VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

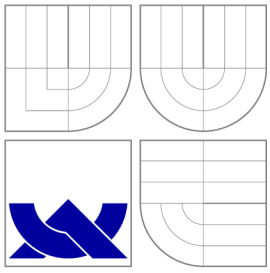
MODEL CHECKING NEKONEČNĚ STAVOVÝCH SYSTÉMŮ ZALOŽENÝ NA INFERENCI JAZYKŮ

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

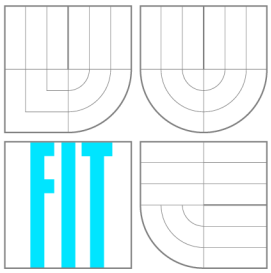
AUTOR PRÁCE
AUTHOR

Bc. PAVEL ROZEHNAL

BRNO 2007



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

MODEL CHECKING NEKONEČNĚ STAVOVÝCH SYSTÉMŮ ZALOŽENÝ NA INFERENCI JAZYKŮ

MODEL CHECKING INFINITE STATE SYSTEMS BASED ON INFERENCE LANGUAGES

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. PAVEL ROZEHNAL

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. TOMÁŠ VOJNAR, Ph.D.

BRNO 2007

Abstrakt

Regulární model checking je metoda pro verfikaci nekonečně stavových systémů. Je založena na kódování jejich konfigurace jako slov nad konečnou abecedou, množiny konfigurací jako konečného automatu a přechodů jako konečných transducerů. Je zde představen nový přístup k regulárnímu model checkingu založený na odvozování regulárních jazyků. Metoda je založena na prozkoumávání nekonečně stavového systému, jehož chování může být modelováno použitím transducerů, které zachovávají délku řetězců a jejich aplikací je možné získat všechny dosažitelné konfigurace systému. Naše metoda regulárního model checkingu je založena na odvozování regulárních jazyků pomocí algoritmu Angluin, který je použit pro nalezení vhodného invariantu (nadaproximace), který je schopen zodpovedět otázku zachování či porušení nějaké vlastnosti. Je zde také uveden úvod do teorie konečných automatů, model checkingu, SAT problémů a popis Angluinova a Biermanova algoritmu pro učení konečných automatů.

Klíčová slova

Konečný automat, Angluin L^* algoritmus, Biermanův algoritmus, model checking, regulární model checking, formální verifikace

Abstract

Regular model checking is a method for verifying infinite-state systems based on coding their configurations as words over a finite alphabet, sets of configurations as finite automata, and transitions as finite transducers. We implement regular model checking using inference of regular languages. The method builds upon the observations that for infinite-state systems whose behavior can be modeled using length-preserving transducers, there is a finite computation for obtaining all reachable configurations. Our new approach to regular model checking via inference of regular languages is based on the Angluin's L^* algorithm that is used for finding out an invariant which can answer our question whether the system satisfies some property. We also provide an intro to the theory of finite automata, model checking, SAT solving and Anguin's L* and Bierman algorithm of learning finite automata.

Keywords

Finite automate, Angluin's L^* algorithm, Bierman algorithm, model checking, regular model checking, formal verification

Citace

Pavel Rozehnal: Model checking nekonečně stavových
systémů založený na inferenci jazyků, diplomová práce, Brno, FIT VUT v Brně, 2007

Model checking nekonečně stavových systémů založený na inferenci jazyků

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Tomáše Vojnara. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Pavel Rozehnal
21. května 2007

Model checking nekonečně stavových systémů založený na inferenci jazyků

Model checking infinite state systems based on inference languages

Vedoucí:
Vojnar Tomáš, Ing., Ph.D., UITS FIT VUT

Oponent:
Křena Bohuslav, Ing., Ph.D., UITS FIT VUT

Přihlášen:
Rozehnal Pavel, Bc.

