá čísla. Pro všechny značky `<symbol>` pak platí, že jsou ve formátu `0xhh`, kde `hh` je číslo v šestnáctkové soustavě od `00` až do `ff`. Formát `fa` tedy pokrývá všechny symboly rozšířené ASCII tabulky.

3.3 Vliv aproximace automatů na hardware zdroje

Nyní je třeba vyhodnotit celkové úspory na FPGA zdrojích a na základě úspor navrhnout nové redukční techniky. Vyhodnocení lze provést na sadě regulárních výrazů z reálného světa. Princip mapování konečného automatu na FPGA je značně komplikovaný a je nad rámec této práce se jím do hloubky zabývat a proto se k vyhodnocení zdrojů nabízí dvou variant, které pro mě budou řešit problematiku mapování a výpočtu spotřeb. Je to možné buď užitím syntézy přímo na FPGA, kdy lze dosáhnout přesných výsledků spotřeby zdrojů za cenu delšího trvání, a nebo pomocí odhadů, které hodnoty spotřeby dokáží určit pouze přibližně ovšem za zlomek času reálné syntézy.

3.3.1 Nástroj Netbench

Pro odhady zdrojů posloužil nástroj NETBENCH od výzkumné skupiny ANT@FIT [12]. Jedná se o relativně komplexní aplikační rámec, jenž poskytuje rozsáhlou funkcionalitu pro zpracovávání paketů, rozpoznávání regulárních výrazů, generování kódu programovacího jazyka VHDL pro syntézu na FPGA a mnoho dalšího. Napsán je v programovacím jazyce Python 2.

Stěžejní část, která byla z nástroje NETBENCH využita, poskytuje odhady spotřebovaných zdrojů FPGA, tedy LUT a registrů, pro automat při FPGA syntéze. Výpočet odhadů pracuje na základě algoritmů popsaných v [1, 2]. Statistiky jsou přehledně vypsány napřed celkově, tedy celková spotřeba LUT a registrů, a navíc ještě jejich distribuce mezi tři hlavní části při FPGA syntéze automatu. Jedná se o:

1) **Dekodér symbolů** – zpracování abecedy automatu

2) **Logika automatu** – přechody a stavy automatu

3) **Relizace koncových stavů automatu**

Výsledkem odhadu je tedy osm hodnot, čtyři týkající se spotřeby LUT a čtyři týkající se spotřeby registrů. Odhadový modul podporuje regulární výrazy ve formátu PCRE (Perl Compatible Regular Expressions) a nebo již přímo konečné automaty v textovém formátu `msfm`.

Textová reprezentace automatu – msfm
Oproti formátu `fa` je `msfm` o něco složitější, avšak strojově je lépe čitelný. Dále je v něm možno navíc specifikovat i víceznakové symboly, tzv. znakové třídy, jenž budou hrát velmi důležitou roli v návrhu dalších redukčních technik, více o nich bude sděleno později.

```
<počet stavů automatu>
<počet přechodů automatu>
<stav>|<číslo symbolu>|<stav>|<epsilon> // přechody
...
### // libovolně dlouhá řádka tvořená znaky '#'
<počet koncových stavů>
<stav>, ... // koncové stavy oddělené čárkou
### // libovolně dlouhá řádka tvořená znaky '#'
<počet symbolů abecedy>
<číslo symbolu>:<znak>|... // symboly
...
```

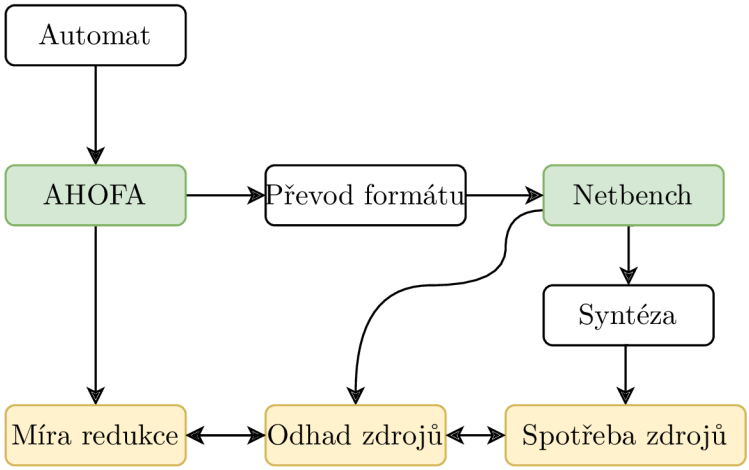
Všechny značky `<stav>` a `<číslo symbolu>` jsou nezáporná celá čísla. Značka `<epsilon>` nabývá hodnoty 1 v případě ϵ přechodu, jinak nabývá hodnoty 0. Pro značku `<znak>` pak platí stejná pravidla jako pro značku `<symbol>` ve formátu `fa`. Formát `msfm` tedy taktéž pokrývá celou rozšířenou ASCII tabulku.

3.3.2 Spotřeba FPGA zdrojů

Nejprve bylo nutné zjistit, jak moc se odhad liší od skutečné spotřeby FPGA zdrojů a jestli je mezi těmito hodnotami jistá korelace. Samotná syntéza je totiž časově relativně náročný proces, který pro rychlou evaluaci výsledků není vhodný. Bylo provedeno několik testů, ke

kterým byly využity automaty získané ze sady pravidel `backdoor` ze systému Snort [13]. Tyto automaty byly zredukovány technikou prořezávání stavů na různé velikosti.

Následující proces je znázorněn na obrázku 3.4. Automat byl nejdříve zredukován pomocí nástroje Ahofa na několik měr. Dále byl pro originální automat i jeho redukované varianty odhadnut počet FPGA zdrojů. Na závěr byla ještě provedena skutečná syntéza, z níž byly rovněž zjištěny hodnoty o zdrojích. Výsledky si lze prohlédnout v tabulkách 3.1 a 3.2. Tabulky vždy ukazují přímo počet LUT/registrů a k tomu ještě procentuální podíl těchto součástek oproti originálnímu automatu. Z výsledků je patrné, že ačkoliv se hodnoty odhadnuté nástrojem Netbench a skutečné hodnoty syntézy liší, tak poměrově k mírám redukce klesají velmi podobně. Na základě toho je tady možné pro další postup rozdíl mezi hodnotami zdrojů ze skutečné syntézy a z odhadu zanedbat a nadále používat z důvodu časových úspor pouze odhad.



Obrázek 3.4: Schéma porovnávání odhadů a reálné spotřeby FPGA zdrojů.

Míra redukce	Netbench [LUT]	Syntéza [LUT]	Netbench [Míra LUT]	Syntéza [Míra LUT]
1.0	4226	3468	1,00	1,00
0.8	3609	2788	0,85	0,80
0.6	3027	2079	0,72	0,60
0.4	2303	1379	0,54	0,40
0.2	1566	796	0,37	0,23

Tabulka 3.1: Odhad a syntéza LUT pro původní automat získaný ze sady pravidel `backdoor` a jeho redukované varianty pomocí prořezávání stavů.

Jak již bylo řečeno, Netbench umí krom samotného odhadu celkových FPGA zdrojů odhadnout i celkovou distribuci těchto zdrojů mezi 3 základní části FPGA syntézy automatů. Jedná se o dekodér symbolů, logiku automatu a koncové stavy. Pro další postup tedy bylo nutné zjistit, jak přesně se zdroje rozdělují mezi jednotlivé části a na kterých částech by se daly další zdroje případně ušetřit. V tabulkách 3.3 a 3.4 si lze tuto distribuci prohlédnout. Jako modelový automat byl opět zvolen ten získaný ze sady pravidel `backdoor`.

Na výsledcích lze pozorovat, že spotřeba zdrojů pro realizaci koncových stavů může být zanedbána, protože jak její spotřeby LUT, tak její spotřeby registrů jsou pro všechny

Míra redukce	NETBENCH [FF]	Syntéza [FF]	NETBENCH [Míra FF]	Syntéza [Míra FF]
1.0	3889	3479	1,00	1,00
0.8	3109	2699	0,80	0,78
0.6	2330	1922	0,60	0,60
0.4	1550	1173	0,40	0,37
0.2	775	498	0,20	0,14

Tabulka 3.2: Odhad a syntéza registrů pro původní automat získaný ze sady pravidel `backdoor` a jeho redukované varianty pomocí prořezávání stavů.

varianty automatu velmi nízké a navíc konstantní. Spotřeby registrů je pak možné zanedbat kompletně, jelikož pro dekodér symbolů jsou nulové a na logice automatu jsou prakticky přímo úměrné počtu stavů automatu. Pro další zkoumání tak tedy zbývá pouze spotřeba LUT na dekodéru a logice automatu.

Je možné si všimnout zajímavého fenoménu, kdy s klesajícím počtem LUT pro automatovou logiku naopak stoupá počet LUT na dekodéru. Tento úkaz není pravidlem a nenastává u všech automatů. U některých může být spotřeba na dekodéru víceméně neměnná, někde se může i zvyšovat a někde může kmitat mezi nízkými a vysokými hodnotami. Právě z tohoto důvodu byl automat ze sady pravidel `backdoor` vybrán jako modelový příklad, jelikož právě na něm lze pozorovat tento narůst na dekodéru symbolů. Tento náhlý narůst je zapříčiněn tzv. tvorbou ekvivalenčních znakových tříd. Tyto znakové třídy budou jedním ze základních kamenů pro návrh nových redukčních technik a proto je třeba si je detailněji představit.

Míra redukce	Celkem [LUT]	Dekodér [LUT]	Logika automatu [LUT]	Koncové stavy [LUT]
1.0	4226	273	3917	36
0.8	3609	436	3137	36
0.6	3027	632	2359	36
0.4	2303	688	1579	36
0.2	1566	760	770	36

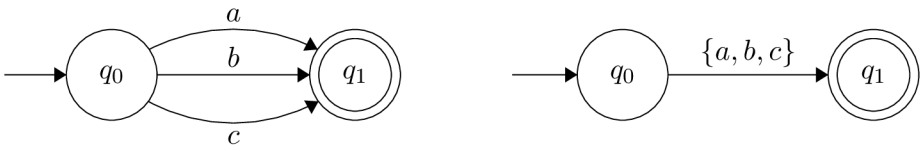
Tabulka 3.3: Odhad distribuce LUT mezi jednotlivé části FPGA pro automat získaný ze sady pravidel `backdoor` a jeho redukované varianty pomocí prořezávání stavů.

Míra redukce	Celkem [FF]	Dekodér [FF]	Logika automatu [FF]	Koncové stavy [FF]
1.0	3889	0	3888	1
0.8	3109	0	3108	1
0.6	2330	0	2329	1
0.4	1550	0	1549	1
0.2	775	0	774	1

Tabulka 3.4: Odhad distribuce registrů mezi jednotlivé části FPGA pro automat získaný ze sady pravidel `backdoor` a jeho redukované varianty pomocí prořezávání stavů.

3.3.3 Ekvivalenční znaková třída

Při syntéze automatu na FPGA je nutné provést různé optimalizace, které také přispívají k úbytku na spotřebovaných zdrojích. Mezi takové optimalizace patří právě vytváření ekvivalenčních znakových tříd. Základní koncept spočívá ve zjednodušení přechodů automatu. Pro představu mějme konečný automat $M = (Q, \Sigma, \delta, q_0, F)$, kde mezi stavy q_0 a q_1 existují přechody přes symboly a, b, c. Jedná se o tři přechody a realizace každého takového přechodu není zadarmo, nýbrž vyžaduje LUT, které jsou spotřebovány pro logiku automatu. Tyto tři přechody ovšem syntéza nahradí pouze jedním přechodem přes nový symbol, který v sobě bude obsahovat všechny symboly přechodů, které takto byly nahrazeny. V tomto případě tedy budou přechody přes symboly a, b, c nahrazeny přechodem přes symbol $\{a, b, c\}$. Takto vytvořený symbol se nazývá ekvivalenční znaková třída viz obrázek 3.5. Jde v podstatě o seskupení symbolů nad přechody mezi dvojicemi stavů do speciálních množin. Důležité je zmínit, že není podmínkou, aby byla znaková třída vytvořena ze symbolů ze všech přechodů mezi určitou dvojicí stavů. Právě naopak, znakové třídy mohou být seskupovány všelijak.



Obrázek 3.5: Automat s třemi přechody přes symboly a, b, c (vlevo) a automat s jedním přechodem přes znakovou třídu $\{a, b, c\}$ (vpravo).

V tomto případě tak na FPGA nebude nutné syntetizovat přechody tři, ale pouze jeden. Samozřejmě taková úspora zdrojů musí být nějakým způsobem vyvážena. Jak už tomu napověděly tabulky z předchozí sekce, tak tvorba znakových tříd může zvýšit spotřebu LUT na dekodéru symbolů. Způsobeno je to faktem, že po vytvoření znakových tříd může abeceda automatu obsahovat více symbolů, než obsahovala předtím. Zároveň také realizování symbolu znakové třídy stojí více LUT, než by stálo realizování pouhého jednoznakového symbolu.

