VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

NÁSTROJ PRE ABSTRAKTNÝ REGULÁRNY STROMOVÝ MODEL CHECKING

TOOL FOR ABSTRACT REGULAR TREE MODEL CHECKING

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **Bc. PATRIK MRÁZ**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. MARTIN HRUŠKA**
SUPERVISOR

BRNO 2017

Abstrakt

Formálna verifikácia sa zaoberá dokazovaním korektnosti systému podľa daných špecifikácií. Jej potrebu znásobuje stále väčšia rozšírenosť počítačov a neustály rast zložitosti aj rozsiahlosti vyvíjaných systémov. Cieľom tejto práce je implementácia nástroja formálnej verifikácie abstraktný regulárny stromový model checking (ARTMC) nad knižnicou VATA. Pre dosiahnutie tohto cieľa bolo potrebné rozšíriť knižnicu VATA o konečné stromové prevodníky, abstrakcie stromových automatov a integrovať ich spolu s nástrojom ARTMC do knižnice VATA.

Abstract

Formal verification deals with proving the correctness of the system according to the given specifications. Its need is driven by an increasing number of computers and a increase in the complexity of the systems being developed. The aim of this work is to implement the formal verification tool abstract regular tree model checking (ARTMC) over the VATA library. To achieve this goal, it was necessary to extend the VATA library on the finite tree transducers, abstractions of tree automata and integrate them together with the ARTMC into the VATA library.

Kľúčové slová

abstraktný regulárny stromový model checking, ARTMC, stromový automat, stromový prevodník, abstrakcia, VATA

Keywords

abstract regular tree model checking, ARTMC, tree automata, tree transducer, abstraction, VATA

Citácia

MRÁZ, Patrik. *Nástroj pre abstraktný regulárny stromový model checking.* Brno, 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Martin Hruška

Nástroj pre abstraktný regulárny stromový model checking

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pána Ing. Martina Hrušku. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Patrik Mráz
16. mája 2017

Poďakovanie

Rád by som sa poďakoval vedúcemu práce Ing. Martinovi Hruškovi a Prof. Ing. Tomášovi Vojnarovi, Ph.D za ich cenný čas a odborné rady pri tvorbe tejto bakalárskej práce.

Obsah

Kapitola 1

Úvod

Keďže pri vývoji a údržbe systémov spravidla vznikajú chyby, ktoré môžu mať v dnešnej dobe rozšírených počítačov fatálne následky, je žiaduce ich odhaliť a odstrániť. Zaužívaný spôsob odstraňovania chýb je testovanie systému. Odhaľovanie chýb pomocou testovania je však postačujúce iba do určitej miery. Pred zahájením testovania je potrebné spísať testovacie scenáre, pri väčších systémoch však nikdy nepokryjú všetky možné prípady a navyše zamestnávať skupinu testerov je pre spoločnosť nákladné.

Niektoré problémy testovania rieši formálna verifikácia, ktorá sa zaoberá dokazovaním korektnosti systému podľa zadaných špecifikácií. Medzi prístupy formálnej verifikácie patrí model checking, statická analýza a preukazovanie teorému (*theorem proving*) [14].

V tejto práci sa ďalej budeme venovať model checkingu [4], ktorý je automatizovateľný, relatívne jednoduchý na používanie, všeobecný a poskytuje prípadný protipríklad vo forme postupnosti prechodov systému, ktoré vedú do niektorého z nekonzistentných (chybných) stavov, ktoré sú vopred špecifikované.

Regulárny model checking (skr. RMC) [3] je metóda formálnej verifikácie ktorá dokáže verifikovať parametrické systémy. K popisu konfigurácií používa množiny a pre modelovanie prechodov používa regulárne prechodové relácie. Abstraktný regulárny stromový model checking (skr. ARTMC) [2] rozširuje RMC o abstrakcie, vďaka ktorým dokáže verifikovať aj systémy s nekonečným stavovým priestorom. Uplatnenie našiel v analýze programov [8, 6]. Konfigurácie a prechody popisuje konečnými stromovými automatmi a prevodníkmi.

V súčasnosti existuje iba prototyp implementácie ARTMC v jazyku OCaml, ktorý už nie je vyvíjaný a rýchlosťou nevyhovujúci na verifikáciu väčších systémov. Cieľom práce je preto efektívna implementácia nástroja ARTMC nad knižnicou VATA [12, 13], ktorá je vyvíjaná hlavne pre účely formálnej verifikácie.

Na tomto projekte sme začali pracovať v rámci predmetov *Projektová prax 1* a *Projektová prax 2* ktoré boli absolvované v predchádzajúcom roku. V týchto predmetoch sme sa venovali štúdiu spomínaných tém a tiež implementácií časti riešenia. Pred začatím bakalárskej práce bol implementovaný prototyp reprezentácie stromových prevodníkov, výšková abstrakcia nad stromovými automatmi a prototyp ARTMC. Implementácia prototypu mala nedostatky v podobe pomalého výpočtu a implementačných chýb. Jedným z čiastočných cieľov je preto odstránenie týchto nedostatkov, refaktorizácia zdrojových kódov, začlenenie stromových prevodníkov do rozhrania príkazového riadku VATA (cli – command-line interface) a vytvorenie testov pre stromové prevodníky a samotného ARTMC. Po tomto kroku budeme správnosť implementácie experimentálne overovať na niekoľkých systémoch a výsledky porovnávať aj s predchádzajúcim riešením.

Štruktúra práce je nasledovná: v kapitole 2 rozvedená teória konečných stromových automatov a prevodníkov, v kapitole 3 sa budeme detailnejšie venovať metóde ARTMC a abstrakciám konečných stromových automatov. V kapitole 4 budeme hovoriť o knižnici VATA, ktorú budeme rozširovať a nad ktorou bude metóda ARTMC implementovaná. Následne v kapitole 5 predstavíme návrh a samotnú implementáciu ARTMC a kapitola 6 sa bude venovať experimentálnym výsledkom verifikácie pomocou implementovaného nástroja.

Kapitola 2

Konečné stromové automaty a prevodníky

V ARTMC je možné regulárne množiny a prechodové relácie reprezentovať konečnými stromovými automatmi a prevodníkmi. Môžeme ich chápať ako rozšírenie konečných automatov na stromy. Predpokladáme, že čitateľ je zoznámený s teóriou konečných automatov nad slovami.

Podobne ako pri konečných automatoch nad slovami, aj stromové automaty delíme na deterministické a nedeterministické. Ďalej sa budeme venovať iba nedeterministickým automatom a prevodníkom, pretože ich reprezentácia je oproti deterministickým stručnejšia. Konečné prevodníky (ang. transducers) nad stromami dokážu okrem prijímania stromov ich aj transformovať na iný strom. V ARTMC sa táto vlastnosť využíva na modelovanie prechodu medzi konfiguráciami systému.

V tejto kapitole bude ďalej popísaná ohodnotená abeceda, stromy a následne konečný stromový automat na prijímanie týchto stromov a konečný stromový prevodník, ktorý slúži na ich transformáciu. Definície sú prebraté z [9, 5, 10, 1].

2.1 Ohodnotená abeceda

Pre rozšírenie teórie konečných automatov nad slovami zavedieme pojem ohodnotená abeceda (ang. *ranked alphabet*). Ohodnotená abeceda je definovaná ako dvojica (F, Arity) kde F je množina symbolov a Arity je mapovanie $\mathrm{F} \to \mathbb{N}$. Číslo $n = \mathrm{Arity}(f)$, kde $f \in \mathrm{F}$ nazývame hodnosť symbolu f. Množinu symbolov hodnosti p budeme označovať ako $\mathrm{F_P}$ a množine symbolov s hodnosťou 0 budeme hovoriť konštanty (ang. *constants*).

2.2 Strom

Strom nad konečnou množinou symbolov je mapovanie $\mathrm{t} : \mathbb{N}^* \to \Sigma$, ktoré spĺňa:

- dom(t) je konečná, prefixovo uzavretá podmnožina $\mathbb{N}^*$, a
- pre každé $p \in \mathrm{dom(t)}$, $\mathrm{Arity(t}(p)) = n \geq 0$ vtedy a len vtedy, ak $\{i \mid pi \in \mathrm{dom(t)}\} = \{1,...,n\}$.

Každú sekvenciu $v \in \mathrm{dom(t)}$ nazývame uzol stromu t. Pre uzol v definujeme i-teho potomka

ako uzol vi, a i-ty podstrom t′ uzlu v tak, že t′(v′) = t(viv) pre každé p′ ∈ ℕ*. Množinu všetkých stromov nad abecedou Σ označujeme T(Σ).

2.3 Konečné automaty nad stromami

(Nedeterministický) konečný stromový automat pracujúci zdola nahor (ang. *bottom-up*) je definovaný ako štvorica A = (Q, Σ, F, σ), kde

- Q je konečná množina stavov,
- Σ je ohodnotená abeceda a
- F ⊆ Q je konečná množina koncových stavov,
- σ je množina prechodov, kde prechody sú trojica ((q1,...,qn),a,q) s dvoma prípadmi:
  1. $f(q_1,...,q_n) \rightarrow_\sigma q$ - automat načíta symbol *f* a prejde do stavu *q*, ak je prvý synovský uzol v stave q_1, ... a posledný synovský uzol v stave q_n
  2. $a \rightarrow_\sigma q$ - automat označí listový uzol *a* stavom *q*

Nedeterminizmus v prípade stromových automatov značí, že môže existovať viac pravidiel v Q ktoré majú rovnakú ľavú stranu pravidla.

2.3.1 Beh konečného stromového automatu

Beh stromového automatu je definovaný ako mapovanie π : dom(t) → Q také, že pre každé $p \in$ dom(t), kde q = π(p), ak $1 \leq i \leq n$, tak σ obsahuje pravidlo $p(q_1,...,q_n) \rightarrow q$.

Beh teda začína označením listov (symbolov stromu s hodnosťou 0) stavmi. Ak je list symbol $a \in \Sigma_0$ a existuje pravidlo $a \rightarrow_\sigma q \in \sigma$, list bude označený stavom q. Uzol so symbolom $f \in \Sigma_k$ bude označený stavom q ak existuje pravidlo $f(q_1, q_2,...,q_k) \rightarrow_\sigma q \in Q$ a prvý synovský uzol je v stave q_1, druhý v stave q_2, ..., k-tý v stave q_k. Strom je prijatý stromovým automatom ak koreňový (najvyšší) stav bude ohodnotený stavom f ∈ F. Množinu všetkých stromov, ktoré prijme stromový automat A nazývame stromový jazyk L(A). Hovoríme, že dva stromové automaty A a B sú rovnaké, ak platí L(A) = L(B).