Zadání:

1. Seznamte se s teorií regulárních jazyků a konečných automatů a s možnostmi jejich využití k formální verifikaci v rámci tzv. regulárního model checkingu.
2. Seznamte s různými metodami inference regulárních jazyků (Angluinova metoda, Biermanova metoda apod.) a s doposud navrženými možnostmi jejich využití pro regulární model checking.
3. Seznamte se s knihovnou FSA pro práci s konečnými automaty v Prologu, s různými systémy pro SAT solving a prostudujte možnosti jejich propojení s FSA za účelem využití v další části projektu.
4. Dle pokynů vedoucího naimplementujte algoritmus regulárního model checkingu založený na kombinaci Angluinovy a Biermanovy metody inference jazyků.
5. Otestujte implementaci na vhodných případových studiích a zhodnoťte její chování.

Část požadovaná pro obhajobu SP:
První tři body zadání.

Kategorie:
Formální verifikace

Implementační jazyk:
Prolog, C

Literatura:
Dle pokynů vedoucího.

Komentář:
Zadání je řešeno ve spolupráci s P. Habermehlem z LIAFA, Université Paris 7 a M. Leuckerem z TU Mnichov.

Licenční smlouva

Licenční smlouva je uložena v archivu fakulty informačních technologií Vysokého učení technického v Brně.

Obsah

Seznam použitých zkratek a symbolů

FA	konečný automat (Finite-state Automata)
DFA	deterministický konečný automat (Deterministic Finite-state Automata)
MC	model checking, kontrola modelů
RMC	regulární model checking, regulární kontrola modelů
$\mathcal{O}$	funkce, přiřazuje vzorkům symbol indikující příslušnost řetězce do jazyka, Biermanův algoritmus
Σ	abeceda, množina symbolů konečného automatu nebo transduceru
$\mathcal{OT}$	tabulka pozorování (Observation Table), Angluin $\mathcal{L}^*$ algoritmus
S_u	proměnná vzorku, Biermanův algoritmus
S_u^j	logický literál (atomická boolean hodnota) přiřazená j-tému stavu v DFA a u-tému vzorku z O, Biermanův algoritmus
$\mathcal{E}$	množina experimentů, Angluin $\mathcal{L}^*$ algoritmus
$\mathcal{S}$	množina řetězců reprezentující různé stavy výsledného DSA, Angluin $\mathcal{L}^*$ algoritmus
trs	model chování systému, reprezentace transducerem
bad	kontrolovaná vlastnost systému, reprezentace konečným automatem
$init$	počáteční stav systému, reprezentace konečným automatem

Kapitola 1

Úvod

V současnosti je software nedílnou částí různých systémů námi používaných v každodenním životě. Často u takových zařízení předpokládáme absolutní bezchybnost softwarové části a svěřujeme jim stále kritičtější funkce, jako je vybavení nemocnic nebo většina automatických řídících systémů (např. brzdové systémy v automobilech). Krom toho jsou techniky softwarového (SW) vývoje stále častěji aplikovány také při programování hardwaru či přímo můžeme hardware programovat[1].

Musíme předpokládat, že všechny systémy obsahují chyby a softwarová výbava není výjimkou - dokonce může potenciálně obsahovat více chyb (či zranitelností) než ostatní druhy systémů (v souladu s komplexností sofwarových celků a relativní novosti softwaru jako technologie) a je nutné mít nějaké automatizované metody pro detekci programových chyb. Historie nás už mohla poučit o závažnosti takových softwarových chyb. Připomeňme například známou chybu v operaci dělení, která se vyskytla u prvních verzí procesorů Intel® Pentium® nebo velké množství chyb v operačních systémech, kde nové chyby jsou objevovány téměř každým dnem a tak dovolují existenci velkého počtu počítačových virů.

Jednou z přirozených cest pro detekci chyb je testování, hledání význačných vstupních dat a ověřování správnosti výstupů (odezvy systému). V případě jednoduchých systémů jsme schopni touto metodou otestovat veškeré možné kombinace a hodnoty vstupů, ale takových systémů v reálném životě nenalezneme mnoho. Většinou potřebujeme ověřit systémy, které jsou velmi rozlehlé (např. části operačních systémů) nebo obtížně simulovatelné (např. paralelní systémy), u kterých je velmi obtížné či dokonce nemožné[2] otestovat veškeré možnosti bez opomenutí některé z nich. Druhou věcí je, že testováním nejsme schopni dokázat, že v systému nejsou chyby[3], ale právě opačně, že v systému nějaké jsou, tzn. testováním nejsme schopni systém verifikovat, tzn. prokázat jeho bezchybnost.