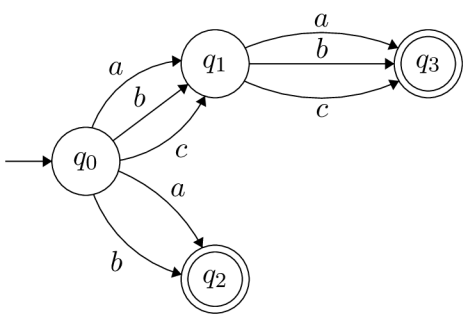
Další velkou výhodou znakových tříd je jejich možné sdílení mezi více dvojicemi stavů. Jelikož znaková třída v automatu při syntéze figuruje jako jeden symbol, tak je samozřejmě možné přes takový symbol vést vícero přechodů. Jako příklad si lze uvést automat na obrázku 3.6. Na něm došlo v důsledku vytvoření znakových tříd k úspoře pěti přechodů a zredukování počtu symbolů na pouhé dva symboly z původních třech.

Formálně si tedy lze definovat ekvivalenční znakovou třídu S jako neprázdnou množinu obsahující symboly nad přechody mezi jednou nebo vícero dvojicemi stavů automatu. Jako dvojici stavů lze uvažovat i dvojici stejných stavů v případě vlastní smyčky.

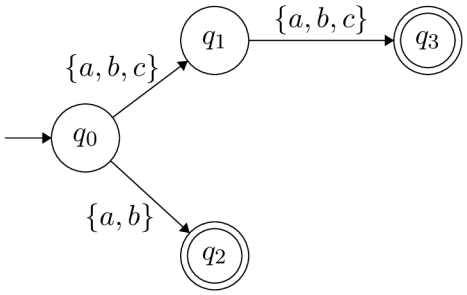
3.3.4 Tvorba ekvivalenčních znakových tříd

Tvorba ekvivalenčních znakových tříd přináší značné úspory na LUT. Pro představu mohou posloužit statistiky již dobře známého automatu z pravidel `backdoor`. Dle tabulky 3.5 je jasně vidět, že úspory získané při vytvoření znakových tříd na logice automatu značně převažují případné zvýšení spotřeby LUT na dekodéru.

Je tedy jasné, že proces tvorby znakových tříd je pro syntézu velmi důležitý. Tvorba znakových třída ovšem není v nástroji NETBENCH implementována optimálně. NETBENCH všechny znakové třídy vytváří pomocí hladového algoritmu. Tento algoritmus je velmi naivní a funguje tak, že mezi každou dvojicí stavů je vytvořena znaková třída ze symbolů nad



(a) Automat M bez znakových tříd



(b) Automat M s vytvořenými znakovými třídami

Obrázek 3.6: Ukázka tvorby ekvivalenčních znakových tříd.

Míra redukce	Znakové třídy [LUT]	Bez znakových tříd [LUT]
1.0	4226	28055
0.8	3609	26382
0.6	3027	24438
0.4	2303	22634
0.2	1566	8363

Tabulka 3.5: Rozdíl odhadu LUT pro automat z pravidel `backdoor` s tvorbou znakových tříd a bez tvorby znakových tříd.

všemi přechody mezi touto dvojicí. Tvořeny jsou tak v podstatě ty největší možné třídy mezi každou dvojicí stavů, ačkoliv se nemusí jednat vždy o optimální řešení. Jak bylo tedy výše zmíněno, že ačkoliv lze vytvářet znakové třídy i z menšího počtu symbolů, než který se právě nachází mezi dvěma stavy, NETBENCH vždy zvolí tu největší možnou znakovou třídu.

V tomto přístupu vidím možný problém, pokud by se v automatu nacházela velká variabilita přechodů. V takovém případě totiž nebude dobře využito sdílení tříd. Pro ilustraci si lze představit konečný automat s abecedou $\Sigma = \{\text{a-z}\}$, který má mezi každou dvojicí stavů sadu přechodů přes unikátní kombinaci znaků. Potom, když jsou utvořeny mezi každou možnou dvojicí stavů znakové třídy, jejich počet může snadno dosáhnout vysokých hodnot a tím pádem značně zatížit spotřebu znakového dekodéru, který je bude muset realizovat. Délka původní abecedy byla 26 symbolů, ovšem po vytvoření všech možných největších znakových tříd mezi všemi možnými dvojicemi stavů, kde každá dvojice má unikátní kombinaci přechodů, může délka abecedy jít klidně do tisíců, desetitisíců, atd. v závislosti na velikosti automatu.

Tento přístup by tedy mohl být značně nevyhovující a dávalo by potom smysl tvořit znakové třídy nějakým více sofistikovanějším způsobem, kdy se budou tvořit i menší třídy, které mají ovšem velké zastoupení a nebude tak tedy doslova nutné tvořit jednu znakovou třídu pouze pro jeden její unikátní výskyt. Nalezení nejoptimálnějšího řešení je bohužel výpočetně velmi náročně. Nabízí se tedy možné řešení ve formě aproximace, kdy by byly přechody v automatu aproximovány tak, aby se počet znakových tříd co nejvíce zredukoval. Konkrétně se v další kapitole chci zaměřit na nadaproximaci přechodů.

Kapitola 4

Aproximace přechodů automatu

Předchozí kapitola nastínila možný problém exploze přechodů mezi jednotlivými stavy automatu. Problémem by zde ovšem nebyl počet přechodů, nýbrž jejich velká variabilita mezi různými stavy. Úkolem je tedy navrhnout heuristiku, která by tento problém variability do jisté míry zredukovala skrze aproximaci. K jejímu realizování bude nejprve nutno si definovat aproximační operace, užitím kterých lze nižší variability dosáhnout. Dále jsou potřeba jisté metriky, jenž budou celý proces aproximace řídit a v neposlední řadě samotný algoritmus zodpovědný za průběh aproximace.

4.1 Aproximační operace

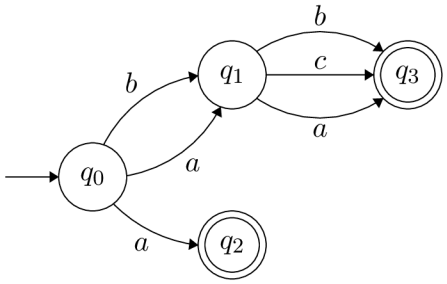
V rámci aproximace přechodů se nabízí operace „spojování znakových tříd“[1]. Základní princip této operace tkví, jak již z jejího samého názvu vyplývá, ve spojení dvou znakových tříd v jednu. Tohoto spojení může být dosaženo jak nadaproximací přechodů (přidáváním), tak jejich podaproximací (odstraňováním). Před samotným zavedením operace je tedy nutné obě metody porovnat a vybrat tu vhodnější pro další postup.

4.1.1 Podaproximace vs. nadaproximace přechodů

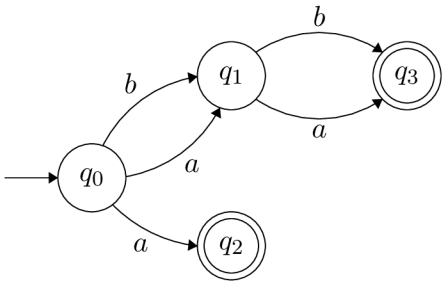
Z hlediska přesnosti automatu a jazyka, jenž bude po redukci přijímán, hrozí u podaproximace ztráta důležitých informací. Bude-li totiž z automatu odebrán byť jen jediný přechod, automat již nemusí přijmout některé řetězce původního jazyka. Nechť M a $M^{'}$ jsou konečné automaty, kde $M^{'}$ byl získán přechodovou podaproximací M. Dále mějme jazyky L a $L^{'}$, kde $L(M)$ a $L^{'}(M^{'})$. Potom platí, že $L^{'} \subseteq L$. Pro názornost si lze prohlédnout podaproximaci na obrázku 4.2a, kde je ilustrován podaproximovaný automat získaný redukcí automatu z obrázku 4.1.

Naproti tomu u nadaproximace dochází k jevu přesně opačnému. Tedy že při přidávání přechodů sice původní jazyk rovněž není zachován, nicméně jazyk nově přijímaný obsahuje všechny řetězce jazyka původního. Samozřejmě kromě toho může obsahovat i řetězce nové. Analogicky toto tvrzení tedy můžeme formálně zapsat následně. Nechť M a $M^{''}$ jsou konečné automaty, kde $M^{''}$ byl získán přechodovou nadaproximací M. Dále mějme jazyky L a $L^{''}$, kde $L(M)$ a $L^{''}(M^{''})$. Potom platí, že $L \subseteq L^{''}$. Pro názornost je nadaproximace opět ilustrována obrázkem 4.2b.

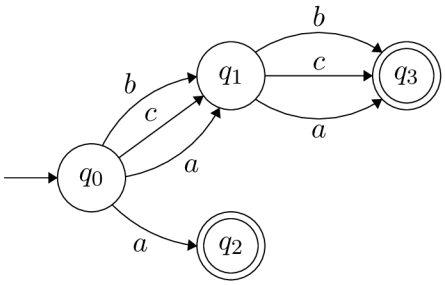
[1] Pozor, spojování znakových tříd nelze zaměňovat s pojmem redukční techniky „spojování“ zmíněné v 3 převzaté z [16, 14]



Obrázek 4.1: Automat M, $L(M)$ = {a, aa, ab, ac, ba, bb, bc}.



(a) Podaproximovaný automat M', $L'(M')$ = {a, aa, ab, ba, bb}.



(b) Nadaproximovaný automat M'', $L''(M'')$ = {a, aa, ab, ac, ba, bb, bc, ca, cb, cc}.

Obrázek 4.2: Redukce počtu znakových tříd aproximací přechodů automatu.

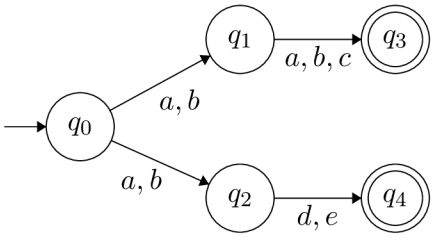
Po porovnání obou typů aproximací nelze s naprostou jistotou prohlásit, že jeden nebo druhý typ by byl celkově výhodnější. Záleží hlavně na oblasti využití aproximace, proto v některých případech může být vhodnější sáhnout po podaproximaci a v jiných zase po nadaproximaci. K učinění rozhodnutí mezi výše zmíněnou dvojicí je třeba vzít v potaz fakt, že stěžejní požadavek na redukované automaty, potažmo regulární výrazy, jenž automaty reprezentují, je aby všechny řetězce z jazyka akceptovaného automatem původním obsahoval i jazyk akceptovaný automatem redukovaným. Tedy vhodnější je pro aktuální situaci nadaproximace.

4.1.2 Spojování znakových tříd

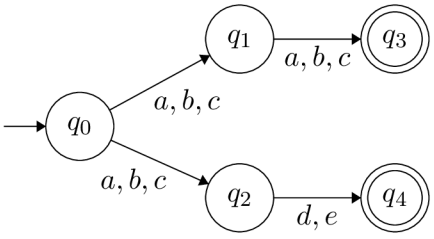
Na základě zvolené metody aproximace, tedy nadaproximace, může být zavedena operace spojování znakových tříd. Vstupem operace je konečný automat $M = (Q, \Sigma, \delta, q_0, F)$ a znakové třídy S_1 a S_2, s nimiž má být spojování provedeno. Výsledkem spojování je znaková třída $S_f = S_1 \cup S_2$. Podle výsledné znakové třídy bude vstupní konečný automat M nadaproximován následujícím způsobem. Pro každou dvojici stavů $q_{i1}, q_{j1} \in Q$, mezi nimiž se nacházela původní znaková třída S_1, bude přidán přechod automatu ze stavu q_{i1} do stavu q_{j1} přes symbol c_1 pro každé $c_1 \in S_f \setminus S_1$. Analogicky pro každou dvojici stavů $q_{i2}, q_{j2} \in Q$, mezi nimiž se nacházela původní znaková třída S_2, přibude nový přechod automatu ze stavu q_{i2} do stavu q_{j2} přes symbol c_2 pro každé $c_2 \in S_f \setminus S_2$. Obrázky 4.3 a 4.4 názorně ukazují možný vstup a výstup této operace.

Algoritmus 1: Spojování znakových tříd

SpojZT $((M = (Q, \Sigma, \delta, q_0, F), S_1, S_2)$
  Vstup: KA $(M = (Q, \Sigma, \delta, q_0, F)$, znakové třídy S_1 a S_2
  Výstup: Znaková třída S_f vytvořená spojením vstupních tříd S_1 a S_2
  $S_f := S_1 \cup S_2$;
  foreach q_{i1}, q_{j1} *mezi kterými je* S_1*, kde* $q_{i1}, q_{j1} \in Q$ **do**
    foreach $c_1 \in S_f \setminus S_1$ **do** $\delta(q_{i1}, c_1) := \delta(q_{i1}, c_1) \cup \{q_{j1}\}$;
  foreach q_{i2}, q_{j2} *mezi kterými je* S_2*, kde* $q_{i2}, q_{j2} \in Q$ **do**
    foreach $c_2 \in S_f \setminus S_2$ **do** $\delta(q_{i2}, c_2) := \delta(q_{i2}, c_2) \cup \{q_{j2}\}$;
  return S_f

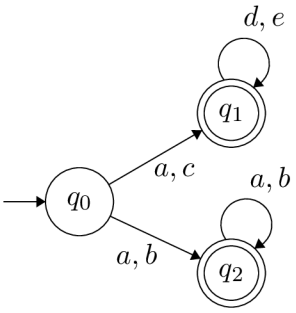


(a) Automat M se znakovými třídami {a,b}, {d,e} a {a,b,c}.