2.3.2 Uzáverové vlastnosti

Trieda stromových jazykov je uzavretá voči určitej operácií vtedy, ak výsledok operácie na stromový jazyk je tiež stromový jazyk. Stromový jazyk nad abecedou Σ je uzavretý voči :

- Zjednoteniu : L = L1 ∪ L2 (popísaný nižšie)
- Prieniku : L = L1 ∩ L2 (popísaný nižšie)
- Doplnku : L = $\overline{L_1}$ (popísaný nižšie)
- Rozdielu : L = L_1 - L_2

Zjednotenie

Majme dva konečné stromové automaty: $A_1 = (Q_1, \Sigma, F_1, \sigma_1)$ a $A_2 = (Q_2, \Sigma, F_2, \sigma_2)$. Potom zostrojíme automat $A = (Q, \Sigma, F, \sigma)$, ktorý prijíma $L(A) = L(A_1) \cup L(A_2)$, kde $Q = Q_1 \times Q_2$, $F = F_1 \times Q_2 \cup Q_1 \times F_2$, a $\sigma = \sigma_1 \times \sigma_2$.

Doplnok

Nech L(A) je regulárny jazyk a $A = (Q, \Sigma, F, \sigma)$ je kompletný konečný stromový automat. Potom automat $A' = (Q, \Sigma, Q \backslash F, \sigma)$ prijíma doplnok množiny L v T(F).

Prienik

Uzáver voči prieniku je priamo odvodený z uzáveru voči zjednoteniu a doplnku s využitím De Morganovho zákona : $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$

2.3.3 Problémy rozhodnuteľnosti

Táto sekcia sa bude venovať problémom rozhodnuteľnosti a ich zložitosti na strojoch s náhodným prístupom do pamäti (RAM machines). Dôkazy môže čitateľ nájsť v [5].

- Problém *prázdnosti*, ktorý určuje či je jazyk prijímaný konečným stromovým automatom prázdny je rozhodnuteľný v lineárnom čase
- Problém *neprázdneho prieniku*, ktorý určuje či existuje aspoň jeden strom prijímaný každým konečným stromovým automatom z množiny zadaných automatov je rozhodnuteľný v exponenciálnom čase.
- Problém *konečnosti*, ktorý určujte či je jazyk konečného stromového automatu konečný je rozhodnuteľný v polynomiálnom čase
- Problém *ekvivalencie* určuje či dva automaty prijímajú rovnaký jazyk a je rozhodnuteľný

2.4 Stromové prevodníky

Konečný stromový prevodník pracujúci zdola nahor je definovaný ako pätica $T = (Q, \Sigma, \Sigma', F, \sigma)$, kde

- Q je konečná množina stavov,
- $F \in Q$ je množina koncových stavov,
- Σ je vstupná ohodnotená abeceda,
- Σ' je výstupná ohodnotená abeceda a
- σ je množina prechodových pravidiel, kde prechody sú trojica ((q1,...,qn),a/b,q) s dvoma prípadmi:

1. f/g (q_1,...,q_n) $\rightarrow_\sigma$ q - prevodník načíta symbol *f*, prepíše ho na symbol *g* a prejde do stavu *q*, ak je prvý synovský uzol v stave q_1, ... a posledný synovský uzol v stave q_n
2. a/b $\rightarrow_\sigma$ q - prevodník načíta symbol *a*, prepíše ho na symbol *b*, a prejde do stavu *q*

, kde $a,b \in \Sigma_0$, $f,g \in \Sigma_n$, a $q,q_1,...,q_n \in Q$.

2.4.1 Beh konečného stromového prevodníka

Beh stromového prevodníka je podobný behu stromového automatu s rozdielom, že prevodník môže zmeniť vstupný strom na iný. Ako prvé sa aplikujú pravidla typu *1.* Ak existuje pravidlo a/b $\rightarrow_\sigma$ q $\in \sigma$ a existuje list so symbolom *a*, symbol sa prepíše na symbol *b* a list sa označí stavom *q*. Ak je uzol označený symbolom *f* , existuje pravidlo 2. typu f/g (q_1,...,q_n) $\rightarrow_\sigma$ q $\in \sigma$ a prvý synovský uzol má stav q_1,..., n-tý synovský uzol má stav q_n, tak symbol *f* bude nahradený symbolom *g* a stav uzlu bude označený *q*. Beh prevodníka je úspešný, ak je koreňový uzol stromu označený stavom z množiny koncových stavov.

2.4.2 Aplikácia prevodníka na automat

Stromový prevodník je možné tiež aplikovať na stromový automat. Túto operáciu nazývame aplikácia (ang. Apply) prevodníka na automat.

Majme automat A = (Q_A, Σ, F_A, σ_A), prevodník T = (Q_T, Σ, Σ', F_T, σ_T), kde $\Sigma' = \Sigma$, a výsledný automat B = (Q_B, Σ, F_B, σ_B). Potom, ak $\exists$ a() $\rightarrow$ q $\in \sigma_A$ a zároveň $\exists$ a/b() $\rightarrow$ p $\in \sigma_T$, tak b() $\rightarrow$ (q,p) $\in \sigma_B$. Následne, ak $\exists$ f(q1,...,qn) $\rightarrow$ q $\in \sigma_A$, $\exists$ f/g(p1,...,pn) $\rightarrow$ p $\in \sigma_T$ a zároveň ((q1,p1),...,(qn,pn)) $\in \sigma_B$, tak g((q1,p1),...,(qn,pn)) $\rightarrow$ (q,p) $\in \sigma_B$.

Algoritmus by sme mohli popísať nasledovne. Pri aplikácií prevodníka na automat sa postupuje od listových pravidiel. V prvom kroku, ak existuje v prevodníku pravidlo a/b() $\rightarrow$ q a automate pravidlo a() $\rightarrow$ p, kde *a,b* sú vstupné symboly a *q*, *p* sú stavy, do výsledného automatu sa vloží pravidlo b() $\rightarrow$ (q,p) a stav (q,p). Stav (q,p) označíme ako koncový, ak je stav *q* v množine koncových stavov prevodník a stav *p* v množine koncových stavov automatu. Rovnakým spôsobom sa spracujú všetky listové pravidlá.

Následne v druhom kroku, ak existuje v prevodníku pravidlo c/d(q_1,...,q_n) $\rightarrow$ q, v automate pravidlo c(p_1,...,p_n) $\rightarrow$ p a z prvého kroku existujú stavy (q_1,p_1), ..., (q_n,p_n), do vytvoreného automatu sa vloží pravidlo d((q_1,p_1),...,(q_n,p_n)) $\rightarrow$ (q,p). Podmienka pre zaradenie stavu medzi koncové je rovnaká ako v prvom kroku.

Kapitola 3

ARTMC

Ako bolo v predošlej kapitole spomenuté, ARTMC vyjadruje Abstraktný Regulárny Stromový Model Checking [10], ktorý vychádza z Regulárneho Model Checkingu, pre popis konfigurácií systému používa automaty a výpočet akceleruje abstrakciami automatov. V tejto kapitole bude popísaný ako RMC tak aj prechod k ARTMC pridaním abstrakcie a použitím konečných automatov nad stromami namiesto automatov nad slovami. Uvedieme si tiež príklad pre RMC pre lepšie pochopenie jeho princípu.

3.1 Regulárny Model Checking

Myšlienka Regulárneho Model Checkingu (RMC) je v zakódovaní konfigurácií systému do slov vhodnej konečnej abecedy a v reprezentácií aj nekonečnej množiny týchto konfigurácií pomocou konečného automatu. Prechody medzi konfiguráciami modelujeme ako krok prechodovej funkcie daného systému, ktorý v našom príklade vyjadrujeme konečnými prevodníkmi nad slovami. Pre verifikovanie pomocou RMC je potrebné vopred poznať počiatočnú konfiguráciu, prechodovú funkciu a neprípustné konfigurácie, do ktorých by sa systém podľa špecifikácií nemal dostať.

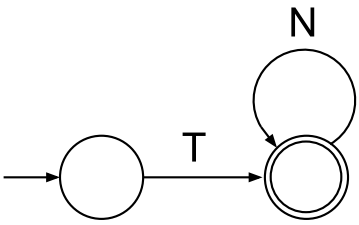
Ako prvé si zadefinujeme počiatočnú konfiguráciu, prevodník simulujúci prechody medzi konfiguráciami a množinu neprípustných stavov. Nech $\rho^*(\text{Init})$ je množina všetkých dosiahnuteľných stavov z počiatočného stavu, potom $\rho^*(\text{Init}) = \text{Init} \cup \rho(\text{Init}) \cup \rho(\rho\ \text{Init})) \cup ...$ Výsledok bude reprezentovať množinu všetkých konfigurácií, do ktorých sa systém môže dostať. Verifikácia pomocou RMC spočíva vo výpočte týchto stavov, ktoré musia mať prázdny prienik s množinou neprípustných stavov BadStates.

Platí, že systém je korektný, ak $\rho^*(\text{Init}) \cap \text{BadStates} = \emptyset$. V opačnom prípade existuje aspoň jedna neprípustná konfigurácia, do ktorej sa systém môže dostať z počiatočnej konfigurácie. Takejto konfigurácií hovoríme protipríklad (ang. *counterexample*). Postup si vysvetlíme na jednoduchom príklade *token passing protocol*.

3.1.1 RMC príklad – Token passing protocol

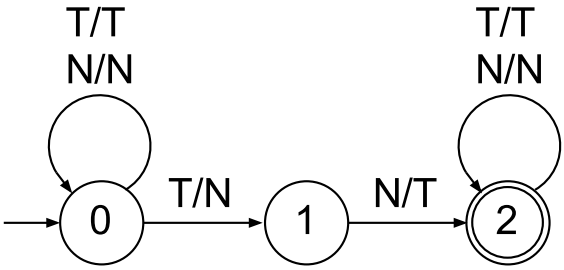
(One way) token passing protocol je systém pozostávajúci z neobmedzeného počtu procesov predávajúcich si tzv. žetón (token) v jednom smere, v našom príklade zľava doprava. Podmienkou korektnosti systému je existencia vždy práve jedného žetónu. V prípade, ak by v systéme nebol ani jeden žetón, alebo naopak, viac ako jeden, systém by sa stal nevalidný.

Z pohľadu počiatočnej konfigurácie musíme overiť, že iba proces, ktorý je najviac vľavo má žetón. Tento stav môžeme popísať nasledovným automatom *Init*, ktorý prijíma všetky korektné počiatočné konfigurácie.