Jiným přístupem je formální verifikace. Jednou z její forem je metodika nazvaná „model checking (MC)“, která se zabývá kontrolou konečně stavových systémů (finite-state). Model checking je kontrola systému (přesněji jeho modelu) zda zachovává nějakou testovanou vlastnost. U konečně stavových systémů je provedeno (na úrovni modelu) prozkoumání všech dosažitelných stavů (nebo pro zefektivnění, na základě heuristicky vybraných stavů) a u každého z nich ověřena kontrolovaná vlastnost.

Novější variantou MC je *regulární model checking (Regular Model Checking, RMC)*, která umožňuje kontrolovat nekonečně stavové systémy. Toto poměrně nové zobecnění je

[1] Například VHDL, v současnosti nejvíce používaný jazyk pro design hardwarových systémů.

[2] Z časových, finančních či jiných důvodů.

[3] U skutečně jednoduchých systémů, kde jsme schopni dokázat, že byly prověřeny veškeré možné kombinace vstupních hodnot toto tvrzení neplatí.

motivováno v současnosti rostoucí důležitostí nekonečně stavových systémů. Mezi takové systémy patří zásobníkové automaty, (ztrátové, lossy) FIFO-kanály, systémy s čítači nebo parametrizované a dynamické sítě procesů. Jelikož regulární model checking nemůže prozkoumávat veškeré dosažitelné stavy systému (stavový prostor není konečný), musí s nekonečnou množinou stavů pracovat symbolicky za pomoci různých formalismů jako jsou logiky nebo automaty. S veškerými stavy je poté manipulováno současně.

V diplomové práci je představena nová metoda RMC založená na novém použití algoritmů Angluin a Bierman (algoritmy pro učení konečných automatů ze vzorků jazyka). V druhé kapitole je proveden detailní úvod do celkové problematiky RM a veškerých souvisejících technik. Třetí kapitola obsahuje rozbor nového přístupu k RMC a v následujících kapitolách je rozebrána implementace a výsledky experimentů.

Diplomová práce navazuje na semestrální projekt, ve kterém byl proveden rozbor problematiky a teoreticky rozebrána nová metoda RMC.

Kapitola 2

Prerekvizity

V prvních kapitolách je rozvedena problematika model checkingu konečně stavových systémů, ve které bych zdůraznil konečný automat (kapitola 2.1.1), prostředek pro modelování konfigurací systémů jako slov příjímaných konečným automatem, které budeme i my používat pro modelování konfigurací systémů.

Dále je rozvedeno rozšíření konečně stavového model checkingu na model checking nekonečně stavových systémů, který je předmětem této práce, a představeny pojmy jako regulární modely nebo regulární relace a transducery (kapitola 2.2.1).

A v posledních kapitolách je představen úvod do problematiky SAT problémů a možnosti jeho řešení následované představením základních variant algoritmů pro učení konečných automatů použitých naší metodou formální verifikace.

2.1 Model checking

Kapitola představuje úvod do metodik a pojmů z oblasti model checkingu (MC), mezi které patří pojem modelu, konfigurace, chování systému a další.

2.1.1 Model

Modelem je označována reprezentace dynamického systému (a jeho chování) za použití matematického popisu založeném na množinách stavů a relaci přechodů. Reprezentace stavu reálného systému je nazývána jeho *konfigurací*. Konfigurací můžeme označovat například hodnoty nějakých programových proměnných nebo obsah nějakého síťového spojení. Je možné nalézt mnoho různých reprezentací konfigurací pro jeden sysém, které ale mohou pouze reprezentovat různé pohledy na zkoumaný systém. Důvod existence mnoha modelů jednoho systému může být dán tím, že na systém je možné pohlížet z pohledu různých vlastností či různé abstrakce. Pak záleží jen na našich požadavcích, kterou reprezentaci vybereme a která nejlépe charakterizuje analyzované chování.