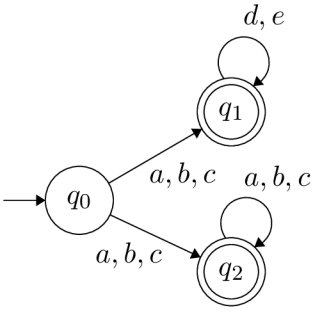


(b) Automat M' se znakovými třídami {d,e} a {a,b,c}.

Obrázek 4.3: Spojení dvou znakových tříd. Výsledná třída je jednou ze vstupních tříd.



(a) Automat M se znakovými třídami {a,b}, {a,c} a {d,e}.



(b) Automat M' se znakovými třídami {d,e} a {a,b,c}.

Obrázek 4.4: Spojení dvou znakových tříd. Výsledná třída je třídou novou.

4.2 Heuristika

Nadaproximace přechodů ve formě spojování znakových tříd bude základním kamenem heuristiky, která bude v dalších sekcích definována. Nabízí se otázka, proč se spojování znakových tříd vztahuje pouze na největší možné znakové třídy mezi každou dvojicí stavů a proč spojování menších znakových tříd není vůbec bráno v potaz. Odpověď se skrývá v názvu této sekce, definována bude totiž pouze heuristika a nikoliv přesné řešení problému exploze znakových tříd. Nalezení nejoptimálnější tvorby a následné nadaproximace by vyžadovalo vyzkoušení všech možných znakových tříd, které lze z přechodů mezi každou dvojicí stavů utvořit.

Pro představu mějme konečný automat $M = (Q, \Sigma, \delta, q_0, F)$, dále mějme stavy $q_1, q_2 \in Q$. Předpokládejme že mezi stavem q_1 a q_2 existují přechody přes symboly a, b, c. NETBENCH by v tomto případě ihned utvořil znakovou třídu {a,b,c}. Nicméně z této trojice symbolů lze utvořit ještě celou řadu dalších tříd. Výčtem se jedná o třídy {a}, {b}, {c}, {a,b}, {a,c} a {b,c}. Celkem tedy lze z těchto tří symbolů utvořit sedm různých znakových tříd. Naproti tomu už při pouhém zvýšení počtu symbolů ze tří na třicet je možné utvořit celkem 1 073 741 823 znakových tříd. Dochází k exponenciálnímu růstu kombinací, které není možno v rozumném čase prozkoumat a porovnat. Místo přesného řešení bude tedy navržena heuristika, jenž bude nadaproximovat pouze ty největší možné znakové třídy mezi dvojicemi stavů, a která bude řízena vhodnými metrikami. Z tohoto důvodu bude chápání pojmu „všechny utvořitelné znakové třídy“ zúženo pouze na ty největší možné mezi dvojicemi stavů.

4.3 Metriky pro aproximaci

Pro sestrojení heuristické techniky, která bude aproximovat přechody automatu je nutná metrika, která pomůže určit, jak velký dopad bude mít přidávání přechodů na celkové zachování přesnosti automatu. Předchozí kapitola zmiňuje frekvenci stavu jakožto hlavní metriku pro stavové redukční techniky prořezávání a spojování. V případě prořezávání nízká frekvence určuje stavy, které mají být prořezány, a v případě spojování jsou pak určeny stavy s podobnou frekvencí, které lze spojit v jeden [16]. Tato metrika je bohužel nevyhovující pro algoritmus pracující na bázi přechodové aproximace a to z toho důvodu, že je z ní velmi nesnadné určit hranici pro terminaci algoritmu. Základní problém tkví ve skutečnosti, že frekvence je úměrná počtu trénovacích řetězců, to jest pokud počet trénovacích dat vzroste stokrát, vroste podobnou měrou i frekvence stavů. Frekvence stavu tedy může nabývat různých hodnot v závislosti na počtu trénovacích řetězců. Na rozdíl od technik prořezávání a spojování, kde terminaci algoritmu spolehlivě určí počet stavů, na který se má automat zredukovat, v nové technice je tomu jinak. Zde se počet stavů nijak nemění, mění se pouze počet přechodů. Nabízí se tedy možnost zavést metriku novou. Jedná se o tzv. „míru aproximace“, která se nebude vztahovat ani k stavu a ani k přechodu automatu, nýbrž k jednotlivým znakovým třídám, které mohou být v rámci automatu vytvořeny. K výpočtu míry aproximace budeme dále potřebovat ještě signifikanci stavu.

4.3.1 Signifikance stavu

K tomu, aby bylo možné přesně určit míru aproximace pro všechny znakové třídy, je přesto nutné znát to, jak je každý stav v rámci celého automatu významný. Posloužit by k tomu mohla výše zmíněná frekvence. Bohužel jak už bylo řečeno, hodnota jednotlivých frekvencí stavů je velmi proměnlivá, potřebujeme tedy všechny frekvence sjednotit a omezit jejich hodnoty pouze na určitý interval. Zavádíme tedy novou metriku a to jest signifikanci stavu, kterou určíme pomocí frekvence stavu a celkového počtu trénovacích řetězců.

Mějme konečný automat $M = (Q, \Sigma, \delta, q_0, F)$, potom pro každý stav $q_i \in Q$ lze jeho signifikanci s_i spočítat jako:

$$s_i = \frac{f_i}{p}, i \in \mathbb{N}_0$$

Kde f_i je frekvence stavu a p je celkový počet trénovacích řetězců. Výsledná signifikance se pro každý stav $q_i \in Q \setminus \{q_0\}$ nachází v intervalu $[0, 1]$, protože každý trénovací řetězec frekvenci běžného stavu zvýší o 0 nebo 1. Signifikance počátečního stavu q_0 pak analogicky

spadá do intervalu $[1, 2]$, neboť každý trénovací řetězec frekvenci počátečního stavu zvýší o 1 nebo 2.

4.3.2 Míra aproximace znakové třídy

Míra aproximace znakové třídy udává, jak velkou aproximaci do celého automatu přineslo spojování právě této třídy, tedy jednodušeji řečeno kombinuje signifikanci stavů s rozsahem modifikace přechodů. Počáteční hodnota míry aproximace je pro každou znakovou třídu rovna 0. V momentě, kdy je provedeno spojování znakových tříd, míra aproximace výsledné třídy spojování musí být nahrazena novou hodnotou. Tedy nově určená hodnota náleží buď již existující znakové třídě, nebo nově vytvořené, v závislosti na vlastnostech tříd, které vstupují do procesu spojování.

Nechť $M = (Q, \Sigma, \delta, q_0, F)$ je konečný automat. Dále mějme znakové třídy S_1 a S_2 a produkt jejich spojování třídu S. Potom míra aproximace znakové třídy m pro třídu S je určena jako součet čtyř hodnot. První dvě hodnoty jsou míry aproximace tříd S_1 a S_2. Třetí hodnota je získána jako signifikance s_{i1} každého stavu q_{i1} z každé dvojice q_{i1}, q_{j1}, kde $q_{i1}, q_{j1} \in Q$, mezi kterými se nachází znaková třída S_1, vynásobená počtem znaků v $S \setminus S_1$, tedy počtem znaků, které do třídy S_1 musí být v rámci spojování tříd přidány. Analogicky čtvrtá hodnota je vypočtena jako signifikance s_{i2} každého stavu q_{i2} z každé dvojice q_{i2}, q_{j2}, kde $q_{i2}, q_{j2} \in Q$, mezi kterými se nachází znaková třída S_2, vynásobená počtem znaků v $S \setminus S_2$.

Algoritmus 2: Výpočet míry aproximace znakové třídy

```
1 Míra_Apx (S_1, S_2, aprox_trid, sig)
      Vstup: znakové třídy S_1 a S_2, mapování všech znakových tříd v automatu na
             jejich míru aproximace aprox_trid : S → ℝ_≥0, mapování všech stavů
             automatu na jejich signifikanci sig : Q → [0, 2]
      Výstup: Hodnota míry aproximace m
2     m := aprox_trid[S_1] + aprox_trid[S_2];
3     S := S_1 ∪ S_2;
4     foreach q_i, q_j ∈ Q, mezi kterými se nachází třída S_1 do
5         m := m + sig[q_i] * delka(S \ S_1)
6     foreach q_i, q_j ∈ Q, mezi kterými se nachází třída S_2 do
7         m := m + sig[q_i] * delka(S \ S_2)
8     return m
```

4.4 Algoritmus nadaproximující přechody automatu

Nyní, když jsou definovány všechny potřebné prerekvizity, může být navržena heuristika ve formě algoritmu, který bude užitím spojování znakových tříd nadaproximovat přechody automatu tak, aby bylo dosaženo zredukování počtu znakových tříd a tedy úspory LUT na dekodéru. Symbolický zápis je uveden v algoritmu 3.

Nejprve se vytvoří seznam T všech možných znakových tříd, které lze v daném automatu vytvořit. Postup je stejný jako při tvorbě znakových tříd, kterou aplikuje nástroj NETBENCH. Jedná se tedy o hladový algoritmus, jenž vždy nabídne tu největší možnou znakovou třídu, kterou lze vytvořit ze všech přechodů mezi dvojicí stavů. U každé znakové

třídy je navíc potřeba uchovat informaci právě o tom, mezi jakými stavy může být utvořena. Hodnota míry aproximace každé znakové třídy je inicializována nulovou hodnotou. Dále jsou z těchto tříd utvořeny všechny možné dvouprvkové kombinace. Tyto kombinace budou uloženy v seznamu K. Pro každý takto utvořený pár $S_i, S_j \in K$ je poté vypočítána pomocí algoritmu 2 míra aproximace viz řádek 8. Ze všech dvojic v K je následně vybrána ta s hodnotou míry aproximace nejmenší. Nyní hraje důležitou roli hranice míry aproximace h, která je jedním ze vstupů tohoto algoritmu. Pokud míra aproximace vybrané dvojice překročí hranici h, je algoritmus ukončen, tato případná termince algoritmu probíhá na řádku 11. Pokud hranice překročena není, je tato vybraná dvojice znakových tříd spojena algoritmem 1, viz řádek 13, a hodnota míry aproximace musí být pro výslednou třídu nahrazena nově spočítanou hodnotou. Dalším krokem je aktualizace všech dvojic znakových tříd v tomto pořadí:

1) Odstranění zaniklých znakových tříd ze seznamu T.

2) Odstranění dvojic znakových tříd z K, kde alespoň jedním členem dvojice byla třída, jenž v důsledku spojování zanikla. Toto může platit pro jednu i obě dvě vstupní třídy spojování.

3) V případě, že výsledná třída spojování je třídou novou, musí být přidána do seznamu tříd T a musí být pro ni vytvořeny a přidány do K všechny příslušné kombinace s ostatními třídami.

Od tohoto místa se již algoritmus opakuje. Opět je pro každý pár $S_i, S_j \in K$ určena hodnota míry aproximace a z těchto hodnot vybráno minimum. Jak již bylo naznačeno, algoritmus pak terminuje právě tehdy, když vybraná minimální hodnota míry aproximace je větší, než hranice h. Produktem algoritmu je přechodově nadaproximovaný automat M.

Algoritmus 3: Nadaproximace přechodů automatu

```
1  Nadaproximace (M, sig, h)
       Vstup: KA M = (Q, Σ, δ, q_0, F), mapování všech stavů automatu na jejich
              signifikanci sig : Q → [0, 2], hranice míry aproximace znakových tříd h
       Výstup: Nadaproximovaný automat M
2      T := Tridy(M);
3      for S ∈ T do
4          aprox_trid[S] := 0;
5      while True do
6          K := Kombinace(T, 2);
7          for S_i, S_j ∈ K do
8              miry[(S_i, S_j)] = Mira_Apx(S_i, S_j, aprox_trid, sig);
9          m, S_1, S_2 := MinDvojice(miry, K);
10         if m > h then
11             break
12         else
13             S := Spoj_ZT(M, S_1, S_2);
14             aprox_trid[S] := m;
15             T := T ∪ {S};
16             if S \ S_1 ≠ ∅ then T := T \ {S_1} ;
17             if S \ S_2 ≠ ∅ then T := T \ {S_2} ;
18     return M
19 MinDvojice (miry, K)
       Vstup: mapování miry : T^2 → ℝ_≥0, seznam všech dvouprvkových kombinací
              znakových tříd K
       Výstup: Nejmenší hodnota míry aproximace m z miry a k ní příslušná dvojice
               znakových tříd S_1 a S_2
20     m := miry.klice()[0];
21     S_1, S_2 := miry.hodnoty()[0];
22     for S_i, S_j ∈ K do
23         if miry[(S_i, S_j)] = m then
24             m := miry[(S_i, S_j)];
25             S_1 = S_i;
26             S_2 = S_j;
27     return m, S_1, S_2
```

Kapitola 5

Implementace

Navrženou aproximační techniku z předchozí kapitoly bylo nutné programově realizovat a následně otestovat. Kompletní programové zázemí této bakalářské práce s skládá z několika komplexnějších nástrojů a menších skriptů, které na sebe navazují a umožňují tak celkovou funkcionalitu. Konkrétně se jedná o dva již zmíněné nástroje třetích stran, Netbench [12] a Ahofa [15], dále generátor testovacích konečných automatů a k nim příslušných testovacích dat, nástroj Tofa (Transition Over-approximation of Finite Automata) obstarávající nadaproximační techniku přechodů, a testovací prostředí ve formě skriptu s názvem Stots (Simple Transition Over-approximation Testing Script).