Obrázok č.1 : Počiatočný automat Init

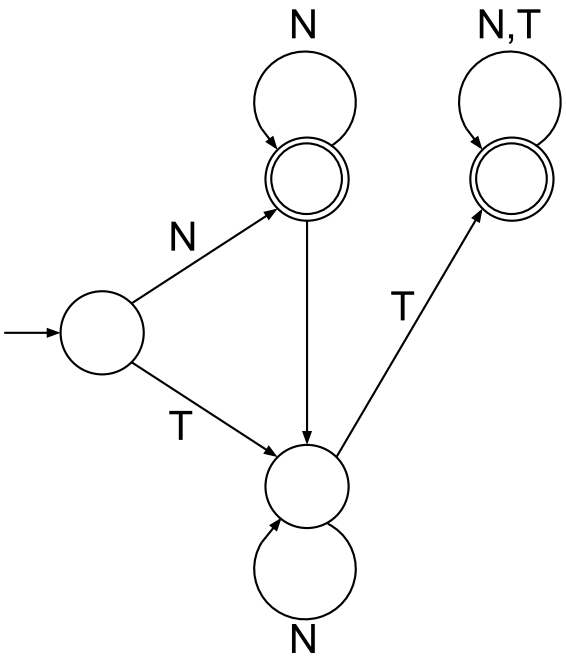
Jednotlivé stavy vyjadrujú procesy a vstupné symboly $\Sigma = \{N, T\}$ vyjadrujú, či proces má žetón (T) alebo nemá (N). Pri zmene konfigurácií musí byť žetón vždy posunutý o jednu pozíciu vpravo. Prechodovú funkciu zapíšeme pomocou konečného prevodníka *Tau*.



Obrázok č.2 : Prevodník Tau

Na obrázku prevodníka môžeme vidieť princíp jeho činnosti. Posun žetónu sa vykoná tak, že prevodník prepíše symbol T na N, teda odoberie žetón, a bezprostredne nasledujúcemu stavu žetón pridá tak, že prepíše vstupný symbol N na T. Všimnime si, že prevodník nekontroluje správnosť aktuálnej konfigurácie a prechod by prebehol aj ak by sa v systéme nachádzal viac ako jeden žetón.

Množina neprípustných stavov bude obsahovať dva prípady. V prvom prípade nebude v systéme ani jeden žetón a v druhom ich bude dva alebo viac. Keďže v RMC je žiadúce, aby sme mali práve jeden konečný automat reprezentujúci neprípustné stavy, použijeme zjednotenie automatov z prvého a druhého prípadu. Automat neprípustných stavov (*BadStates*) je znázornený nižšie.



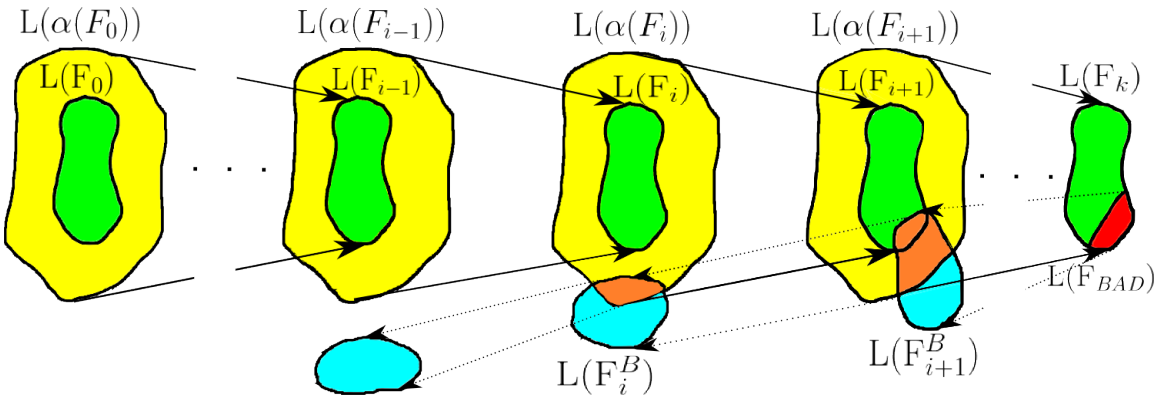
Obrázok č.3 : Automat BadStates

Automat z obrázku 3 prijíma všetky neprípustné konfigurácie systému, ktoré môžu nastať v ľubovoľnom kroku simulácie. V hlavnom cykle RMC sa vypočítajú všetky dostupné stavy z počiatočného stavu. Dosiahne sa to modelovaním prechodov, teda postupnou aplikáciou prevodníka *Tau* na automat *Init* až kým nedosiahneme stavu, kedy sa novým prechodom nezíska žiadny nový stav. Tomuto stavu hovoríme pevný bod (ang. *fixpoint*).

Výpočet $\rho^*(Init)$ prebieha vždy rozbalením cyklu o jeden krok, pokračovať bude do nekonečna a pevný bod nebude dosiahnutý. Ide o klasickú vlastnosť systémov s nekonečným stavovým priestorom. Pre zaistenie terminovania musíme použiť niektorú z akceleračných metód výpočtu. Z popisu je zrejmé, že žetón sa aktuálne posúva krok po kroku doprava. Preto je potrebné použiť abstrakciu, ktorá nadaproximovaním stavového priestoru dokáže reprezentovať nekonečný stavový priestor.

3.2 Abstraktný Regulárny Stromový Model Checking

Rámec abstraktného regulárneho stromového model checkingu [10] môžeme definovať podobne k regulárnemu model checkingu nad slovami. Z pohľadu automatov použijeme namiesto konečných automatov nad slovami konečné stromové automaty a výpočet akcelerujeme neskôr popísanými abstrakciami.



Obrázok č.4 : ARTMC - postup verifikácie

Intuitívne, verifikácia (podľa obrázku č.4) prebieha iteratívne, kde sa v každej iterácií nadaproximuje (abstrakciou) prijímaný jazyk aktuálnej konfigurácie a vykoná sa prechod medzi konfiguráciami. V prípade, ak vznikne neprázdny prienik $\mathrm{L}(F_i) \cap \mathrm{L}(F_{BAD}) \neq \emptyset$, zaháji sa spätný beh. Táto situácia nastala na obrázku pri jazyku $\mathrm{L}(F_{i+1})$. Neprázdna množina značí protipríklad a je potrebné zistiť či ide o reálny alebo falošný protipríklad. V spätnom behu aplikujeme inverznú prechodovú reláciu a prienik aktuálneho automatu s pôvodným z dopredného behu. V prípade, ak v niektorom kroku vznikne prázdny prienik medzi aktuálnym automatom a automatom z dopredného behu (rovnakej iterácie), išlo o falošný protipríklad zapríčinený hrubou abstrakciou. Výpočet opakujeme so zjemnenou abstrakciou. V prípade, ak neprázdny prienik pretrvá až po $\mathrm{L}(F_0)$, išlo o reálny protipríklad.

Uvedený postup si formalizujme. Konečný prevodník a prechodovú reláciu, ktorú reprezentuje označme $\tau(\mathrm{L})$. Nech $\iota \subseteq \mathrm{T}_\Sigma \times \mathrm{T}_\Sigma$ je relácia totožnosti a $\circ$ je zloženie relácií. Potom rekurzívne definujeme reláciu $\tau^0 = \iota$, $\tau^{i+1} = \tau \circ \tau^i$ a $\tau^* = \cup_{i=0}^{\infty} \tau^i$. Ďalej predpokladáme, že $\iota \subseteq \tau$ znamená, že $\tau^i \subseteq \tau^{i+1}$ pre každé i$\geq$0. Nech Σ je ohodnotená abeceda a $\mathbb{M}_\Sigma$ označuje množinu všetkých stromových automatov nad abecedou Σ. Funkciu abstrakcie stromových automatov potom definujeme ako mapovanie $\alpha : \mathbb{M}_\Sigma \to \mathbb{A}_\Sigma$, kde $\mathbb{A}_\Sigma \subseteq \mathbb{M}_\Sigma$ a $\forall \mathrm{M} \in \mathbb{M}_\Sigma$: $\mathrm{L}(\mathrm{M}) \subseteq \mathrm{L}(\alpha(\mathrm{M}))$.

Nech *Init* je stromový automat reprezentujúci množinu počiatočných konfigurácií a *Bad* je stromových automat definujúci množinu neprípustných stavov. Teraz môžeme iteratívne počítať sekvenciu $(\tau_\alpha^i(\mathrm{Init}))_{\mathrm{i}\geq 0}$. Pokiaľ predpokladáme, že $\iota \subseteq \tau$, z uvedeného vyplýva, že existuje také $k\geq$0, pre ktoré platí $\tau_\alpha^{k+1}(\mathrm{Init}) = \tau_\alpha^k(\mathrm{Init})$. Znamená to, že v konečnom počte krokov môžeme vypočítať nadmnožinu množiny dosiahnuteľných stavov $\tau^*(\mathrm{L}(\mathrm{Init}))$.

V prípade ak $\mathrm{L}(\tau_\alpha^k(\mathrm{Init})) \cap \mathrm{L}(\mathrm{Bad}) = \emptyset$ môžeme výsledok verifikácie, či $\tau^*(\mathrm{L}(\mathrm{Init})) \cap \mathrm{L}(\mathrm{Bad}) = \emptyset$ označiť za kladný, teda systém spĺňa dané špecifikácie. Naopak, ak je prienik neprázdny, nemusí to nutne znamenať nekorektnosť systému.

Predpokladajme, že $\mathrm{L}(\tau_\alpha^*(\mathrm{Init})) \cap \mathrm{L}(\mathrm{Bad}) \neq \emptyset$, teda existuje postupnosť krokov Init, $\tau_\alpha(\mathrm{Init})$, $\tau_\alpha^2(\mathrm{Init})$, ..., $\tau_\alpha^n(\mathrm{Init})$ taká, že $\mathrm{L}(\tau_\alpha^n(\mathrm{Init})) \cap \mathrm{L}(\mathrm{Bad}) \neq \emptyset$. Pre overenie, či daná postupnosť vedie k reálnemu alebo falošnému protipríkladu vypočítame množinu $X_n = \mathrm{L}(\tau_\alpha^n(\mathrm{Init})) \cap \mathrm{L}(\mathrm{Bad})$ a pre každé $k\geq$0, $X_k = \mathrm{L}(\tau_\alpha^k(\mathrm{Init})) \cap \tau^{-1}\ (X_{k+1})$. Môžu nastať dve situácie: (1) $X_0 = \mathrm{L}(\mathrm{Init}) \cap (\tau^{-1})^n(X_n) \neq \emptyset$, čo znamená, že výsledok verifikácie je záporný, alebo (2) existuje $k\geq$0 také, že $X_k = \emptyset$, a to znamená, že daná symbolická cesta vedie k fa-

lošnému protipríkladu, pretože α je príliš hrubá. Pre tento prípad využívame zjemňovanie abstrakcie.