Konečný automat (FA)

Základním modelem pro modelování systémů je konečný automat [20, 9, 18], což je pětice $M = (Q, \Sigma, \delta, q_O, F)$, kde Q je množina konečných stavů, Σ konečná abeceda, $\delta \subseteq Q \times \Sigma \times Q$ množina přechodů, $q_0 \in Q$ označuje počáteční stav a $F \subseteq Q$ množinu koncových stavů. M je označován jako deterministický, když $\forall q \in Q, a \in \Sigma$ existuje nejvíce jeden přechod q' s $(q, a, q') \in \delta$.

Relace přechodu (transition relation) $\rightarrow \subseteq Q \times \Sigma^* \times Q$ z M je definována jako nejmenší relace zachovávající:

1. $\forall q \in Q : q \xrightarrow{\epsilon} q$,
2. když $(q, a, q') \in \delta$, pak $q \xrightarrow{a} q'$
3. když $q \xrightarrow{w} q'$ a $q' \xrightarrow{a} q''$, pak $q \xrightarrow{wa} q''$.

Jazyk rozpoznávaný (generovaný, přijímaný) automatem M ze stavu $q \in Q$ je definován jako $L(M, q) = \left\{\exists q' \in F : q \xrightarrow{w} q'\right\}$. Jazyk $L(M)$ je ekvivalentní $L(M, q_0)$. Množina $L \subseteq \Sigma^*$ je regulární, když existuje konečný automat M takový, že platí $L = L(M)$. Také lze definovat jazyk slov do nějaké konkrétní délky $L^{\leq n} = \{w \in L || w| \leq n\}, L^{\leq n}(M, q) = \{w \in L(M, q) || w| \leq n\}$, a $\Sigma^{\leq n} = \{w \in \Sigma^* || w| \leq n\}$.

Definujme hloubku automatu $M = (Q, \Sigma, \delta, q_0, F)$ označovánu d_M jako maximální délku nejkratší cesty vedoucí do některých stavů M z počátečního stavu q_0, např. $d_M = max_{q \in Q} min_{w \in \Sigma^* \wedge q_0 \xrightarrow{w} q} |w|$.

Říkáme, že $q, q' \in Q$ automatu M jsou k-nerozlišitelné, značeno $q \equiv_k q'$, když $L^{\leq k}(M, q) = L^{\leq k}(M, q')$. Pak lze definovat stupeň rozlišitelnosti ρ_M automatu M jako minimální k takové, že každé dva stavy q, q' of M jsou k-rozlišitelné, např. $q \not\equiv_k q'$.

Büchiho automat

Dalším známým modelem pro popis systémů pro účely model checkingu jsou Büchiho automaty, rozšíření běžných konečných automatů, představené švýcarským matematikem Juliusem Richardem Büchim.

Definice 1 ***(Büchiho automat)*** *Büchiho automat M je pětice $(\Sigma, Q, \delta, q_0, F)$, kde*

- *Σ je konečná množina akcí nebo znaků*
- *Q je konečná množina stavů*
- *$\delta : Q \times \Sigma \rightarrow 2^Q$ je přechodová funkce*
- *$q_0 \in Q$ je počátečním stavem a*
- *$F \subseteq Q$ je množina koncových stavů*

Každý běh začíná počátečním stavem a pak je v každém kroku nedeterministicky vybírán následník aktuálního stavu.

Definice 2 ***(Nekonečný běh)*** *Nechť je Büchiho automat $M = (\Sigma, Q, \delta, q_0, F)$.* Nekonečný běh γ *automatu M přes slovo $w = a_0 a_1 \ldots \in \Sigma^w$ je nějaký běh $\gamma = \gamma_0 \gamma_1 \ldots \in Q^w$ takový, že $\gamma_0 = q_0$ a $\gamma_i \xrightarrow{a_i} \gamma_{i+1}$, pro všechna $i \geq 0$.*

Více je možné najít v [4].

Kripkeho struktura (Kripke structure, KS)

Dalším často používaným modelem jsou Kripkeho struktury, kde uzly označují atomické výroky a značené přechody (labeled transition) označují akce.