5.1 Nástroje třetích stran

Ačkoliv byly již nástroje Netbench a Ahofa představeny a popsány v kapitole 3, tak je nutné si zasadit do souvislostí, jak přesně byly použity. Rovněž byly na nástrojích provedeny jisté modifikace pro zajištění funkčnosti s ostatními skripty, o kterých je potřeba se taktéž zmínit. Jelikož oba nástroje nepřijímají automaty ve stejném formátu, bylo nutné ještě implementovat převodník formátů. Napsán byl v jazyce Python 3 a je schopen obousměrného převodu konečného automatu mezi formáty `fa` a `msfm`.

5.1.1 Využití nástroje Netbench

Z nástroje Netbench byla využita část pro odhady spotřeb LUT a registrů pro počáteční vyhodnocování automatů ze síťového provozu z kapitoly 3. Dále byl tento nástroj s menší úpravou částečně využit i na experimenty s přechodovými nadaproximacemi. Nástroji Netbench byly předávány automaty v textovém formátu `msfm`, výstupem pak byl onen zmíněný odhad FPGA zdrojů. Změněn byl pouze výpis programu, jelikož již nebylo nutné detailně zobrazovat statistiky a postačil pouze celkový odhad LUT. Počet registrů totiž pro dekodér symbolů není důležitý.

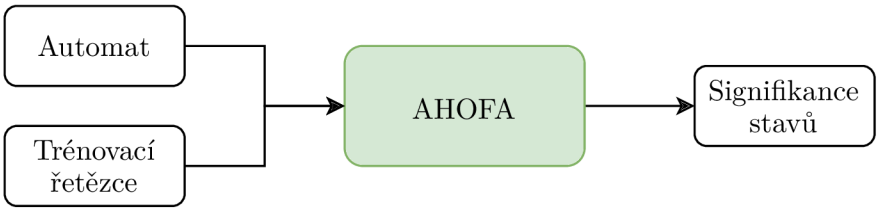
Obrázek 5.1: Schéma využité funkce nástroje Netbench.

5.1.2 Využití nástroje AHOFA

Tento nástroj měl hned několik případů užití. V prvé řadě jej bylo zapotřebí pro aplikace aproximačních technik spojování a prořezávání.

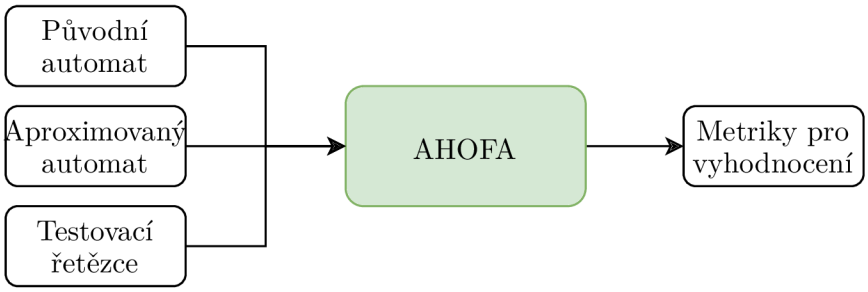
Obrázek 5.2: Schéma realizace stavových aproximačních technik nástrojem AHOFA.

Dále AHOFA posloužil pro získání frekvence stavů a z ní následné signifikance stavů, jenž je nezbytná k fungování nadaproximace přechodů. K tomu bylo nutné provést zásadní úpravu části nástroje, která je schopná z trénovacích dat vypočítat frekvenci stavů automatu. Bohužel jako trénovací data akceptuje pouze pakety ve formě binární souborů PCAP, které jsem neměl pro své testovací automaty, které byly převážně synteticky vygenerovány, k dispozici. Z tohoto důvodu byly očekávané vstupní testovací data změněny z PCAP souborů na obyčejné textové soubory s řetězci. Jak o syntetických automatech, tak o generovaných datech, bude více řečeno v podsekci 5.2.1. Po spočítaní frekvence stavů na základě testovacích řetězců a příslušného automatu je z každé frekvence vypočítaná signifikance stavu pomocí vzorce ze sekce 4.3.1. Výsledná signifikance je pak z nástroje AHOFA vyexportována ve formě jednoduchého textové souboru, kdy na každém řádku je číslo stavu a k němu příslušná hodnota.



Obrázek 5.3: Schéma výpočtu signifikance stavů pro automat nástrojem AHOFA.

Poslední využitou částí je potom vyhodnocení chyby redukovaných automatů. Chyba je určena hlavně z testovacích dat, pro které stejně jako v předchozím případě platí, že byly očekávány ve formě PCAP souborů s pakety, a proto byla provedena stejná modifikace vstupu na pouhé textové řetězce. Vstupem je tedy nyní redukovaný automat, původní automat ve formátu `fa` a jeden nebo více souborů s testovacími řetězci. Testovací řetězce jsou předány originálnímu i aproximovanému automatu a na základě toho, jak oba dva automaty tyto řetězce přijaly nebo zamítly, jsou dále vypočítány vyhodnocovací metriky. Tyto metriky jsou dále diskutovány v kapitole 6.



Obrázek 5.4: Schéma výpočtu metrik pro vyhodnocení chyby automatu nástrojem AHOFA.

5.2 Sada vlastních nástrojů

Krom nástrojů třetích stran jsem napsal několik vlastních nástrojů, které využívám pro získávání, vyhodnocování a manipulaci s daty. Všechny níže uvedené skripty a nástroje jsou implementovány v programovacím jazyce Python 3. Tento jazyk jsem zvolil hlavně kvůli jeho flexibilitě a snadné práci s ním.

5.2.1 Generátor konečných automatů

Hlavní funkcí tohoto skriptu je generování syntetických konečných automatů. Tyto automaty dále slouží jako testovací automaty pro aproximační nástroj TOFA. Skript je spuštěn pomocí programu `generate.py`, mezi jehož parametry patří mimo jiné počet stavů generovaného konečného automatu a cesta k souboru se vstupní abecedou. Abeceda je skriptu předána ve formě textového souboru s koncovkou `alph`. Struktura souboru `alph` vypadá následně.

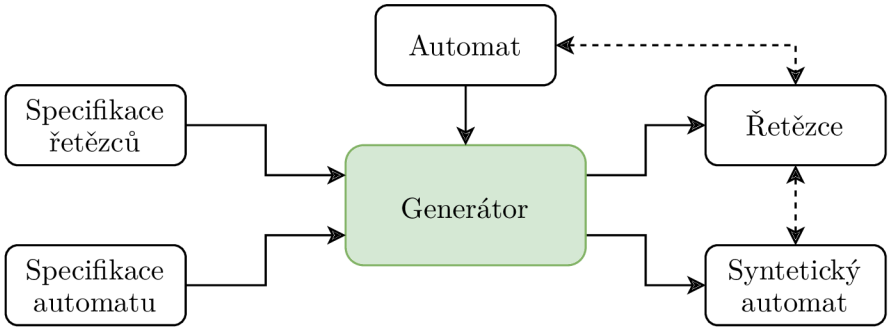
```
<číslo symbolu>:<symbol>|
...
```

Značka `<číslo symbolu>` je nezáporné celé číslo. Značka `<symbol>` pak opět symbolizuje to samé, co v popisu textových reprezentací `fa` a `msfm` v podsekcích 3.2.4 a 3.3.1.

Strategie generování automatu je postavena tak, aby byl v automatu přítomen velký počet znakových tříd a tím pádem se projevil problém spotřeby FPGA zdrojů na znakovém dekoréru, kterým se navržená aproximační technika zabývá. Nejprve jsou vygenerovány stavy a je určen stav počáteční, od kterého začíná celkové větvení automatu. Aktuálně zpracovávanému stavu je vždy přidělen určitý počet následníků. Mezi tímto stavem a následníky jsou vygenerovány přechody. Sada přechodů mezi dvěma stavy je vždy zvolena tak, aby se pokud možno v automatu neopakovala a vedla tak na unikátní znakovou třídu. Nad každým stavem jsou rovněž vytvářeny vlastní smyčky pro získání ještě většího počtu znakových tříd. Pokud již stavu nemůže být přidělen žádný následník, stává se z něj stav koncový. Automat v základu generuje NKA, ovšem pomocí volitelného parametru programu `generate.py` lze generovat i DKA.

Generátor má ještě druhou funkci, kterou je generování testovacích a trénovacích dat. Programu `generate.py` může být specifikována další trojice parametrů, která udává kolik souborů s řetězci se má vygenerovat, kolik má mít každý vygenerovaný soubor řetězců a zvlášť pro každý soubor, jaký má být poměr mezi řetězci automatem přijímanými a nepřijímanými. Data lze generovat buď rovnou společně při generování konečného automatu nebo lze generování automatu vynechat a vytvořit pouze data pro již existující automat.

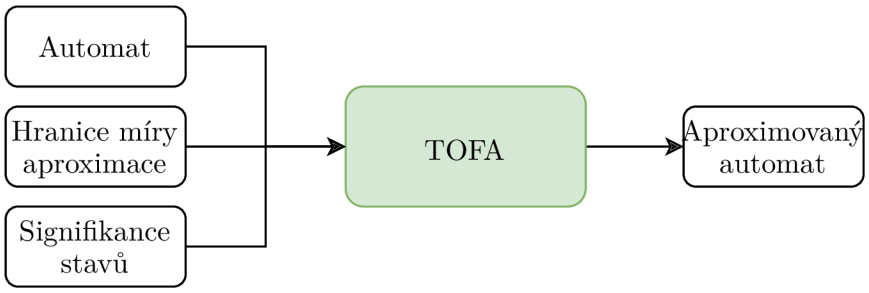
V takovém případě je automat načten generátorem z formátu `msfm`. Výstupem je tedy požadovaný počet souborů s testovacími řetězci, kde na prvním řádku každého souboru je počet řetězců a na každém dalším řádku jsou již samostatné řetězce. Strategie pro generování dat se v průběhu experimentování s automaty relativně měnila a proto bude důkladněji rozebrána až v kapitole 6. Generování automatů a řetězců je implementováno v třídě `AutomataGenerator`.



Obrázek 5.5: Schéma generátoru syntetických konečných automatů a testovacích řetězců.

5.2.2 Aproximační nástroj TOFA

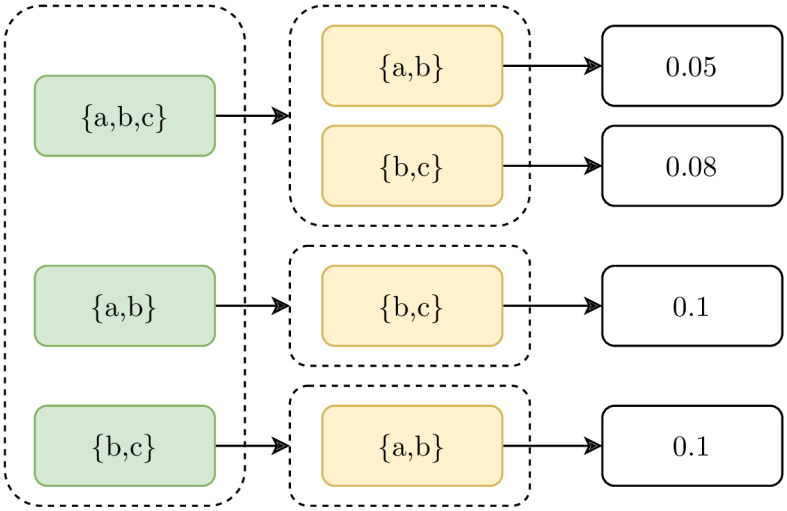
Tento nástroj realizuje algoritmus 3. Spuštění nástroje Tofa se provádí skrze program `app-overapproximation.py`, kterému je možné předat několik parametrů. Klíčovými parametry jsou cesta k souboru s konečným automatem ve formátu `msfm` určenému k aproximaci, hranice míry aproximace znakových tříd a cesta k souboru s hodnotami signifikance stavů. Celý algoritmus je implementován ve třídě `TransitionOverapproximator`. Výstupem nástroje Tofa je pak aproximovaný automat v obou používaných formátech, tedy `msfm` i `fa`.



Obrázek 5.6: Schéma aproximačního nástroje Tofa.