3.3 Abstrakcia

Ako si môžeme na príklade všimnúť, bez akceleračnej metódy výpočet všetkých dosiahnuteľných konfigurácií nemusí terminovať. Ďalším častým problémom ktorému pri výpočte čelíme je explózia stavov, ktoré je potrebné preskúmať. V mnohých prípadoch je postačujúce namiesto výpočtu všetkých dosiahnuteľných stavov vypočítať iba ich nadmnožinu. Takýmto spôsobom môžeme úspešne verifikovať aj systémy s veľkými či nekonečnými stavovými priestormi. Nadmnožinu konečného (stromového) automatu získame jeho abstrakciou. Vo všeobecnosti ide o metódu, ktorá mapuje automat M na automat M′, v ktorom platí, že jazyk L(M′) je nadmnožina jazyka L(M) podľa pravidla daného konkrétnou metódou. Metódy abstrakcie používané v ARTMC budú vysvetlené nižšie.

3.3.1 Abstrakcia založená na stromových jazykoch konečnej výšky

Metóda abstrakcie založenej na konečnej výške bude popísaná ako prvá, pretože je jednoduchšia a priamočiara. Ako už názov napovedá, hlavná myšlienka výškovej abstrakcie spočíva v porovnávaní stromových jazykov do istej výšky. Schéma abstrakcie $\mathbb{H}_{\mathrm{n}}$ považuje dva stavy automatu M za ekvivalentné, ak ich jazyk do zadanej výšky n je rovnaký. V každom automate M existuje konečný počet stromových jazykov s maximálnou výškou n a preto má táto metóda abstrakcie konečný rozsah. Množiny ekvivalentných stavov tvoria ekvivalenčné skupiny.

Výpočet schémy abstrakcie $\mathbb{H}_{\mathrm{n}}$ je do veľkej miery podobný s výpočtom minimálneho stromového automatu s rozdielom, že každý výpočet bude ukončený po n iteráciách. Formálne, pre automat $\mathrm{M} = (\mathrm{Q}, \Sigma, \mathrm{F}, \sigma)$ definuje schéma ekvivalencie $\mathbb{H}_{\mathrm{n}}$ dva stavy za ekvivalentné $\sim_M^n$ vtedy, keď $\forall \mathrm{q}_1,\mathrm{q}_2 \in \mathrm{Q} : \mathrm{q}_1 \sim_M^n \mathrm{q}_2 \Leftrightarrow L^{\leq \mathrm{n}}(\mathrm{M}, \mathrm{q}_1) = L^{\leq \mathrm{n}}(\mathrm{M}, \mathrm{q}_2)$.

Zvyšovaním výšky n dochádza k tzv. zjemňovaniu abstrakcie, čím sa abstrahovaný automat svojim jazykom viac približuje k pôvodnému automatu. Túto vlastnosť využívame v ARTMC v prípade ak výpočet narazil na falošný protipríklad.

3.3.2 Abstrakcia založená na predikátových stromových jazykoch

Schéma abstrakcie $\mathbb{P}_{\mathcal{P}}$ v abstrakcií založenej na predikátových stromových jazykoch [10] považuje dva stavy konečných stromových automatov za ekvivalentné v prípade, že ich jazyky majú neprázdny prienik s tými istými predikátmi z množiny predikátov P. Ekvivalenčnú triedu tvorí množina stavov, ktorá má neprázdny prienik s rovnakou množinou predikátov $\mathcal{P}$.

Nech $\mathcal{P} = \mathrm{P}_1, \mathrm{P}_2, \ldots, \mathrm{P}_{\mathrm{n}}$ je konečná množina predikátov. Každý predikát $\mathrm{P} \in \mathcal{P}$ je reprezentovaný jazykom konečného stromového automatu. Nech $\mathrm{M} = (\mathrm{Q}, \Sigma, \mathrm{F}, \sigma)$ je konečný stromový automat, potom dva stavy $\mathrm{q}_1,\mathrm{q}_2 \in \mathrm{Q}$ sú v rovnakej ekvivalenčnej triede ak ich jazyky $\mathrm{L}(\mathrm{M}, \mathrm{q}_1)$ a $\mathrm{L}(\mathrm{M}, \mathrm{q}_2)$ majú neprázdny prienik s rovnakou podmnožinou predikátov z množiny $\mathcal{P}$, formálne $\forall \mathrm{P} \in \mathcal{P} : \mathrm{L}(\mathrm{P}) \cap \mathrm{L}(\mathrm{M},\mathrm{q}_1) \neq \emptyset \Leftrightarrow \mathrm{L}(\mathrm{P}) \cap \mathrm{L}(\mathrm{M},\mathrm{q}_2) \neq \emptyset$. Keďže množina predikátov $\mathcal{P}$ je konečná, aj počet všetkých podmnožín s ktorými môžu mať stavy automatu neprázdny prienik je konečný. Zjemňovanie abstrakcie prebieha pridávaním nových predikátov do množiny $\mathcal{P}$.

Kapitola 4

VATA

Pri implementácií ARTMC bude najviac operácií prebiehať na stromových automatoch a prevodníkoch a tým vzniká závislosť efektivity ARTMC od efektivity implementácie konečných stromových automatov a prevodníkov. VATA [12] je knižnica pre prácu s konečnými stromovými automatmi pod licenciou GPLv3. Je vyvíjaná skupinou VeriFIT [13] hlavne pre účely formálnej verifikácie, ale uplatnenie môže nájsť aj v iných oblastiach. Implementovaná je v jazyku C++.

VATA obsahuje vysoko optimalizované operácie [7] nad nedeterministickými konečnými automatmi nad slovami aj stromami. Súčasťou knižnice je aj rozhranie príkazového riadku cli (ang. command line interface) cez ktoré je možné s knižnicou jednoducho experimentovať.

Konečné stromové automaty na vstupe musia byť v Timbuk [11] formáte, ktorý vyzerá nasledovne:

```
Ops           a:0 b:2
Automaton     aut
States        q0 q1
Final States  q1
Transitions
a          -> q0
b(q0, q0)  -> q1
```

V prvom riadku sú po kľúčovom slove *Ops* vypísané všetky použité symboly s ich hodnosťou. Ak by bol v definícií automatu použitý ten istý symbol s dvoma rôznymi hodnosťami, interne by boli reprezentované ako dva rôzne symboly. Ďalej sa zadáva meno automatu, výpis použitých stavov, zoznam konečných stavov a prechody. Táto reprezentácia prechodov predpokladá stromový automat pracujúci zdola nahor. Prvé pravidlo `a -> q0` reprezentuje listové pravidlo v ktorom listové symboly a majú byť označené stavom `q0`. Druhé pravidlo nám hovorí, že ak má interný uzol stromu symbol `b` a oba synovské uzly sú v stave `q0`, automat prejde do stavu `q1`.

Timbuk zápis budeme využívať aj pri reprezentácií počiatočných a neprípustných konfigurácií systémov pri verifikácií pomocou ARTMC.

4.1 Možnosti kódovania automatu vo VATA

V knižnici VATA je na výber z dvoch typov kódovania automatu: Explicitné a polo-symbolické. Líšia sa hlavne v dátových štruktúrach pre ukladanie prechodov automatu.

4.1.1 Explicitné kódovanie automatu

Pomocou explicitného kódovania [7] sa stromový automat ukladá v smere zhora-nadol. Prechodové pravidlá sú uložené v hierarchickej dátovej štruktúre založenej na hashovacích tabuľkách.

Na najvyššej úrovni sa mapujú stavy automatu na takzvané prechodové klastre (ang. transition clusters), ktoré slúžia ako vyhľadávacie tabuľky a mapujú symboly vstupnej abecedy na množiny stavov. Vkladanie nových pravidiel v tomto prípade vyžaduje vždy konštantný počet krokov.

V explicitnom kódovaní je oddelená implementačná trieda stromových automatov (*ExplicitTreeAutCore*) od proxy triedy (*ExplicitTreeAut*), ktorej účelom je zjednodušiť prácu s automatmi. Cieľom je rovnakú štruktúru tried dodržať aj v prípade stromových prevodníkov.

4.1.2 Polo-symbolické kódovanie automatu

Prechodové funkcie automatu sú v polo-symbolickom kódovaní [7] ukladané v MTBDD (*multi-terminal binary decision diagrams*), čo predstavuje rozšírenie binárnych rozhodovacích diagramov. Polo-symbolické kódovanie podporuje okrem reprezentáciu stromového automatu zhora-nadol aj zdola-nahor. Je vhodné hlavne pre automaty s rozsiahlou vstupnou abecedou, ktorú v našich príkladoch nemáme a preto ho v našej práci využívať nebudeme.

4.2 Konečný automat vo VATA

Načítavanie automatu v knižnici VATA prebieha vo viacerých krokoch. V prvom kroku je reťazec s automatom načítaný a spracovaný Timbuk [11] parserom. Ak je zápis automatu syntakticky správny, preloží sa do internej reprezentácie VATA, ktorá používa vhodne zvolené dátové typy pre maximalizáciu rýchlosti výpočtov.

4.2.1 Interná reprezentácia automatu vo VATA

Pri operáciách nad automatmi ako zjednotenie, inklúzia alebo prienik je vykonané veľké množstvo elementárnych operácií medzi stavmi a symbolmi vstupných automatov. Pre minimalizáciu času potrebného pre výpočet je preto potrebné sa zamerať na optimalizáciu týchto operácií. V dnešných počítačoch sú základné operácie ako porovnanie, sčítanie alebo odčítanie najrýchlejšie nad celočíselnými dátovými typmi a preto sú použité pre reprezentáciu každého stavu a symbolu v internej reprezentácií automatu.

K mapovaniu názvu stavu (reťazca) na celočíselný typ sa používajú takzvané prekladače, ktoré umožňujú aj spätný preklad pri výpise automatu, zjednotenie ich slovníkov a podobne. Pri vstupných symboloch je postup takmer rovnaký s rozdielom, že slovník je vždy iba jeden a je zdieľaný (statický) medzi všetkými automatmi danej reprezentácie.

Stromové automaty na rozdiel od automatov nad slovami často prechádzajú z jedného stavu do n-tice stavov. Tieto množiny sú vo VATA tiež pri načítaní automatu prekladané na jeden celočíselný dátový typ. Pri operáciách s automatmi v explicitnom kódovaní je

teda porovnanie dvoch množín stavov rovnako rýchle ako porovnanie dvoch celých čísel, pretože každý stav a každá množina stavov je interne reprezentovaná celočíselným ukazovateľom. Takáto reprezentácia šetrí okrem výpočtového času aj pamäťové nároky, keďže každý stav, symbol a množina stavov bude v pamäti uložená maximálne jeden krát a môže byť referencovaná z ľubovoľného počtu automatov.