Pro implementaci byla napřed využita knihovna `itertools` pro jazyk Python 3. Bohužel to se ukázalo být velmi nevhodné, protože příliš zpomalovala celou aproximaci a tak od ní bylo nakonec upuštěno. Funkce, které jsem z ní chtěl využít jsem nakonec převážně realizoval pomocí základních funkcí a konstrukcí jazyka Python 3. Dvouprvkové kombinace znakových tříd a výsledné hodnoty míry aproximace získané jejich spojením jsou reprezentovány zanořenými slovníky. Slovník je v jazyce Python ekvivalentem asociativního pole. Konstrukce zanořených slovníků tedy vypadá následně. Vnější a tedy hlavní slovník obsahuje klíče s každou znakovou třídou. Každý klíč odkazuje na nový slovník, tedy slovník vnitřní, jenž obsahuje další klíče se znakovými třídami. Zde už ovšem platí omezení, že

každý vnitřní slovník obsahuje pouze klíče těch znakových tříd, které jsou do počtu znaků menší nebo stejně velké jako klíč vnějšího slovníku, pod kterým se vnitřní slovník nachází. Je to z toho důvodu částečného odstranění redundance. Pod každým klíčem všech vnitřních slovníků se pak již nachází hodnota míry aproximace, které by bylo dosaženo spojením znakové třídy v klíči vnějšího slovníku a znakové třídy slovníku vnitřního. Znakové třídy jsou reprezentovány datovým typem `frozenset`, tedy neměnitelnou množinou, která může být použita pro indexování ve slovníku, na rozdíl od datového typu `set`, tedy měnitelné množiny. Tento princip je ilustrován na obrázku 5.7.



Obrázek 5.7: Schéma datové struktury zanořených slovníků pro realizování dvouprvkových kombinací znakových tříd.

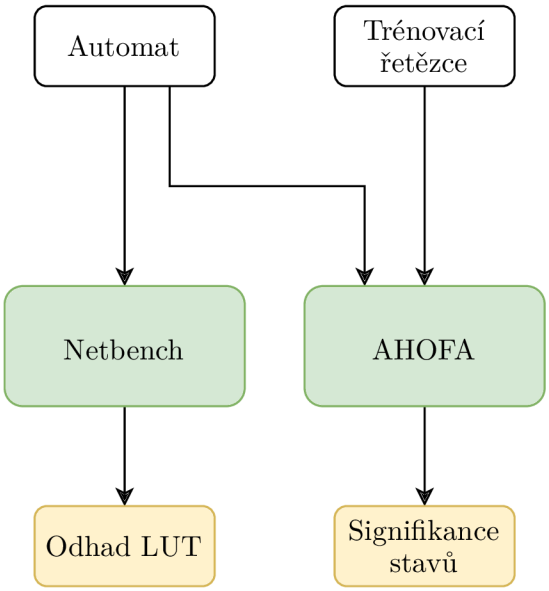
5.2.3 Testovací prostředí

Posledním z mnou implementovaných nástrojů je testovací prostředí STOTS. Slouží k provádění experimentů s přechodovou aproximací. Základní myšlenka tkví v otestování přechodové aproximace napřed samostatně a následně v kombinaci se stavovou aproximací. Testovací proces lze spustit přes program `stots.py`. Jako vstupní parametry je nutné uvést hlavně jméno souboru s automatem, jehož otestování má být provedeno.

Veškerá práce se soubory v rámci testování je vykonávána v adresáři `automata`. Zde se pro testování očekává přítomnost automatu, jenž je určen k testování, ve formátu `fa` a `msfm` v adresářích `original_fa` a `original_msfm`. Dále je nutné, aby v adresáři `testing_data` byl přítomen podadresář s názvem shodným s testovaným automatem, jenž musí obsahovat právě jeden soubor s trénovacími řetězci a alespoň jeden soubor s testovacími řetězci. Souborů s testovacími řetězci je ovšem doporučeno použít více. V případě absence této adresářové struktury je možné ji nechat vygenerovat. Skript ještě navíc potřebuje pro své správné fungování nástroje NETBENCH, AHOFA a TOFA, jenž jsou z testovacího skriptu postupně spouštěny za pomoci knihovny `subprocess` pro jazyk Python.

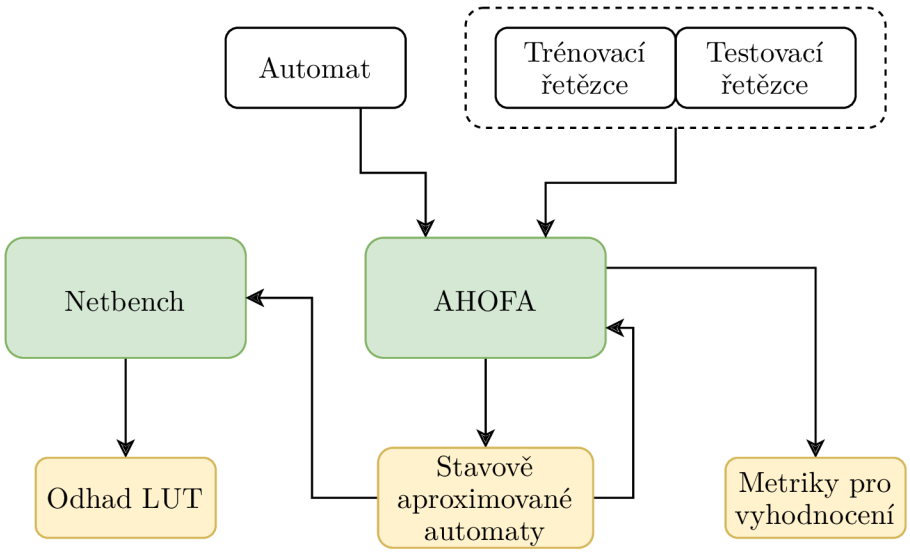
Celé testování lze rozdělit na tři samostatné celky. Nejprve je získán odhad LUT pro testovací automat z nástroje NETBENCH a signifikance jeho stavů z AHOFA.

Ve druhé části je testovací automat zredukován pomocí nástroje AHOFA na několik předem zvolených měr. Pro všechny takto stavově aproximované automaty je proveden opět



Obrázek 5.8: Schéma první fáze testovacího skriptu STOTS.

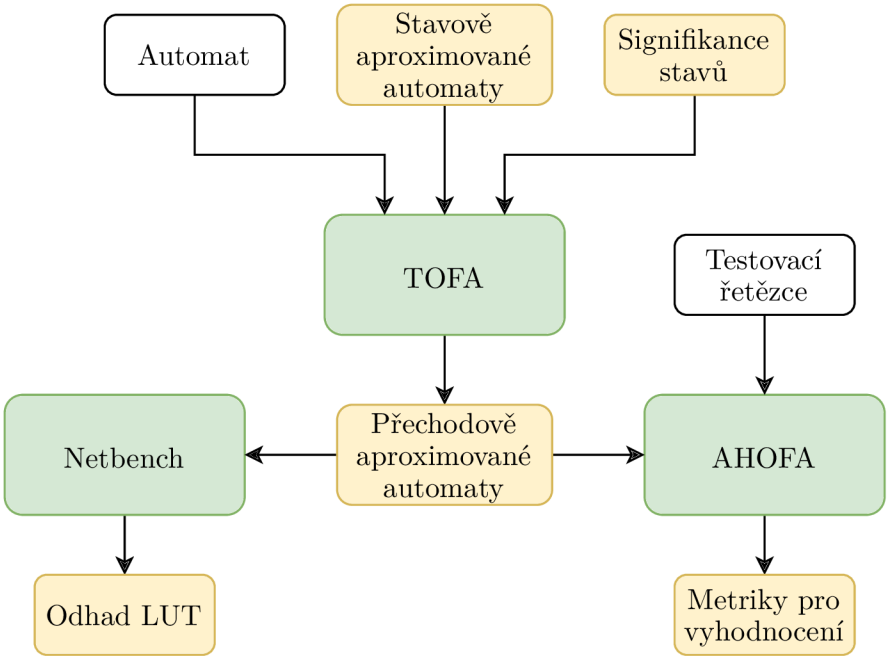
odhad LUT. Navíc je tentokrát nutné ještě ověřit přesnost takto redukovaných automatů získáním vyhodnocovacích metrik. Opět k tomuto účelu poslouží AHOFA.



Obrázek 5.9: Schéma druhé fáze testovacího skriptu STOTS.

Poslední část pak již testuje přímo přechodovou aproximaci. Jak originální testovací automat, tak jeho stavově aproximované verze jsou společně se signifikancemi stavů předány skriptu TOFA, který je přechodově aproximuje na předem zvolenou hranici. Nakonec je potřeba pro všechny výstupní automaty znova spočítat odhad LUT a získat evaluační metriky. Výsledkem celého testovacího procesu je tedy aproximace automatu v několika variantách, odhady spotřeb FPGA zdrojů a metriky pro ověření přesnosti. První a druhou fázi experimentů je potřeba provést vždy pouze jednou, jelikož všechny data z ní získané jsou znovupoužitelné. Opakovat je tedy potřeba pouze fázi poslední s různými hranicemi.

Skript celkové statistiky ukládá přímo do webové aplikace Google tabulky za použití modulu `gspread` pro jazyk Python a vývojářského API pro Google tabulky[1].



Obrázek 5.10: Schéma třetí fáze testovacího skriptu Stots.

[1] Viz https://developers.google.com/sheets/api

Kapitola 6

Experimenty

V této kapitole budou detailně rozebrány výsledky experimentů, jenž byly provedeny k otestování účinnosti a přesnosti nově navržené heuristiky nadaproximující přechody automatu. Na začátek je nutno si položit dvě hlavní otázky, na které se experimenty budou snažit najít odpověď:

1) Jaká je úspěšnost přechodové aproximace ve srovnání s aproximací stavovou?

2) Má smysl přechodové a stavové aproximace kombinovat?

6.1 Nastavení experimentů

V této sekci bude ukázáno s jakými daty a na jakých automatech byly experimenty prováděny. Jak automaty, tak testovací a trénovací data, tedy řetězce, byly vygenerovány pomocí již zmíněného generátoru z podsekce 5.2.1. Ke správnému vyhodnocení nestačí pouze data k experimentování, důležité jsou rovněž sledované statistiky. Mezi ně patří hranice míry aproximace, odhad LUT a metriky reflektující přesnost redukovaného automatu. Počty registrů nejsou podstatné.

- **Hranice míry aproximace znakové třídy** – jedná se o hranici pro metriku představené v podsekci 4.3.2, tedy míry aproximace znakové třídy. Tato hodnota dovoluje celou přechodovou aproximaci řídit a přesně určit toleranci pro spojování znakových tříd. Uváděna bude v tabulkách pouze orientačně, protože sama nemá velkou výpovědní hodnotu, jelikož její nastavení nijak zvlášť nekoreluje se zavedenou chybou nebo získanou úsporou LUT. Ve všech tabulkách bude označována pouze jako „Hranice“.

- **LUT** – již velmi dobře známá statistika, která bude sloužit jako ukazatel reálné úspory, kterou aproximace přinesla. V tabulkách bude vždy uveden počet LUT a za ním v závorce, kolik procent LUT z původního nijak neredukovaného automatu představuje aktuální hodnota.

- **Přesnost akceptování (PA)** – první ze dvou vyhodnocovacích metrik určovaných nástrojem Ahofa viz podsekce 5.4. Jedná se o poměr řetězců přijatých automatem před redukcí a po redukci. Rovnicí lze přesnost akceptování vyjádřit jako $PA = \frac{A_{SA}}{A_{SA}+A_{NA}}$, kde A_{SA} (správně akceptovány) je počet řetězců, které automat správně přijal a A_{NA} (nesprávně akceptovány), je počet řetězců, které redukovaný automat akceptoval, ale původní automat nikoliv. Pro přesnost akceptování platí, že $PA \in (0, 1]$.

Teoreticky by se dalo uvažovat i o tom, že $PA \in [0, 1]$, jelikož $PA = 0$ může nastat, když $A_{SA} = 0$. V takovém případě by tato metrika ovšem nebyla o ničem vypovídající a proto takový případ nebude uvažován a bude předpokládáno, že sada testovacích řetězců obsahuje alespoň jeden akceptovaný řetězec.

- **Přesnost celková (PC)** – druhá z vyhodnocovacích metrik naopak ukáže, jaký poměr všech řetězců byl klasifikován správně. Výpočet lze vyjádřit jako $PC = \frac{S - A_{NA}}{S}$, kde S je celkový počet testovacích řetězců. Protože se jedná o nadaproximaci a vždy budou testovací řetězce obsahovat alespoň jeden akceptovaný řetězce, tak vždy platí, že $S \neq A_{NA}$ a proto $PC \in (0, 1]$.

6.1.1 Data pro experimenty

Nejprve byla strategie generování řetězců založena na principu rovnoměrné distribuce řetězců, tudíž každý akceptující řetězec se generoval na základě náhodného průchodu konečným automatem až do nějakého koncového stavu, a každý neakceptující řetězec se generoval na základě částečného průchodu automatem, kdy po navštívení každého nekoncového stavu byla náhodná šance, že generování řetězce bude ukončeno, aby bylo zachována jeho neakceptovatelnost. Pro každý automat bylo vždy vygenerováno 11 souborů, každý s 300 000 řetězci. Zastoupení akceptujících řetězců se v každém souboru vždy pohybovalo od 6 % do 14 %. První soubor byl vždy použit jako trénovací, zbylých 10 jako testovací. Experimenty tedy byly provedeny s 300 000 trénovacími řetězci a s 3 000 000 testovacími řetězci pro každý konečný automat.