4.3 Operácie nad stromovými automatmi v explicitnom kódovaní

Knižnica VATA podporuje viacero operácií nad stromovými automatmi v oboch kódovaniach. Ich implementácia je veľmi efektívna a maximálne optimalizovaná. Pre explicitné kódovanie sú to:

- Zjednotenie
- Prienik
- Overenie inklúzie (zdola-nahor aj zhora-nadol)
- Doplnok
- Odstránenie nepoužívaných stavov
- Odstránenie nedosiahnuteľných stavov

4.4 Rozšíriteľnosť knižnice VATA

Cieľom práce je implementácia nástroja ARTMC v jazyku C++ nad knižnicou VATA s využitím konečných stromových automatov a prevodníkov. Jedným z čiastkových cieľov je implementácia konečného stromového prevodníka do VATA. Potrebné bude implementovať aj niektoré jeho operácie, ako napr. aplikovanie prevodníka na automat alebo zjednotenie dvoch prevodníkov. Vďaka modulárnej implementácií knižnice VATA je možné pridať ďalší modul vo forme stromového prevodníka, pričom niektoré časti implementácie (napr. *parser*, *serializér*) môžu byť zdieľané.

Kapitola 5

Návrh a implementácia

V tejto kapitole bude popísaný návrh a implementácia nástroja ARTMC nad knižnicou VATA. V prvej časti sa budeme venovať implementácií stromových prevodníkov do VATA, ich načítavaniu, vypisovaniu a implementácií operácií zjednotenie a aplikovanie prevodníka na stromový automat. Spomenieme tiež jej integráciu do rozhrania príkazového riadku VATA a implementované testy. Následne budú popísané implementácie abstrakcie založenej na stromových jazykoch konečnej výšky a abstrakcie založenej na predikátových stromových jazykoch. Na konci tejto kapitoli popíšeme návrh a implementáciu samotnej slučky ARTMC, jej zapúzdrenie do samostatnej triedy a spôsob použitia.

5.1 Konečné stromové prevodníky vo VATA

Knižnica VATA doteraz neimplementuje konečné stromové prevodníky (popísané v 2.4), ktoré v ARTMC využijeme ako aparát pre modelovanie regulárnych prechodových relácií v ARTMC. Zameriame sa iba na štruktúru zachovávajúce (*structure-preserving*) prevodníky, ktoré pri zmene pravidla ponechajú pôvodný počet synovských stavov a tiež hodnosť symbolu.

Zápis prevodníkov bude rovnako ako pri stromových automatoch vo formáte Timbuk s rozdielom, že namiesto symbolu použijeme dvojicu symbolov oddelených znakom "/" *vstupný_symbol*/*výstupný_symbol*. Zápis prevodníka potom bude vyzerať nasledovne:

```
Ops           a:0 c:0 b:2 d:2
Automaton     aut
States        q0 q1
Final States  q1
Transitions
a/c           -> q0
b/d(q0, q0)   -> q1
```

5.1.1 Načítanie a kódovanie

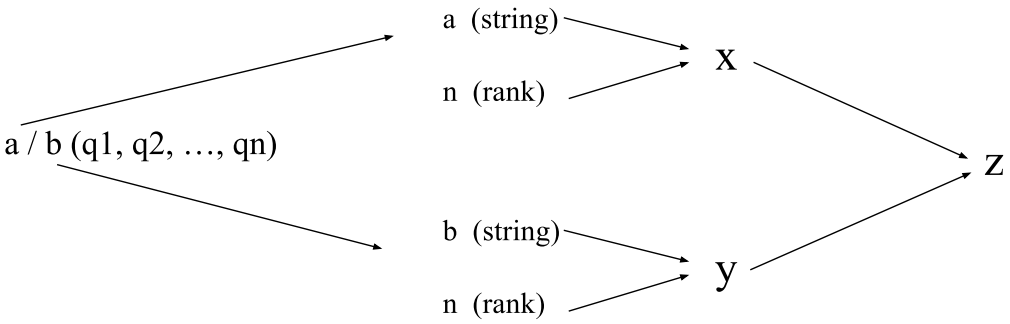
Implementovaný konečný prevodník má mať explicitné kódovanie, preto sme sa rozhodli vychádzať z existujúcej implementácie konečných stromových automatov. Boli vytvorené dve triedy pre prácu s prevodníkmi : *ExplicitTreeTrans* a *ExplicitTreeTransCore*. Kým prvá z nich poskytuje iba rozhranie pre jednoduchšiu manipuláciu, druhá z nich implementuje

jednotlivé metódy nad prevodníkmi. Načítanie prevodníka prebieha vo viacerých krokoch. Parsovanie z reťazca pomocou predaného parseru (v našom prípade Timbuk parser popísaný v sekcií 4) je zdieľané vo všetkých kódovaniach VATA. Jeho výsledkom je interný popis automatu vo forme rozparsovaných reťazcov, ktoré predstavujú jednotlivé stavy, symboly a pod.

Symboly

Keďže prevodník môže vstupný symbol prepísať na iný symbol, skladá sa z dvoch symbolov oddelených lomítkom, v tvare `a/b`, kde `a` je vstupný a `b` výstupný symbol. Pre zachovanie efektivity a rýchlosti pri výpočtoch bolo potrebné preložiť dvojicu symbolov na jednu celočíselnú hodnotu, ale zároveň rozlíšiť a vedieť porovnávať aj symboly samostatne. Je potrebné si uvedomiť, že samostatný symbol nesie aj informáciu o jeho hodnosti *arity*.

Dvojica symbol-hodnosť je vo VATA použitá v stromových automatoch a predstavuje štruktúru *StringRank*. Ako už názov napovedá, skladá sa z mena symbolu (typu reťazec) a jeho hodnosti. Preklad do internej celočíselnej reprezentácie teda prebieha v dvoch krokoch. V prvom kroku sa preloží každý symbol oddelene na celočíselnú hodnotu (na obrázku č. 4 znaky x, y) a následne sa obe hodnoty spolu preložia do jednej celočíselnej hodnoty (na obrázku č. 4 znak z), ktorá predstavuje konkrétnu dvojicu. Pre lepšiu ilustráciu je preklad zobrazený na obrázku č. 4.



Obrázok č.4 : Preklad symbolu konečného stromového prevodník do internej reprezentácie

Takýto spôsob internej reprezentácie šetrí výpočtový čas a znižuje aj nároky na pamäť, pretože každý symbol a každá dvojica symbolov bude v pamäti uložená iba jeden krát. Z pohľadu implementácie sú tieto hodnoty abecedy uložené v statickej premennej, ktoré je zdieľaná a prístupná všetkým inštanciám konkrétnej triedy. Uvedomme si, že stromové prevodníky a stromové automaty sú dve rozdielne triedy, preto je potrebné pri práci s prevodníkmi a automatmi nad rovnakou abecedou tieto abecedy zdieľať medzi sebou. Dosiahneme to najjednoduchšie metódami *getAlphabet* a *setAlphabet*.

Ak sme v programe ako prvé načítali stromové automaty, pred načítaním prevodníka je potrebné z triedy automatov získať abecedu (*getAlphabet*) a v triede prevodníkov ju vložiť (*setAlphabet*). To nám zabezpečí plné zdieľanie abecedy a korektnosť pri porovnávaní symbolov automatu so symbolmi prevodníka. Pre prácu so symbolmi prevodníkov boli doplnené

metódy *parseSymbol* a *mergeSymbol*, pomocou ktorých sú symboly prekladané do internej reprezentácie pri načítavaní a spájané dvojice symbolov pri výpise prevodníkov.

Stavy

Spôsob načítania a uloženia stavov stromového prevodníka je prebratý a totožný so stromovými automatmi. Po úspešnom parsovaní stromového prevodníka je vytvorená štruktúra AutDescription, ktorá reprezentuje jednotlivé syntaktické časti automatu po parsovaní. Z tejto štruktúry sa následne načítavajú a prekladajú jednotlive symboly, stavy a prechody do explicitného kódovania. Triedy ako *TwoWayDict* a *TranslatorStrict* sú použité pre jednoduchý preklad a uchovanie mapovania jednotlivých stavov na celočíselné interné reprezentácie.

Prechody

Po načítaní a uložení stavov a symbolov sa ukladajú prechody. Interne je prechod reprezentovaný ako trojica children-symbol-parent, kde children je množina synovských stavov, symbol je načítaný symbol vstupnej abecedy a parent je rodičovský stav. Každý element prechodu už predpokladá internú celočíselnú reprezentáciu.

5.1.2 Operácie

V tejto sekcií budú popísané implementované operácie nad konečnými stromovými prevodníkmi. Pre implementáciu ARTMC je potrebná iba aplikácia prevodníka na stromový automat, avšak pre zjednodušenie verifikácie rozsiahlejších systémov je implementované aj ich zjednotenie. To sa využíva v prípade ak je počet stromových prevodníkov simulujúcich prechody väčší ako jeden.

Zjednotenie

Návrh zjednotenia prevodníkov je prebratý z [5] a je podobný zjednoteniu stromových automatov. Majme dva jazyky L_1 a L_2 a dva stromové prevodníky $A_1 = (Q_1, \Sigma, F_1, \sigma_1)$ a $A_2 = (Q_2, \Sigma, F_2, \sigma_2)$, kde $L_1 = L(A_1)$ a $L_2 = L(A_2)$.

Nakoľko môžeme premenovať stavy prevodníka bez zmeny prijímaného jazyka, môžeme predpokladať, že $Q_1 \cap Q_1 = \emptyset$. Potom automat $A = (Q, \Sigma, F, \sigma)$ definovaný ako $Q = Q_1 \cup Q_2$, $F = F_1 \cup F_2$, a $\sigma = \sigma_1 \cup \sigma_2$ reprezentuje zjednotený automat, ktorý prijíma $L(A) = L(A_1) \cup L(A_2)$. Výsledný automat A je nedeterministický a nekompletný a to aj v prípade ak automaty A_1 a A_2 sú deterministické a kompletné.

Implementácia zjednotenia prevodníkov je rozdelená podľa internej reprezentácie stavov, či majú alebo nemajú disjunktné množiny. Prípad disjunktných množín implementuje metóda *UnionDisjointStates*, ktorá si jednoducho vytvorí kópiu jedného z prevodníkov a do nej vloží prechody a stavy (vrátane koncových) z druhého z prevodníkov. V druhom prípade, kedy môžu mať prevodníky rovnaké stavy (v internej celočíselnej reprezentácií) sa v prvom kroku nanovo preindexujú stavy prevodníkov (pričom mená ostanú pôvodné – iba interne) a do vytvoreného nového prevodníka sa vložia prechody a stavy (vrátane koncových) z oboch prevodníkov. Implementácia zjednotenia s prepisovaním stavov je metóde *Union*. Obe metódy majú zvolený názov aj postup v súlade so zachovaním notácie zo stromových automatov. Rozhodnutie, či sa použije metóda s disjuktnými množinami stavov,

alebo všeobecná metóda, ktorá stavy preindexuje je čisto v réžií programátora.

Aplikácia

Návrh aplikácie prevodníka na stromový automat vychádza z 2.4.2, kde je popísaný jeho algoritmus. Implementácia sa nachádza v triede *ExplicitTreeTransCore*, kde bola vytvorená metóda s názvom *Apply*. Táto metóda okrem vstupného prevodníka a stromového automatu voliteľne načíta aj informácie o smere aplikácie a slovníky.

Smer aplikácie (prevodníka) určuje v akom smere budú symboly prepisované. Spomeňme si, že ľavá strana prevodníka sa skladá z dvoch symbolov abecedy oddelených lomítkom a množiny synovských uzlov. V štandardnom režime sa vo vstupnom automate hľadajú pravidlá so symbolom rovnakým ako ľavý symbol z dvojice. V prípade zhody a splnenia podmienok z 2.4.2 sa do výsledného automatu zapisuje pravidlo so symbolom z pravej strany prevodníka. Tento smer je štandardný a preddefinovaný v metóde *Apply*. V prípade potreby je možné zmeniť smer aplikácie prevodníka pomocou parametra *backward*, ktorý ak je nastavený na *true*, prebehne obrátene. Túto možnosť budeme v ARTMC využívať pri protipríkladoch, aby sme zistili či ide o skutočný, alebo falošný protipríklad.

Pri aplikácií prevodníka vzniká nový stromový automat s novým slovníkom stavov *stateDictApply*. Keďže môžu existovať prípady, kedy ho užívateľ nepotrebuje, je predávaný ako voliteľný parameter, rovnako ako slovník existujúceho prevodníka *stateDictTrans* (v tabuľke č.1 prvý stĺpec) a slovník automatu, na ktorý sa prevodník aplikuje *stateDictAut* (v tabuľke č.1 druhý stĺpec). Interne sa slovník nového automatu vytvorí vždy, avšak ak bol metóde *Apply* ako ukazovateľ na slovník zadaný *nullptr*, bude zahodený. Keďže nové stavy sú tvorené ako dvojice stavov, kde prvý stav pochádza z prevodníka a druhý stav z automatu, rovnako bude aj nový stav pomenovaný do jeho slovníka (v tabuľke č.1 tretí stĺpec). Postup tvorby stavu závisí od toho, či bol alebo nebol predaný slovník prevodníka/automatu. Ak bol, použije sa tento názov, ak nie, použije sa jeho interná (celočíselná) reprezentácia prevedená na reťazec.

Pre lepšiu ilustráciu si uveďme príklad : Majme pravidlo prevodníka `a/b -> q` a pravidlo automatu `a -> p`, kde interné celočíselne reprezentácie sú `q = 3`, `p = 8`. Jednotlivé možnosti zadania slovníkov sú nasledovné:

Slovník prevodníka	Slovník automatu	Výsledný slovník
zadaný	zadaný	q_p
zadaný	nezadaný	q_8
nezadaný	zadaný	3_p
nezadaný	nezadaný	3_8

Tabuľka č.1: Možnosti zadávania slovníkov pri aplikácií prevodníka na automat

Vytváranie nových stavov z dvojice a ich preklad do internej reprezentácie je implementovaný v metóde *mergeStates*, ktorej sú predané slovníky a podľa ich (ne)existencie vytvára nové stavy konktatenovaním do vyššie uvedenej podoby.

Nová implementácia stromových prevodníkov bola tiež zaradená do CLI (*command-line interface*) rozhrania knižnice VATA, pomocou ktorého sa dajú interaktívne testovať implementované operácie. V rámci prevodníkov je CLI doplnené o operáciu *apply*, teda aplikácia prevodníka na stromový automat.

Spracovanie príkazu môžeme rozdeliť na tri kroky. V prvom sa načítajú a parsujú vstupy, ak sú korektné, v druhom kroku sa vykoná požadovaná operácia nad zadanými vstupmi a v treťom kroku sa výsledok vypíše na konzolu. Rozšírenie CLI o operáciu *apply* znamenalo úpravu parseru, aby bol schopný načítať aj stromový prevodník, pridanie parametru predstavujúceho operáciu *apply* a vytvorenie novej metódy pre vykonanie tejto operácie. Tá jednoducho načíta stromový prevodník a automat, vyvolá metódu pre aplikáciu a vypíše výsledok. Voliteľne môže vypísať i čas potrebný na jej vykonanie. Syntax je následovná :

```
./vata apply automata transducer
```

kde *vata* je názov spustiteľného súboru CLI, *apply* je meno parametru pre operáciu aplikácia, *automata* je súbor s uloženým stromových automatom a *transducer* je súbor so stromovým prevodníkom. Výsledkom bude nový automat vypísaný na konzolu.

5.2 Abstrakcia

V implementácií ARTMC sú použité dva typy abstrakcie konečných stromových automatov : abstrakcia založená na stromových jazykoch konečnej výšky a abstrakcia založená na predikátových jazykoch. Popísané sú v sekcií 3.3. Vďaka abstrakciám dokážeme verifikovať aj systémy s nekonečným stavovým priestorom, čo považujeme za jednu z najväčších predností ARTMC. V tejto sekcií si detailne popíšeme implementáciu oboch variant aj s možnosťami ich zjemňovania.

5.2.1 Abstrakcia založená na stromových jazykoch konečnej výšky

Skrátene výšková abstrakcia je implementovaná v triede *VATAAbstraction* v statickej metóde *GetHeightAbstraction*. Z popisu v sekcií 3.3.1 vieme, že je potrebné porovnávať stromové jazyky jednotlivých stavov automatu do určitej výšky. Táto výška je zadaná ako vstupný parameter metódy.

Samotný algoritmus je implementovaný iteratívne. Výsledkom každej iterácie je dátová štruktúra (*unordered_map* – hash tabuľka) ktorá mapuje stav na stav. Jeden stav automatu je namapovaný na druhý stav, ak prijímajú rovnaký jazyk. To sa overuje porovnaním prechodov, konkrétne či existuje pravidlo v ktorom prechádzajú cez rovnaký symbol a z rovnakého rodičovského uzlu. Ak áno, stavy budú mapované na do rovnakej (ekvivalenčnej) skupiny.

Uveďme si tento postup na príklade. Majme dva stavy `q1` a `q2` pri ktorých zisťujeme, či sa nachádzajú v rovnakej ekvivalenčnej triede. Ak pre stav `q1` existuje práve jedno pravidlo `a(p1,p2) -> q1`, potom ak existuje pre stav `q2` práve jedno pravidlo `a(p3,p4) -> q2`,

stavy `q1` a `q2` budú v rovnakej ekvivalenčnej triede, pretože majú pravidlá s rovnakou množinou vstupných symbolov. Pripomeňme si, že dva vstupné symboly s rovnakým názvom ale rôznou hodnosťou sa nepovažujú za rovnaké. Interne, ak sú stavy `q1` a `q2` v rovnakej ekvivalenčnej triede, tak stav `q1` sa namapuje na stav `q2` (ak predpokladáme, že stav `q2` má vyšší index).

V prípade, že je požadovaná výška vyššia ako 1, celý postup sa opakuje, s rozdielom, že sa vychádza z výsledku (mapy) predchádzajúcej iterácie.

Zo získaného mapovania stavu na stav vieme vytvoriť množinu stavov, ktoré tvoria ekvivalenčnú triedu reprezentovanú jedným stavom z tejto množiny (*reprezentant*). V našej implementácií je reprezentant stav s najvyšším indexom z danej skupiny.

Majme skupinu stavov `q1`, `q3` a `q5`, ktoré majú rovnaký stromový jazyk do výšky h. Mapovanie bude v tomto prípade vyzerať : `[q1]->[q5]`, `[q3]->[q5]`,`[q5]->[q5]`. Pri vytváraní abstrahovaného automatu sa vychádza z prechodov pôvodného automatu a mapy. Ak má pôvodný automat prechody `c(q1) -> q6` a `c(q3) -> q6`, v novom automate sa stavy `q1`, `q3` a `q5` zlúčia a vložené bude iba pravidlo `c(q5) -> q6`. Počas vkladania pravidiel sa teda nevkladajú pôvodné stavy, ale reprezentanti ekvivalenčnej skupiny v ktorej sa nachádzajú. Každý stav sa nachádza minimálne v jednej ekvivalenčnej skupine, v ktorej sa mapuje sám na seba – v tom prípade bude nový stav identický s pôvodným. Z vyššie uvedeného vyplýva, že abstrahovaný automat bude mať menej (alebo v najhoršom prípade rovnako veľa) stavov a tiež prechodov.

Zjemnenie výškovej abstrakcie dosiahneme zvýšením zadanej výšky h. Čím väčšiu výšku nastavíme, tým je výsledný automat väčší a prijímaným jazykom bližšie k pôvodnému. Po prekročení istej výšky prestane abstrakcia fungovať a výsledný automat bude identický s pôvodným.

5.2.2 Abstrakcia založená na predikátových stromových jazykoch

Predikátová abstrakcia je implementovaná v rovnakej triede VATAAbstraction v metóde GetPredicateAbstraction, ktorá si ako parametre načíta pôvodný automat na abstrakciu a množinu predikátov vo forme vektoru stromových automatov. Zo sekcie 3.3.2 vieme, že schéma abstrakcie $\mathbb{F}_{\mathrm{p}}$ považuje dva stavy za zlučiteľné, ak majú neprázdny prienik s rovnakou množinou predikátov, kde ako predikát berieme stav predikátového automatu.

Výpočet prienikov stavov je realizovaný pomocou už implementovanej metódy *IntersectionBU* (prienik zdola-nahor), ktorá okrem samotného prieniku vracia aj štruktúru mapovania stavov jedného automatu na stavy druhého automatu. Túto štruktúru využijeme pre výpočet prieniku stavov automatu s množinou predikátov.

Predikátové automaty sú uložené v dátovom type vektor (*std::vector*), pretože ich môže byť ľubovoľný počet. Cieľom je získať jednu mapu, ktorá v sebe nesie informácie o mapovaní automatu na všetky predikátové automaty. Výsledná mapa bude obsahovať mapovania typu `[q] -> [p]`, kde `q` je predikát (stav predikátového automatu) a `p` je stav abstrahovaného automatu. Následne vytvoríme novú mapu *StateToSetOfStatesMap* zoskupením všetkých predikátov (vľavo) mapovaných na rovnaký stav (vpravo), ktorá bude predstavovať ekvivalenčné skupiny. Po tomto kroku tieto skupiny jednoducho porovnáme a ak majú dva alebo viac stavov neprázdny prienik s rovnakou skupinou predikátov, v novom automate budú zlúčené.

Uvedomme si, že pre správne fungovanie predikátovej abstrakcie je potrebné zaistiť, aby mali všetky stavy naprieč všetkých stromových automatov reprezentujúcich predikáty unikátne celočíselné reprezentácie stavov. Táto podmienka štandardne nie je splnená, keďže

slovníky stavov vo VATA nie sú zdieľané a môže existovať ľubovoľný počet stromových automatov s rovnakou internou reprezentáciou stavu. Pre zaistenie unikátnosti všetkých stavov predikátových automatov vo vektore predikátov boli implementované funkcie *AddNewPredicate* a *GetLastIndex* v menovom priestore `VATA::VATAAbstraction`.