Rovnoměrná distribuce měla velmi nežádoucí efekt na stavovou aproximaci prořezávání. Ta totiž počítá s faktem, že některými stavy automatu projde velké množství řetězců a jinými naopak malé množství. Právě takové stavy pak odstraní. V případě, že ovšem všechny testovací řetězce mají rovnoměrnou distribuci, tak prakticky jakékoliv odebrání stavu potom způsobí rychlý narůst nepřesnosti. Z tohoto důvodu byla dále změněna strategie generování testovacích a trénovacích řetězců tak, že se jednotlivým cestám automatu přidělily váhy. Generování dat pak probíhalo přes nejvíce váženou cestu s daleko větší frekvencí než u cest ostatních.

Pro reálné automaty z filtrace síťového provozu byly navíc krom vygenerovaných textových řetězců využity i opravdové pakety z binárních souborů PCAP pro volné využití, získané z veřejně dostupného repozitáře na platformě GitHub[1].

6.1.2 Automaty pro experimenty

Jednou z možných oblastí, kde by se exploze znakových tříd mohla objevit, bylo využití kódování UTF-8 nebo UTF-16 a proto byl původní plán syntetické automaty generovat s abecedou pokrývající tyto velmi rozsáhlé skupiny znaků. Bohužel zde bylo naraženo na technické limity, jelikož jak Ahofa, tak Netbench podporují pouze automaty s abecedou o rozsahu `0x00` až `0xFF`, tedy rozšířenou ASCII. Snaha byla i o nalezení reálných regulárních výrazů využívající právě UTF-8 nebo UTF-16, transformace těchto regulárních výrazů na konečný automat a následné experimenty, avšak tento pokus ztroskotal ze stejného důvodu. Experimenty s automaty s abecedou za hranice rozšířené ASCII by tudíž vyžadovaly reimplementaci obou výše zmíněných nástrojů. Experimenty s abecedou o větším rozsahu než-li má rozšířená ASCII mám v plánu provést v další práci týkající se tohoto tématu.

[1] Viz https://github.com/ondrik-network-hw/traffic-dumps

Prvotní experimenty tak byly provedeny se syntetickými deterministickými konečnými automaty. Přechody experimentálních automatů byly generovány tak, aby vedly na velký počet unikátních znakových tříd. Ty byly pro tyto experimenty generovány tak, aby měly navzájem podobný počet znaků a jejich spojování bylo tak relativně výhodné, na rozdíl od dalších experimentů, kde již se budou generovat znakové třídy s většími velikostními rozdíly. Celkem bylo takto vytvořeno 12 syntetických automatů. Polovina těchto automatů byla testována s rovnoměrným rozložením testovacích a trénovacích dat a druhé polovina s nerovnoměrným rozložením. Dále bylo vygenerováno ještě 6 syntetických NKA s nerovnoměrným rozložením dat. Krom přidání nedeterminismu byl u automatů ještě změněn způsob generování přechodů. Ty byly generovány tak, aby znakové třídy byly počtem znaků více různorodé. NKA byly navíc po vygenerování ještě minimalizovány nástrojem Reduce [7], aby bylo zajištěno, že neexistuje jejich verze, která je mnohonásobně menší, jelikož by v takovém případě nebyly výsledky experimentů moc vypovídající. Dalo by se uvažovat i využití minimalizace na již přechodově aproximované automaty, nicméně to by mohlo vést k porušení znakových tříd a mohlo by to teoreticky i zhoršit spotřeby.

Pro druhou fázi experimentů pak byl použit automat získaný ze sady pravidel linuxového klasifikátoru L7-filter[2], který lze ve formátu `fa` nalézt v repozitáři nástroje Ahofa [15] pod jménem `l7-all`. Experimenty s ním byly provedeny napřed s vygenerovanými řetězci s nerovnoměrným rozložením a posléze i s reálnými pakety z volně dostupného síťového provozu.

6.2 Výsledky experimentů se syntetickými automaty

Výsledky experimentů budou rozděleny do tří částí. V první části budou představeny experimenty s DKA na datech s uniformním rozložením generovaných dat, v druhé části pak DKA s neuniformním rozložením dat a ve třetí části pak NKA opět se neuniformním rozložením dat.

6.2.1 DKA s rovnoměrným rozložením dat

První experimenty byl provedeny se šesti DKA s různými velikostmi a s rovnoměrně generovanými daty. Nejdříve je nutné ukázat hlavní rozdíl mezi metrikou PC a PA a určit, která bude důležitější pro sledování. Jako modelový příklad na to posloužil automatem s názvem `DKA 5` s 900 stavy a 11 400 přechody.

Tabulka 6.1 obsahuje statistiky `DKA 5`, na který bylo aplikováno prořezávání stavů a pak dodatečně navíc aproximace přechodů. Lze vidět, že PA oproti PC klesá výrazněji. Je to způsobeno tím, že PA v sobě zahrnuje pouze akceptované řetězce, zatímco PC počítá se všemi řetězci, tím pádem je PA striktnější a může velmi prudce klesnou na nízké hodnoty, ačkoliv PC zůstane na hodnotách relativně vysokých. Celkově objektivnější je ovšem pro experimenty sledování metriky PC, jelikož právě ona bere v potaz všechny testovací řetězce. Metrika PA je sama o sobě vhodnější a důležitější v oblasti síťového provozu a zde nebude nadále využívána. Posloužila pouze k ukázce kontrastu mezi všemi řetězci a akceptujícími řetězci.

[2] Viz http://l7-filter.sourceforge.net/

Hranice	LUT	PC	PA
-	3017 (95.11%)	0.969097	0.717189
0.0005	3001 (94.61%)	0.967954	0.706000
0.0010	2832 (89.28%)	0.966098	0.689080
0.0015	2609 (82.25%)	0.963944	0.669325
0.0020	2460 (77.55%)	0.960969	0.642038
0.0025	2378 (74.97%)	0.955875	0.595320
0.0030	2339 (73.74%)	0.954467	0.582403
0.0035	2291 (72.23%)	0.953283	0.571674
0.0040	2265 (71.41%)	0.951640	0.556806
0.0045	2226 (70.18%)	0.950593	0.547229
0.0050	2201 (69.39%)	0.949749	0.539497

Tabulka 6.1: `DKA 5` s kombinací přechodové aproximace a prořezáváním stavů (míra 0.9).

6.2.2 DKA s nerovnoměrným rozložením dat

Jako hlavní reprezentant první šestice DKA byl vybrán automat s názvem `DKA 6` s 1000 stavy a 12 660 přechody. Tabulka 6.2a ukazuje automat `DKA 6` s pouhou aplikací přechodové aproximace. Naproti tomu tabulka 6.2b ukazuje tentýž automat navíc se stavovou aproximací prořezávání s mírou 0.9 a navíc s dodatečnou aplikací přechodové aproximace. Míra 0.8 uvedená není, protože výsledky v kombinaci s ní dosahovaly již relativně velkých chyb. Celkové srovnání `DKA 6` je pak znázorněno v grafu na obrázku 6.1. Z tabulek lze vyvodit, že zatímco přechodová aproximace je schopná se zhoršením přesnosti PC zhruba o 3 % ušetřit automatu 871 LUT, tedy skoro 25 %. Samotné prořezávání stavů pak ušetří pouhých 182 LUT při zavedeních velmi podobné chyby.

Hranice	LUT	PC
-	3517 (100.00%)	1
0.0005	3491 (99.26%)	0.999951
0.0010	3418 (97.19%)	0.997385
0.0015	3262 (92.75%)	0.994821
0.0020	3050 (86.72%)	0.986202
0.0025	2965 (84.30%)	0.979859
0.0030	2899 (82.43%)	0.978112
0.0035	2847 (80.95%)	0.977215
0.0040	2786 (79.22%)	0.974378
0.0045	2731 (77.65%)	0.972160
0.0050	2646 (75.23%)	0.968087

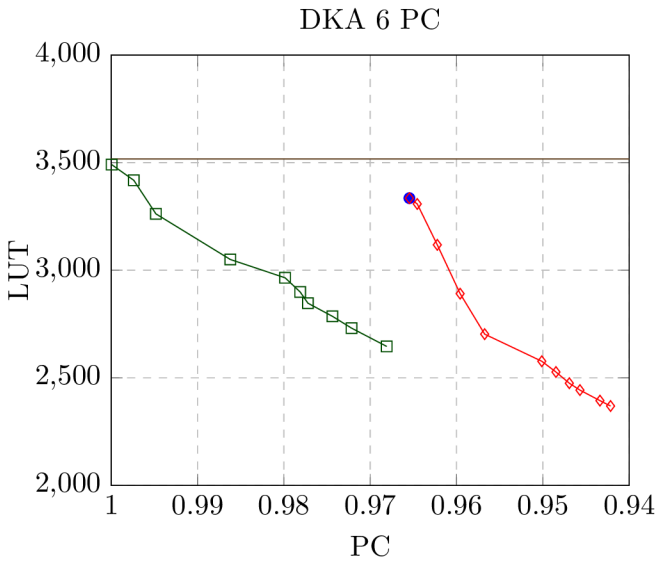
(a) Přechodová aproximace

Hranice	LUT	PC
-	3335 (94.83%)	0.965457
0.0005	3308 (94.06%)	0.964547
0.0010	3118 (88.66%)	0.962226
0.0015	2891 (82.20%)	0.959578
0.0020	2703 (76.86%)	0.956738
0.0025	2577 (73.27%)	0.950120
0.0030	2527 (71.85%)	0.948479
0.0035	2475 (70.37%)	0.946934
0.0040	2443 (69.46%)	0.945703
0.0045	2394 (68.07%)	0.943385
0.0050	2369 (67.36%)	0.942156

(b) Prořezávání stavů (míra 0.9) + přechodová aproximace

Tabulka 6.2: Srovnání přechodových, stavových a kombinovaných aproximací `DKA 6`.

S další šesticí automatů byly provedeny stejné experimenty, ovšem nyní za použití nerovnoměrného generování řetězců. Pro ukázku byl vybrán opět jeden automat a to `DKA 12` taktéž s 1000 stavy a 12 660 přechody. Tabulka 6.3 znázorňuje originální automat s užitím přechodové aproximace a tabulky 6.4a a 6.4b užití stavové a kombinované aproximace.



Obrázek 6.1: Srovnání PC a LUT pro přechodovou, stavovou a kombinovanou aproximaci pro automat `DKA 6`. Hnědá čára značí hodnotu LUT originálního automatu, zelená značí přechodovou aproximaci, červená čára pak značí po kombinaci přechodové a stavové aproximace s mírou 0.9, přičemž modrý bod označuje samotnou stavovou aproximaci.

Z tabulek vyplývá, že nerovnoměrné generování testovacích řetězců mělo kladný vliv na prořezávání stavů, kde se velmi zlepšila metrika PC oproti tabulce 6.2b. Pozitivní vliv mělo neuniformní generování řetězců i na přechodovou aproximaci, která za snížení PC o necelá 2 % přináší úsporu 1 272 LUT, což je úspora cca o 36 %. Dále lze pozorovat, že v tabulkách 6.4a a 6.4b přechodová aproximace pozitivně doplňuje prořezávání stavů a za snížení chyby o další 1 % šetří velmi značné procento LUT. Celkové srovnání `DKA 12` je možné si prohlédnout v grafu na obrázku 6.2.

Hranice	**LUT**	**PC**
-	3517 (100.00%)	1
0.0005	2967 (84.36%)	0.998412
0.0010	2756 (78.36%)	0.996065
0.0015	2639 (75.04%)	0.994133
0.0020	2542 (72.28%)	0.991878
0.0025	2470 (70.23%)	0.990189
0.0030	2411 (68.55%)	0.987854
0.0035	2357 (67.02%)	0.986664
0.0040	2300 (65.40%)	0.985576
0.0045	2286 (65.00%)	0.984340
0.0050	2245 (63.83%)	0.982768

Tabulka 6.3: `DKA 12` s aplikací přechodové aproximace.