Pomocná funkcia *GetLastIndex* prijíma ako vstupný parameter vektor stromových automatov a vracia internú celočíselnú reprezentáciu stavu s najvyšším indexom. Vo funkcií *AddNewPredicate* je nový predikátový stromový automat preindexovaný (metódou *ReindexStates*) tak, aby jeho prvý index bol inkrementovaný aktuálne posledný index stavu z existujúceho vektoru predikátových automatov. Takýmto spôsobom zabezpečíme správne mapovanie stavov abstrahovaného automatu na stavy predikátových automatov vo vektore.

5.3 ARTMC

V tejto sekcií sa budeme venovať implementácií nástroja ARTMC nad knižnicou VATA s využitím našich implementácií stromového prevodníka a abstrakcií. Implementáciu ARTMC sme rozdelili na dve časti, (1) hlavný cyklus ARTMC a (2) spätný beh pre overenie protipríkladu (popísaného v 3.2).

5.3.1 Hlavný cyklus

ARTMC je implementované ako statické funkcie v menovom priestore `VATA::ARTMC`. Rozhodli sme sa vytvoriť zvlášť funkciu pre ARTMC s výškovou abstrakciou (*artmc*) a ARTMC s predikátovou abstrakciou (*artmcPredicate*). Užívateľ (resp. programátor) má tak možnosť si vybrať vhodnú metódu abstrakcie pre verifikáciu daného systému.

V rámci každej z týchto možností sú implementované dve variácie pomocou preťaženia funkcií. Prvá prijíma ako vstupné parametre počiatočný automat *Init*, automat neprípustných stavov *BadStates* a stromový prevodník *Tau* modelujúci prechody v explicitnom kódovaní (*ExplicitTreeAut*/*ExplicitTreeTrans*). Druhý variant prijíma automaty *Init*, *Bad* a prevodník *Tau* ako reťazce (*std::string*) a načítania do explicitného kódovania prebehnú v rámci preťaženej funkcie. Pri funkcii *artmc* s výškovou abstrakciou je možné zadať aj voliteľný parameter definujúci počiatočnú výšku h, v opačnom prípade bude nastavené na hodnotu 1. Obe funkcie majú návratový typ *bool*, kde *true* znamená, že systém je korektný podľa daných špecifikácií a *false* znamená, že existuje reálny protipríklad. Pseudokód implementovaného algoritmu s výškovou abstrakciou je uvedený nižšie.

Algorithm 1 ARTMC

1: $AutAlpha \leftarrow$ Automaton Init
2: $BadStates \leftarrow$ Automaton Bad
3: $Transducer \leftarrow$ Transducer Tau
4: $AutFixpoint \leftarrow$ AutAlpha
5: $AutVector \leftarrow \emptyset$ (vector of automata for backwardRun)
6: $isFixpoint \leftarrow$ FALSE
7: $h \leftarrow$ *initial height for abstraction*
8:
9: **while** !isFixpoint **do**
10: *add AutAlpha to AutVector vektor*
11: **if** $L(AutAlpha \cap BadStates) \neq \emptyset$ **then**
12: **if** backwardRun == true **then**
13: Return FALSE
14: **else**
15: $AutAlpha \leftarrow AutomatonInit$
16: $AutFixpoint \leftarrow AutomatonInit$
17: $h = h + 1$
18: **continue**
19: *make abstraction of AutAlpha with height h*
20: *apply Transducer on AutAlpha*
21: **if** $L(AutAlpha) \subseteq L(AutFixpoint)$ **then**
22: Return TRUE
23: **else**
24: $AutFixpoint = AutFixpoint \cup AutAlpha$
25: Return TRUE

V implementovanom algoritme si pred začatím samotného cyklu načítame vstupné automaty. Automat pevného bodu (*fixpoint*) a automat *Alpha*, na ktorom budú modelované prechody systému bude identický s automatom *Init*.

Na začiatku každej iterácie sa overí prienik (pomocou metódy *Intersection*) s automatom *BadStates*, ktorý reprezentuje množinu neprípustných stavov. V prípade, ak je prienik neprázdny, spustí sa *spätný beh* (bude popísaný neskôr) kvôli preskúmaniu, či ide o falošný alebo reálny protipríklad. Ak je jeho návratová hodnota *true*, ide o reálny protipríklad a verifikácia končí s negatívnym výsledkom. V opačnom prípade bol neprázdny prienik znakom hrubej abstrakcie a keďže išlo o falošný protipríklad, cyklus sa opakuje s inkrementovanou výškou pre abstrakciu (v prípade výškovej abstrakcie). Ak bol prienik s automatom *BadStates* prázdny, pokračuje sa abstrahovaním automatu *Alpha* a aplikáciou prevodníka *Tau* na automat *Alpha*.

Po tomto kroku sa overí inklúzia (metódou *CheckInclusion*) voči automatu *Fixpoint*, teda či bol po abstrakcií a modelovanom prechode preskúmaný širší stavový priestor oproti minulým iteráciám. Ak áno, automat *Alpha* sa prizjednotí (metódou *Union*) k automatu *Fixpoint* a pokračuje sa ďalšou iteráciou. Ak nie, dosiahli sme pevného bodu, stavový priestor bol preskúmaný a výsledok verifikácie je pozitívny, teda systém spĺňa dané špecifikácie.

V prípade implementácie ARTMC s predikátovou abstrakciou je postup podobný. V úvodnej časti algoritmu sa okrem vyššie spomenutých automatov vytvorí aj vektor predikátových

automatov, ktoré reprezentujú množinu predikátov. Tento vektor predikátových automatov je v prípade pozitívneho výsledku *spätného behu* (falošný protipríklad) doplnený o prienikový automat automatov *Alpha* a *BadStates*.

5.3.2 Spätný beh

Spätný beh ARTMC je implementovaný ako samostatná funkcia (*backwardRun*) v rovnakom menovom priestore ako funkcie *artmc* a *artmcPredicate*. Je volaná výlučne z hlavného cyklu ARTMC a ako vstupné parametre prijíma automat *Alpha*, prevodník *Tau*, vektor automatov *AutVector* z dopredného behu a voliteľne (ak bola použitá predikátová abstrakcia) ukazovateľ na automat (*NewPredicate*), do ktorého bude vložený nový predikátový automat. Ako automat *Alpha* je v spätnom behu zadávaný prienikový automat s automatom *BadStates*. Implementácia algoritmu bude popísaná nižšie, ako prvé si uvedieme jeho pseudokód:

Algorithm 2 backwardRun

```
1:  AutAlpha ← input automata
2:  Transducer ← input transducer
3:  AutVector ← input automata vector
4:  n ← size of vector AutVector
5:  ActualAut ← ∅
6:
7:  while n > 0 do
8:      ActualAut ← automata from AutVector at index n
9:      backward apply Transducer on AutAlpha
10:     AutAlpha ← AutAlpha ∩ ActualAut
11:     if L(AutAlpha) == ∅ then
12:         Return FALSE
13:     else
14:         h = h − 1
15:         continue
16: Return TRUE
```

Vo funkcií spätného behu používame parametre zadané z dopredného behu z momentu, kedy bol prienik automatu *Alpha* s automatom *BadStates* neprázdny. Počet iterácií, ktoré vykonáme bude rovnaký ako počet iterácií vykonaných v doprednom behu. V každej iterácií aplikujeme prevodník (z dopredného behu) v obrátenom smere a pokračujeme s automatom, ktorý dostaneme jeho prienikom s automatom z *AutVector* na príslušnom indexe. Ak dostaneme automat s prázdnym jazykom, išlo o falošný protipríklad zapríčinený abstrakciou v tomto kroku, v opačnom prípade pokračujeme. Ak neprázdny jazyk prienikového automatu pretrváva až po $n = 0$, funkcia vracia *true* čo značí, že išlo o reálny protipríklad.

Ak bol funkcii predaný aj voliteľný parameter *NewPredicate*, bude v prípade falošného protipríkladu obsahovať posledný automat *Alpha*, ktorý mal ešte neprázdny jazyk. Tento automat bude v hlavom cykle použitý ako nový predikátový automat a vložený do vektoru predikátov funkciou *AddNewPredicate*.

Kapitola 6

Experimentálne výsledky

Implementáciu nástroja ARTMC sme overili verifikáciou niekoľkých stromovo-orientovaných protokolov. Verifikačné programy v jazyku C++ obsahujú vstupné stromové automaty (*Init* a *BadStates*) a prevodník (*Tau*) vo formáte Timkuk dátového typu reťazec. V príkladoch je volaná preťažená metóda *artmc* a *artmcPredicate*, ktorej sú predané priamo reťazce.

Experimentálne výsledky boli namerané na virtuálnom počítači (s virtualizačným nástrojom VMWare) s operačným systémom *Ubuntu 16.04 LTS*. Hostiteľský počítač je osadený procesorom Intel Core i5-2500K s frekvenciou 3,3 GHz a operačnou pamäťou DDR3 8 GB.

V tejto kapitole si popíšeme jednotlivé verifikované protokoly a uvedieme si ich tiež časy potrebné na ich verifikáciu s výškovou aj predikátovou abstrakciou. Na záver spomenieme spôsob testovania implementácie stromových prevodníkov vo VATA.

6.1 Testované protokoly

Všetky nižšie uvedené protokoly boli reprezentované stromovými automatmi a sú súčasťou repozitára so zdrojovými kódmi, ktoré čitateľ môže nájsť na (https://github.com/xmrazp00/libvata).

Token Passing Protocol

Tento typ protokolu sa využíva napríklad v sieťovej komunikácií, kde si jednotlivé uzly posúvajú takzvaný žetón (*token*). Uzol, ktorý má aktuálne žetón je oprávnený využívať prenosové pásmo. V našom príklade je modelovaný *stromový* Token Passing Protocol, v ktorom sa na začiatku žetón nachádza v niektorom z listov stromu a postupne je posúvaný smerom nahor až do koreňového uzla. Verifikáciou overujeme, že žetón bude vždy práve jeden, nemôže sa stratiť ani znásobiť.

Two-Way Token Passing Protocol

Modifikovaný Token Passing Protocol, v ktorom si rovnako v stromovej štruktúre uzly posúvajú žetón, tentokrát obojsmerne, teda aj v smere od koreňového uzlu k listom. Podmienky ostávajú rovnaké.

Tree Arbiter Protocol

Protokol Tree Arbiter v stromovej štruktúre sa používa pre vzájomné vylúčenie procesorov v listoch. Každá požiadavka procesoru o vstup do kritickej sekcie sa posúva v stromovej štruktúre nahor, až kým sa nenájde uzol, ktorý toto povolenie vlastní alebo uzol, ktorý vie, kde sa uzol s povolením nachádza, pretože ho udelil niektorému zo svojich synovských uzlov. Každý uzol môže povolenie o vstup do kritickej časti posunúť nahor, alebo ho predať synovským uzlom. V rámci verifikácie overujeme vzájomné vylúčenie.

Leader Election Protocol

Leader Election protokol sa využíva pre zvolenie vodcovského procesoru. Listy s procesormi sú rozdelené na kandidátske a nekandidátske. Informácie o kandidátskych listoch sú posúvané nahor ku koreňovému uzlu. Následne, cesta zhora od koreňového uzlu smerom k listom je nedeterministicky zvolená a značí cestu k zvolenému vodcovskému procesoru. Verifikovaním overujeme, že vždy je zvolený práve jeden vodcovský procesor.

6.2 Verifikačné časy

Uvedené protokoly sme verifikovali s použitím výškovej aj predikátovej abstrakcie. Zaznamenané hodnoty vyjadrujú priemer 10-tich meraní štandardným príkazom `time`. Pri výškovej abstrakcií bola použitá počiatočná výška 1 a pri predikátovej abstrakcii je počiatočná množina predikátov prázdna. Testovanie prebiehalo v doprednom (*fw*) aj spätnom (*bw*) behu, v ktorom sú vstupné automaty *Init* a *BadStates* vymenené, teda overuje sa, či je z neprípustných stavov prechodmi možné prejsť do stavov počiatočných konfigurácií.

Protokol	$\mathbb{H}_n$ (fw)		$\mathbb{P}_\mathcal{P}$ (fw)		$\mathbb{H}_n$ (bw)		$\mathbb{P}_\mathcal{P}$ (bw)	
	t [s]	k/z	t [s]	k/z	t [s]	k/z	t [s]	k/z
Token Passing Protokol	0.009	5/0	0.006	3/0	0.010	3/0	0.008	2/0
Two Way Token Passing Protokol	0.031	5/0	0.011	1/0	0.007	1/0	0.021	1/1
Tree Arbiter	0.024	5/0	0.017	5/0	0.009	1/0	0.028	1/1
Leader Election	1.381	10/1	-	-	0.014	1/0	0.019	2/0

Tabuľka č.2: ARTMC - výsledky experimentov (t - celkový čas potrebný na verifikáciu v sekundách, k - počet vykonaných krokov pred dosiahnutím pevného bodu, z - počet zjemnení abstrakcie počas verifikácie)

Pre protokol Leader Election v doprednom behu s predikátovou abstrakciou nebol nameraný verifikačný čas, pretože bol príliš dlhý a pamäťovo náročný. Aplikácia prevodníka na automat ako nedeterministická operácia môže výrazne zvýšiť počet prechodov v novom automate. I keď predikátová abstrakcia po falošnom protipríklade od neho abstrahovala, nebolo to dostatočné k zabráneniu stavovej explózie. Tento fakt mohol byť zapríčinený aj neefektívnym zápisom konfigurácií a prechodov do stromových automatov a prevodníka, čomu nasvedčuje aj dlhšia doba potrebná pre verifikáciu pomocou výškovej abstrakcie. Na druhú stranu, v spätnom behu tento problém nenastáva a systém je verifikovaný relatívne rýchlo.

6.3 Testy stromových prevodníkov

Pri integrácií prevodníkov, konkrétne operácie *Apply*, do rozhrania CLI boli pre túto operáciu vytvorené testy. Nachádzajú sa v zložke `tests`, v podpriečinku `transducer`, ktorý obsahuje stromové automaty, prevodníky a referenčné (očakávané) výstupné automaty daného prevodníka na daný automat.

Pre testovanie stromových prevodníkov vo VATA bol vytvorený spustiteľný súbor `transducer_apply_tests.sh`, ktorý spúšťa aplikáciu prevodníka na automat pomocou CLI rozhrania nad vstupmi zo zložky *automata*.

Na výslednom automate po aplikácií a referenčnom automate sa vykoná test na jazykovú inklúziu oboma smermi. Ak je výsledok oboch inklúzií kladný, test bol úspešný. Ak je minimálne jedna inklúzia záporná, test zlyhal.

Kapitola 7

Záver

Cieľom tejto bakalárskej práce bolo rozšírenie knižnice VATA o stromové prevodníky v explicitnom kódovaní, výškovej a predikátovej abstrakcie a implementácia nástroja formálnej verifikácie ARTMC.

Po naštudovaní teoretického pozadia bol vytvorený modul pre stromové prevodníky modifikovaním modulu pre stromové automaty, pričom syntax pre jednotlivé operácie (načítanie, výpis, zjednotenie) ostala zachovaná. Pridaná bola operácia *Apply*, ktorá aplikuje stromový prevodník na automat. Táto operácia bola tiež integrovaná do *CLI* rozhrania príkazového riadku, pomocou ktorého je možné jednoducho overiť jej funkčnosť. Pre stromové prevodníky boli tiež vytvorené testy, ktoré sú súčasťou repozitára so zdrojovými kódmi.

Nad stromovými automatmi sme implementovali abstrakciu založenú na stromových jazykoch konečnej výšky a abstrakciu založenú na predikátových jazykoch. Obe abstrakcie sú umiestnené v menovom priestore knižnice VATA a boli zapuzdrené pre maximálne zjednodušenie použitia. Spôsoby zjemňovania abstrakcie pre obe metódy boli navrhnuté a otestované.

Nástroj pre abstraktný regulárny stromový model checking bol naimplementovaný nad knižnicou VATA s využitím nami implementovaných prevodníkov a abstrakcií. Bol tiež vytvorený menový priestor `VATA::ARTMC` s funkciami *artmc* a *artmcPredicate*, ktorým môže užívateľ jednoducho predať reťazce s automatmi a prevodníkom vo formáte Timbuk. Pre testovanie implementácie sme vytvorili hotové testy protokolov *(Two-Way) Token Passing Protokol*, *Leader Election* a *Tree Arbiter* s využitím oboch abstrakcií. Doby potrebné pre verifikácie sme namerali a považujeme ich za dostatočne krátke, i keď išlo o relatívne malé systémy.

Pre verifikáciu komplexnejších systémov vhodné rozšíriť implementáciu stromových prevodníkov o štruktúru nezachovávajúce (*non-structure-preserving*) prevodníky. K ďalšiemu rozvoju ARTMC by tiež prispel nástroj pre prevod zdrojových kódov do reprezentácie ARTMC, teda do formy konečných (stromových) automatov a konečných (stromových) prevodníkov. To by umožnilo verifikovať i bez rozsiahlej znalosti teórie konečných (stromových) automatov.

Literatúra

[1] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar: *Abstract Regular Tree Model Checking of Complex Dynamic Data Structures*. In Proc. of 13th International Static Analysis Symposium—SAS'06, 2006, [Online; navštívené 18.4.2017].
URL
http://www.fit.vutbr.cz/~vojnar/Publications/bhrv-artmc_point.full.pdf

[2] Ahmed Bouajjani, Peter Habermehl, Adam Rogalewicz, and Tomáš Vojnar: *Abstract regular tree model checking*. 7th International Workshop on Verification of Infinite-State Systems, 2006, [Online; navštívené 3.3.2017].
URL http://www.liafa.jussieu.fr/~haberm/Publications/bhrvinfinity05.ps.gz

[3] Bouajjani, A.; Jonsson, B.; Nilsson, M.; aj.: *Regular Model Checking*, ročník 1855. Springer, 2000, 403-418 s.
URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.31.62

[4] Clarke, E.; Grumberg, O.; Peled, D.: *Model Checking*. MIT Press, 1999, ISBN 9780262032704.
URL https://books.google.cz/books?id=Nmc4wEaLXFEC

[5] Comon, H.; Dauchet, M.; Gilleron, R.; aj.: Tree Automata Techniques and Applications. Available on: http://www.grappa.univ-lille3.fr/tata, 2007, release October, 12th 2007.

[6] L. Holik, O. Lengal, J. Simacek, A. Rogalewicz, and T. Vojnar: *Fully Automated Shape Analysis Based on Forest Automata*, ročník 8044. Saint Petersburg, Russia: Springer-Verlag, In Proc. of 25th International Conference on Computer Aided Verification, 2013, 740-755 s.
URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.31.62

[7] Ondřej Lengál, Jiří Šimáček, and Tomáš Vojnar: *VATA: A Library for Efficient Manipulation of Non-deterministic Tree Automata*. In Proc. of TACAS 2012, volume 7214, pages 79–94. Springer-Verlag, 2012, [Online; navštívené 25.3.2017].
URL https://link.springer.com/chapter/10.1007/978-3-642-28756-5_7

[8] P. Habermehl, L. Holik, J. Simacek, A. Rogalewicz, and T. Vojnar: *Forest Automata for Verification of Heap Manipulation*, ročník 41. Springer-Verlag, In Formal Methods in System Design, 2012, 83-106 s.
URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.31.62

[9] Parosh Aziz Abdulla, Yu-Fang Chen, Lukáš Holík, Richard Mayr, and Tomáš Vojnar: *When Simulation Meets Antichains: On Checking Language Inclusion of*

Nondeterministic Finite (Tree) Automata. 2010, [Online; navštívené 3.3.2017].
URL http://www.grappa.univ-lille3.fr/tata

[10] Rogalewicz, A.; Vojnar, T.; Habermehl, P.; Bouajjani, A.: *Abstract Regular (Tree) Model Checking*. International Journal on Software Tools for Technology Transfer, roč. 14, č. 2, s. 167-191, 2012, [Online; navštívené 25.3.2017].
URL http://www.springerlink.com/content/137uu7118p2054j2/

[11] Timbuk: *Webové stránky Timbuk*. [Online; navštívené 28.3.2017].
URL http://www.irisa.fr/celtique/genet/timbuk/

[12] VATA: *Webové stránky knižnice VATA*.
URL http://www.fit.vutbr.cz/research/groups/verifit/tools/libvata/.cs

[13] VeriFIT: *Webové stránky výskumnej skupiny VeriFIT*.
URL http://www.fit.vutbr.cz/research/groups/verifit/.cs

[14] Vojnar, T.: *Prednášky k predmetu FAV*. Fakulta informačných technológií, VUT v Brne, 2016, [Online; navštívené 5.5.2017].
URL http://www.fit.vutbr.cz/study/courses/FAV/public/Lectures/fav-lecture-01.pdf

Prílohy

Príloha A

Obsah CD

- `mraz_bp.pdf` - Elektronická forma textovej časti bakalárskej práce
- `sources` - Zdrojové súbory pre generovanie textovej časti bakalárskej práce v LaTeX
- `VATA` - Knižnica VATA rozšírená o náplň tejto práce