Z experimentů na DKA lze vyvodit, že přechodová aproximace má víceméně lineární pokles LUT i PC a dovoluje dělat pozvolnější kompromis mezi LUT a PC oproti prořezávání stavů. Rovněž tyto experimenty ukazují, že na takové automaty je vhodnější použít

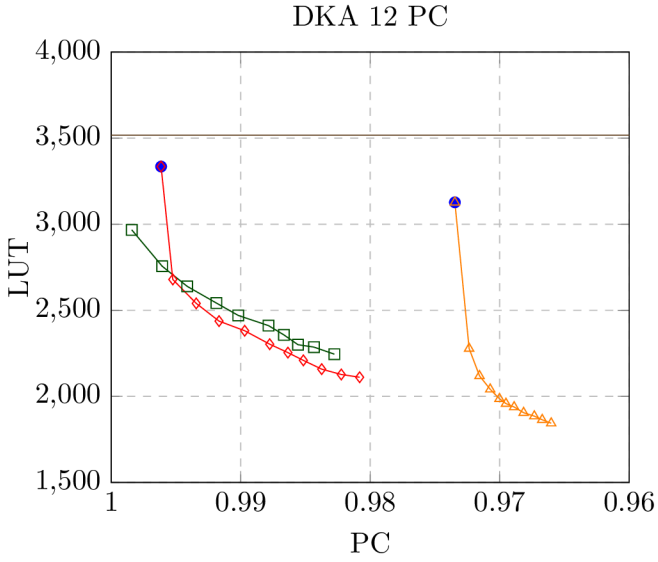
Hranice	LUT	PC
-	3335 (94.83%)	0.996147
0.0005	2679 (76.17%)	0.995243
0.0010	2540 (72.22%)	0.993434
0.0015	2437 (69.29%)	0.991672
0.0020	2381 (67.70%)	0.989691
0.0025	2303 (65.48%)	0.987759
0.0030	2254 (64.09%)	0.986353
0.0035	2209 (62.81%)	0.985160
0.0040	2158 (61.36%)	0.983745
0.0045	2127 (60.48%)	0.982234
0.0050	2111 (60.02%)	0.980815

(a) Prořezávání stavů (míra 0.9) + přechodová aproximace

Hranice	LUT	PC
-	3127 (88.91%)	0.973459
0.0005	2278 (64.77%)	0.972360
0.0010	2119 (60.25%)	0.971557
0.0015	2041 (58.03%)	0.970731
0.0020	1986 (56.47%)	0.970015
0.0025	1958 (55.67%)	0.969515
0.0030	1938 (55.10%)	0.968884
0.0035	1905 (54.17%)	0.968152
0.0040	1885 (53.60%)	0.967325
0.0045	1864 (53.00%)	0.966715
0.0050	1844 (52.43%)	0.966022

(b) Prořezávání stavů (míra 0.8) + přechodová aproximace

Tabulka 6.4: Srovnání stavových a kombinovaných aproximací `DKA 12`.



Obrázek 6.2: Srovnání PC a LUT pro přechodovou, stavovou a kombinovanou aproximaci pro automat `DKA 12`. Hnědá čára značí hodnotu LUT originálního automatu, zelená značí přechodovou aproximaci, červená a oranžová čára pak značí po řadě kombinaci přechodové a stavové aproximace s mírou 0.9 a 0.8, přičemž modré body označují samotnou stavovou aproximaci.

přechodovou aproximaci nebo případně se i vyplatí použít napřed prořezávání stavů a pak dodatečně přechodově aproximovat.

6.2.3 NKA s nerovnoměrným rozložením dat

Další experimenty byly provedeny s opět se šesticí automatů, tentokrát ovšem s NKA. Bylo použito nerovnoměrné generování dat a větší velikostní variabilita znakových tříd. Pro demonstraci byl vybrán automat `NKA 6`, který měl před minimalizací pomocí nástroje

Reduce 800 stavů a 20 111 přechodů. Po minimalizaci se počet stavů nezměnil a počet přechodů se snížil na 20 107.

Hranice	LUT	PC
-	5231 (100.00%)	1
0.05	4894 (93.56%)	0.980416
0.06	4774 (91.26%)	0.964641
0.07	4688 (89.62%)	0.951385
0.08	4574 (87.44%)	0.936554
0.09	4484 (85.72%)	0.929426
0.10	4399 (84.09%)	0.917895
0.15	3990 (76.28%)	0.868394
0.20	3700 (70.73%)	0.836129
0.25	3488 (66.68%)	0.817179
0.30	3311 (63.30%)	0.802009

(a) Přechodová aproximace

Hranice	LUT	PC
-	4810 (91.95%)	0.711648
0.05	4196 (80.21%)	0.708310
0.06	4102 (78.42%)	0.708018
0.07	4035 (77.14%)	0.707708
0.08	3957 (75.65%)	0.706992
0.09	3872 (74.02%)	0.706751
0.10	3820 (73.03%)	0.706575
0.15	3594 (68.71%)	0.704953
0.20	3356 (64.16%)	0.701853
0.25	3124 (59.72%)	0.697594
0.30	2971 (56.80%)	0.694314

(b) Prořezávání stavů (míra 0.9) + přechodová aproximace

Hranice	LUT	PC
-	4320 (82.58%)	0.549869
0.05	3630 (69.39%)	0.550545
0.06	3494 (66.79%)	0.550460
0.07	3380 (64.61%)	0.550414
0.08	3293 (62.95%)	0.550328
0.09	3186 (60.91%)	0.550085
0.10	3120 (59.64%)	0.549934
0.15	2833 (54.16%)	0.549607
0.20	2633 (50.33%)	0.549187
0.25	2530 (48.37%)	0.549135
0.30	2451 (46.86%)	0.548928

(c) Prořezávání stavů (míra 0.8) + přechodová aproximace

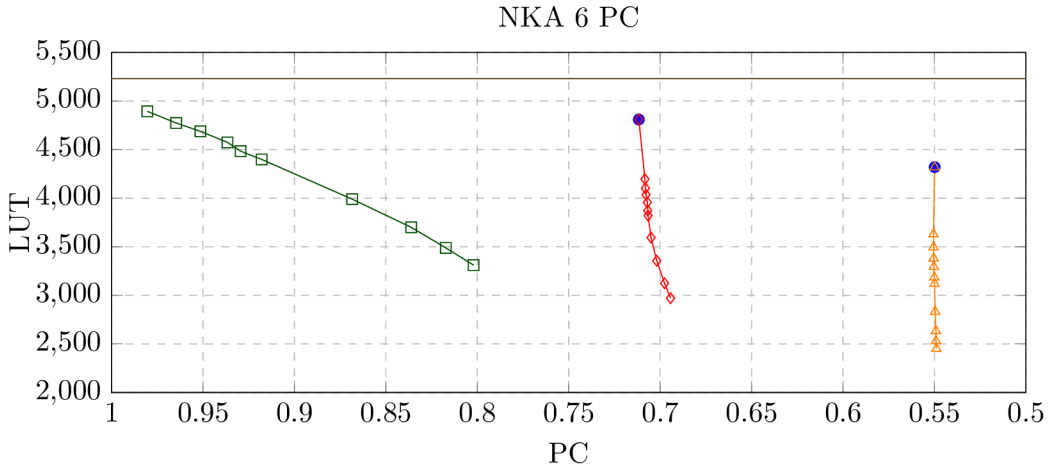
Tabulka 6.5: Srovnání přechodových, stavových a kombinovaných aproximací NKA 6.

Tabulka 6.5a ilustruje `NKA 6` s užitím pouze přechodové aproximace a tabulky 6.5b a 6.5c pak i prořezávání stavů a kombinovanou aproximaci. Oproti předchozí sekci zde musely být více benevolentní hranice. Příčinou toho je větší různorodost znakových tříd co do počtu znaků, kdy při podobných hranicích, jako byly užity na DKA z předchozí sekce, by došlo k velmi mizivým redukcím. V tabulce 6.5a už je zavedená chyba o něco horší a ačkoliv stále klesá víceméně lineárně, tak při zhoršení PC o 2 % dojde k úspoře pouze 337 LUT, což je asi 6.5 %. Daleko zajímavější redukce LUT lze potom získat při zhoršení PC o 20 %, kde je za tuto cenu ušetřeno 1920 LUT, tedy skoro 37 %.

V kontrastu s DKA z tabulky 6.3, kde byla úspora 36 % získána za pouhé 2 % chyby PC lze usoudit, že přechodová aproximace se lépe vede na automatech, kde nejsou takové velikostní rozdíly mezi znakovými třídami. Na druhou stranu, úspora je stále velice značná a chyby nejsou na nižších hranicích tolik markantní.

Na tabulkách 6.5b a 6.5c při použití samotné stavové i kombinované aproximace pak lze pozorovat větší úspory LUT, než tomu bylo u DKA. Bohužel výměnou za to, lze též vidět velké skoky u snižování PC a to zvlášť při aplikaci samotného prořezávání stavů. Dále

je třeba si povšimnout faktu, že kombinace přechodové a stavové aproximace má v tomto případě velmi pozitivní výsledek. Například v tabulce 6.5b dodatečná přechodová aproximace ušetří dalších 1839 LUT, což je 35 % ze spotřeby originálního automatu a to pouze za zhoršení PC o další 2 %. Tabulka 6.5c pak ukazuje podobný fenomén. Celkové srovnání PC u pouhé přechodové aproximace, stavové aproximace a aproximace kombinované znázorňuje graf v obrázku 6.3.



Obrázek 6.3: Srovnání PC a LUT pro přechodovou, stavovou a kombinovanou aproximaci pro automat `NKA 6`. Hnědá čára značí hodnotu LUT originálního automatu, zelená značí přechodovou aproximaci, červená a oranžová čára pak značí po řadě kombinaci přechodové a stavové aproximace s mírou 0.9 a 0.8, přičemž modré body označují samotnou stavovou aproximaci.

Shrnutím experimentů se syntetickými konečnými automaty lze konstatovat několik drobnějších „závěrů“. Nejedná se ovšem o závěr v pravém slova smyslu, jelikož se zatím jednalo pouze o první část experimentování. Nejprve lze říci, že na syntetických konečných automatech, a to na DKA i NKA, nedosahují techniky stavových aproximací tak dobrých výsledků, jako v článku [16]. Způsobeno je to zejména způsobem generování syntetických testovacích a trénovacích dat, jelikož i při neuniformním rozložení při generování dat je pořád využit skoro celý automat a tím pádem pak jakákoliv stavová aproximace velmi negativně ovlivní celkovou přesnost. Důkazem pro to jsou i signifikance jednotlivých stavů, kdy u synteticky generovaných automatů a dat jsou i ty nejmenší získané hodnoty signifikancí stále relativně vysoké, oproti získaným signifikancím při experimentech s reálnými daty v nadcházející sekci. Na druhou stranu přechodová aproximace zde dosahuje značně lepších výsledků, kdy dokáže stavové obstojně konkurovat v ušetřených zdrojích i přesnosti. Kombinace obou typů aproximací pak jednoznačně funguje nejlépe, jelikož při dodatečné přechodové aproximaci pak velmi nepatrně roste chyba výměnou za velmi značný pokles spotřeby LUT. Přechodovou aproximační techniku je dále potřeba otestovat na modelech z reálného světa.

6.3 Experimenty na automatech z filtrace síťového provozu

Z oblasti filtrace síťového provozu se experimenty týkaly automatu `l7-all` s 7 280 stavy a 2 647 620 přechody, jenž byl získán z pravidel síťového klasifikátoru L7-filter. Jako trénovací i testovací data zde byly využity reálné pakety místo obyčejných řetězců.

Hranice	LUT	PC
-	8792 (100.00%)	1
0.5	8638 (98.25%)	0.998558
1.0	8620 (98.04%)	0.998195
1.5	8624 (98.09%)	0.995940
2.0	8617 (98.01%)	0.995866
2.5	8867 (100.85%)	0.971939
3.0	8880 (101.00%)	0.965221

(a) Přechodová aproximace

Hranice	LUT	PC
-	8094 (92.06%)	1
0.5	7910 (89.97%)	0.998558
1.0	7892 (89.76%)	0.998222
1.5	7896 (89.81%)	0.995967
2.0	7889 (89.73%)	0.995897
2.5	8139 (92.57%)	0.971967
3.0	8152 (92.72%)	0.965250

(b) Prořezávání stavů (míra 0.9) + přechodová aproximace

Hranice	LUT	PC
-	7441 (84.63%)	1
0.5	7175 (81.61%)	0.998558
1.0	7157 (81.40%)	0.997794
1.5	7161 (81.45%)	0.995539
2.0	7154 (81.37%)	0.995401
2.5	7404 (84.21%)	0.971538
3.0	7417 (84.36%)	0.964821

(c) Prořezávání stavů (míra 0.8) + přechodová aproximace

Hranice	LUT	PC
-	6779 (77.10%)	0.999996
0.5	6450 (73.36%)	0.998554
1.0	6432 (73.16%)	0.998183
1.5	6436 (73.20%)	0.995928
2.0	6429 (73.12%)	0.995852
2.5	6680 (75.98%)	0.971929
3.0	6693 (76.13%)	0.965211

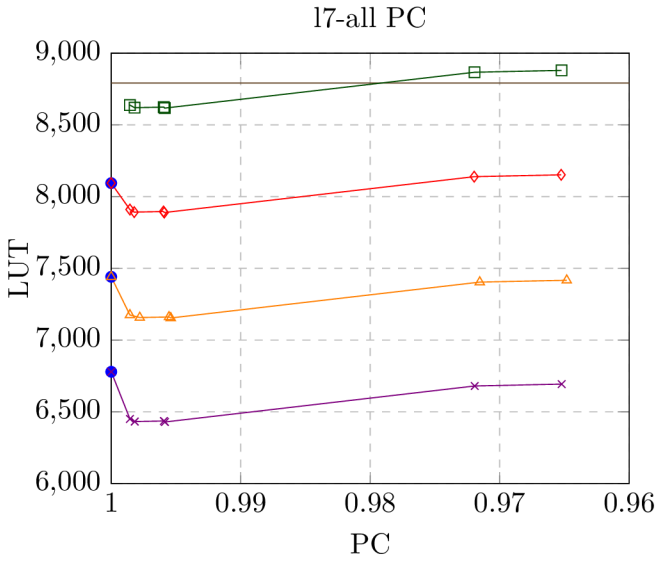
(d) Prořezávání stavů (míra 0.7) + přechodová aproximace

Tabulka 6.6: Srovnání přechodových, stavových a kombinovaných aproximací `l7-all` při užití paketů jako testovacích a trénovacích dat.

Tabulka 6.6a opět znázorňuje automat s aplikací pouhé přechodové aproximace a tabulky 6.6b, 6.6c a 6.6d pak stavové a kombinované aproximace. Celkové srovnání pak sumarizuje graf z obrázku 6.4. Z tabulek lze vidět, že samotná přechodová aproximace dokáže ušetřit 2 % LUT a kombinovaná dodatečně až 4 % LUT z celkových 27 % LUT při míře 0.7 za velmi nepatrné snížení přesnosti. Naproti tomu pouhá stavová aproximace ušetřila 23 % LUT při použité míře 0.7 za zachování takřka naprosté přesnosti. Z toho by se dalo usuzovat, že na reálný automat ze síťového provozu má přechodová aproximace jenom relativně malý účinek. Je ovšem potřeba brát v potaz fakt, že automat `l7-all` netrpěl explozí znakových tříd. Bylo jich zde relativně malé množství oproti syntetickým automatům z předchozí sekce a přesto přechodová aproximace dokázala dosáhnout úspor, byť ani zdaleka ne tak velkých, za nepatrné snížení chyby.

Dále je nutné si všimnout i úkazu, který se projevuje u výše nastavených hranicích pro přechodovou aproximaci. Ukazuje se, že na tomto automatu přechodová aproximace do určitého bodu šetří LUT a pouze lehce znepřesňuje automat a od tohoto bodu pak nadále spotřebu LUT zvyšuje, někdy dokonce i na vyšší hodnotu, než byla spotřeba původního automatu, a daleko více zhoršuje přesnost. Vysvětlení pro tento narůst LUT se skrývá ve vlastnostech znakových tříd, k jejichž spojení v rámci přechodové aproximace dochází. Je třeba si uvědomit, že nejprve se spojují znakové třídy, které jsou svými vlastnostmi výhodné

pro spojení. Zejména výhodné je spojování tříd, kde jedna ze spojovaných je podmnožinou té druhé, jelikož je třeba doplnit nové znaky pouze do jedné z nich. Naopak s postupem aproximování pak dojde i na třídy, jejichž spojení už tolik výhodné není, ovšem nastavená hranice toto spojení stále dovolí. Takové třídy jsou většinou velkého počtu znaků a navíc zde většinou neplatí to, že jedna ze spojovaných tříd je podmnožinou té druhé, tím pádem jejich spojením vznikne třída, která je do počtu znaků větší než byly původní třídy. Spotřeba LUT pro takto nově vytvořenou znakovou třídu pak může být větší, než byla spotřeba obou tříd, ze kterých byla utvořena. K tomuto jevu pak skutečně dochází právě ve zkoumaném automatu `l7-all`. Vyvodit z toho tedy lze to, že přechodová aproximace může mít i negativní vliv, pokud je jí dovoleno aproximovat na příliš vysokou hranici.



Obrázek 6.4: Srovnání PC a LUT pro přechodovou, stavovou a kombinovanou aproximaci pro automat `l7-all`. Hnědá čára značí hodnotu LUT originálního automatu, zelená značí přechodovou aproximaci, červená, oranžová a fialová čára pak značí po řadě kombinaci přechodové a stavové aproximace s mírou 0.9, 0.8 a 0.7, přičemž modré body označují samotnou stavovou aproximaci.

Pro shrnutí experimentů s automaty z filtrace síťového provozu lze říci opět několik poznatků. Stavová aproximace na takových automatech dosahuje bezesporu lepších úspor s velmi dobrým zachováním přesnosti, ostatně jak už bylo popsáno v článku [16]. Naopak přechodová aproximace zde tak dobrých výsledků nedosahuje a rovněž dokonce při určitých hranicích působí navíc negativní dopady. Nicméně při rozumných hranicích dokáže poskytnout mírnou dodatečnou úsporu s minimálním zhoršení přesnosti, což lze vzhledem ke skutečnosti, že u testovaných automatů nevzniká fenomén exploze znakových tříd, považovat za úspěch. Opět lze říci, že kombinace stavové i přechodové aproximace zde vychází nejlépe.

6.4 Shrnutí experimentů

Celkově lze experimenty shrnout ve třech krocích. Prvním krokem jsou relativně dobré výsledky, kterých přechodová aproximace dosáhla na syntetických automatech, které byly přímo modelovány tak, aby se u nich ve velké míře vyskytoval fenomén, jenž má být pře-

chodovou aproximací postupně eliminován, tedy exploze znakových tříd. Bylo tak dosaženo lepších výsledků, než u aproximace stavové. Konkrétně na DKA bylo dosaženo až 36 % LUT úspor za pouhé 2% snížení přesnosti s přechodovou aproximací, zatímco stavová aproximace se snížením přesnosti o skoro 3 % dosahuje pouhé 11% úspory na LUT. U NKA pak samostatná přechodová aproximace dokáže ušetřit cca 14 % LUT za zavedení 7% chyby a stavová aproximace je podobných čísel schopná dosáhnout při velice drastických chybách, které se pohybují kolem několika desítek procent.

Za druhé pak modely z reálného světa a to konkrétně automaty z filtrace síťového provozu ukázaly, že zde si vedla přechodová aproximace poněkud hůře a velmi zde naopak dominovala aproximace stavová. Nutno dodat, že to bylo i očekávaným výsledkem, jelikož automaty, které jsem měl z této oblasti k dispozici, netrpí fenoménem exploze znakových tříd a skromnější úspory přechodové aproximace lze i tak považovat za úspěch. Přechodová aproximace zde samostatně dokázala ušetřit zhruba 2 % LUT při zanedbatelné chybě necelého 1 % a je sama o sobě schopná ušetřit až skoro 4 %, při využití v kombinaci se stavovou aproximací. Stavová aproximace naopak zvládala ušetřit 23 % LUT, při podobně zanedbatelné chybě.

V třetím kroku pak je třeba vyzdvihnout užívání kombinované aproximace, které se ukázalo ve všech experimentech, které byly provedeny, jako nejvýhodnější.

Kapitola 7

Závěr

Tato práce pojednává o aproximačních technikách pro redukci konečných automatů. Práce vychází z technik již existujících, které se zaměřují hlavně na stavové aproximace, a více zapojuje do návrhu nových technik i hardwarovou stránku konečných automatů, tedy vlastnosti takových automatů při jejich syntéze do hardware. Nově pak navrhuje heuristickou redukční techniku přechodové nadaproximace, která je založena na bázi hledání a spojování znakových tříd, což jsou v podstatě množiny znaků, přes které je možné vést přechody mezi určitými stavy konečného automatu. Užitím těchto spojení je pak redukována spotřební zátěž LUT na znakovém dekodéru.

Navrženou heuristiku jsem implementoval ve formě aproximačního nástroje Tofa, který pro celé fungování využívá pouze základních konstrukcí programovacího jazyka Python 3. Nástroj Tofa spoléhá na dodatečné poskytnutí hodnot signifikance stavů nástrojem Ahofa. Vyhodnocení spotřeby LUT pak zajišťuje nástroj Netbench.

Funkčnost aproximačního nástroje jsem následně ověřil řadou experimentů na rozličných automatech, které jsem buďto generoval pomocí skriptů a nebo jsem je získal přímo z reálných případů užití. Během experimentů jsem se zaměřoval zejména na spotřeby LUT a chybu, jenž byla do automatu zanesena nadaproximací. Výsledky ukázaly velmi dobrou funkčnost na automatech syntetických, kde se záměrně objevoval problém velkého počtu znakových tříd, jenž tato přechodová aproximace primárně řeší. Střídmějších výsledků pak bylo dosaženo na automatech pro filtraci síťového provozu, kde se ovšem problém znakových tříd vyskytoval pouze ve velmi malé míře a proto i tyto úspory lze považovat za vyhovující. Experimenty by bylo potřeba dále provést i na automatech s rozsáhlejší abecedou, než je pouze rozšířená ASCII. V tomto případě jsem byl ovšem limitován abecedním rozsahem, který podporovaly oba mnou využívané nástroje třetích stran.

Do budoucna je v plánu provést zmíněné experimenty s regulárními výrazy s abecedou za hranicemi rozšířené ASCII a tím tak experimentálně ověřit využitelnost navržené heuristiky ve větším měřítku. Získání těchto výsledků je nezbytným podkladem pro možnost sepsání plnohodnotného vědeckého článku, který by mohl být dále publikován na některé z vědeckých konferencích, například TACAS nebo FCCM.

Literatura

[1] Clark, C. R. a Schimmel, D. E. Efficient Reconfigurable Logic Circuits for Matching Complex Network Intrusion Detection Patterns. In: Y. K. Cheung, P. a Constantinides, G. A., ed. *Field Programmable Logic and Application*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, s. 956–959. ISBN 978-3-540-45234-8.

[2] Clark, C. R. a Schimmel, D. E. Scalable Pattern Matching for High Speed Networks. In: *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. USA: IEEE Computer Society, 2004, s. 249–257. FCCM '04. ISBN 0769522300.

[3] Kozen, D. C. *Automata and Computability*. 1. vyd. Berlin, Heidelberg: Springer-Verlag, 1997. ISBN 0387949070.

[4] Martínek, T. *Návrh číslicových systémů* [online]. [cit. 2020-06-05]. Dostupné z: https://wis.fit.vutbr.cz/FIT/st/cfs.php?file=%2Fcourse%2FINC-IT%2Flectures%2Ftechnologie_fpga.pdf&cid=12169.

[5] Matoušek, D., Kořenek, J. a Puš, V. High-speed Regular Expression Matching with Pipelined Automata. In: *Proceedings of the 2016 International Conference on Field Programmable Technology*. IEEE Computer Society, 2016, s. 93–100. DOI: 10.1109/FPT.2016.7929431. ISBN 978-1-5090-5602-6. Dostupné z: https://www.fit.vut.cz/research/publication/11288.

[6] Matoušek, P. *Síťové služby a jejich architektura*. Publishing house of Brno University of Technology VUTIUM, 2014. 396 s. ISBN 978-80-214-3766-1. Dostupné z: https://www.fit.vut.cz/research/publication/10567.

[7] Mayr, R. et al. *RABIT and Reduce* [online]. [cit. 2020-05-14]. Dostupné z: http://languageinclusion.org/doku.php?id=tools.

[8] Meduna, A. a Lukáš, R. *Formální jazyky a překladače - Abecedy, řetězce a jazyky* [online]. [cit. 2020-05-09]. Dostupné z: https://www.fit.vutbr.cz/study/courses/IFJ/private/prednesy/Ifj01-cz.pdf.

[9] Meduna, A. a Lukáš, R. *Formální jazyky a překladače - Modely pro regulární jazyky* [online]. [cit. 2020-05-09]. Dostupné z: https://www.fit.vutbr.cz/study/courses/IFJ/private/prednesy/Ifj03-cz.pdf.

[10] National Instruments. FPGA Fundamentals. *Innovations*. Dostupné z: https://www.ni.com/cs-cz/innovations/white-papers/08/fpga-fundamentals.html.

[11] Paxson, V. et al. *ZEEK* [online]. [cit. 2020-04-24]. Dostupné z: https://www.zeek.org/.

[12] Pus, V., Tobola, J., Kosar, V., Kastil, J. a Korenek, J. Netbench: Framework for Evaluation of Packet Processing Algorithms. *Symposium On Architecture For Networking And Communications Systems*. Los Alamitos, CA, USA: IEEE Computer Society. 2011, s. 95–96.

[13] Roesch, M. et al. *SNORT* [online]. [cit. 2020-04-24]. Dostupné z: https://www.snort.org/.

[14] Semrič, J. *Redukce automatů používaných ve filtraci síťového provozu* [online]. Brno, CZ, 2018. [cit. 2020-04-24]. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Vojnar, T. Dostupné z: https://www.fit.vut.cz/study/thesis/21170/.

[15] Semrič, J. et al. *AHOFA* [online]. GitHub, 2018 [cit. 2020-05-03]. Dostupné z: https://github.com/jsemric/ahofa.

[16] Češka, M., Havlena, V., Holík, L., Kořenek, J., Lengál, O. et al. Deep Packet Inspection in FPGAs via Approximate Nondeterministic Automata. In: *Proceedings - 27th IEEE International Symposium on Field-Programmable Custom Computing Machines, FCCM 2019*. Institute of Electrical and Electronics Engineers, 2019, s. 109–117. DOI: 10.1109/FCCM.2019.00025. ISBN 978-1-7281-1131-5. Dostupné z: https://www.fit.vut.cz/research/publication/11951.

[17] Češka, M., Havlena, V., Holík, L., Lengál, O. a Vojnar, T. Approximate Reduction of Finite Automata for High-Speed Network Intrusion Detection. In: *Proceedings of TACAS'18*. Springer Verlag, 2018, sv. 10806, č. 2, s. 155–175. DOI: 10.1007/978-3-319-89963-3_9. ISSN 0302-9743. Dostupné z: https://www.fit.vut.cz/research/publication/11657.

[18] Češka, M. a Vojnar, T. *Teoretická informatika* [online]. [cit. 2020-05-09]. Dostupné z: http://www.fit.vutbr.cz/study/courses/TIN/public/Prednasky/tin-pr01-rj1.pdf.