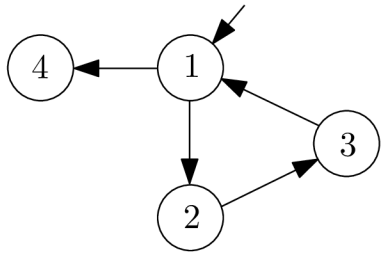
Definice 3 ***Kripkeho struktura*** *nad množinou AP atomických výroků je trojice (S, R, I), kde*

- *S je množina stavů*
- *$R \subseteq S \times S$ je relace přechodů*
- *$I : S \to 2^{AP}$ je funkce značení (labeling function)*

Každý výrok z množiny AP popisuje lokální vlastnosti stavu systému. Ke každému stavu systému je pomocí funkce značení (labelign function) $I : S \to 2^{AP}$ přiřazen atomický výrok, funkce I přiřazuje stavu jeho platné výroky. Funkce značení je nazývána interpretací.

Kripkeho struktura se nazývá *totální (total)*, když R je totální relace. V ostatních případech se nazývá *částečná (partial)*. Kripkeho struktura je nazývána *kořenová (rooted)*, když jeden stav $s_0 \in S$, je označen jako počáteční. Pro použití v model checkingu jsou množiny S a AP obvykle konečné.

Na obrázku 2.1 je znázorněn příklad jednoduché Kripkeho struktury, která reprezentuje zjednodušený životní cyklus procesu v operačním systému.



Obrázek 2.1: Příklad Kripkeho struktury

Definice KS předchozího obrázku je

$AP = \{$běhuschopný, přerušený, běžící, ukončený$\}$

$S = \{1, 2, 3, 4\}$

$R = (1, 2), (1, 4), (2, 3), (3, 1)$

$I : \{I(1) = \{$běhuschopný, běžící$\}, I(2) = \{$přerušený$\}, I(3) = \{$běhuschopný, přerušený$\}, I(4) = \{$ukončený$\}\}$

Příklad nějakého běhu nad KS z příkladu je: $1, 2, 3, 1, 4$.

Temporální logiky (Temporal Logics, TL)

Účelem modelování je popis systémů, tzn. popis jejich chování mezi stavy a vlastností stavů. Je ovšem nutné takové popisy formalizovat. Příkladem může být ověření dosažitelnosti, která může být volně popsána následovně: „Je možné dosáhnout nějakého stavu, který je

nějakých vlastností a dosažitelný z počátečního stavu?“. *Temporální logiky[4, 2]* jsou logické formalismy vytvořené pro vyjádření takových podmínek.

Základ TL je tvořen skutečností, že výrok nemusí být pouze staticky pravdivý nebo nepravdivý, ale jeho pravdivost se může měnit s časem. Pravdivost formule v temporální logice je posuzována vždy k jednomu stavu systému v jednom časovém okamžiku, tzn. stav systémů se může měnit s časem.

Existují dva typy TL - linear-time a branching-time. První z nich (linear-time) je blíže k přirozenému pohledu na čas, tzn. představujeme si jej jako lineárně uspořádanou množinu. Pokud vezmeme dva stavy s a t, tak z časového hlediska platí $s < t$, $s = t$ nebo $t < s$, kde operátor $<$ představuje časovou následnost. V branching-time TL ma čas stromovou strukturu, tzn. každá minulost nějakého stavu je jednoznačná a jeho budoucnost je nedeterministicky větvena. Formule tedy mohou v různých větvích budoucího vývoje nabývat různých hodnot.

Lineární temporální logiky (Linear Temporal Logic, LTL)

Lineární temporální logiky si lze představit jako standardní linear-time TL, které jsou často prezentovány ve spojení s Kripkeho strukturami. Formule mohou být konstruovány následovně:

$$\phi ::= \top \mid p \mid \neg\phi \mid \phi \wedge \phi \mid X(\phi) \mid \phi U \phi, \ kde\ p \in AP$$

poznámka: symbol $\top$ označuje tautulogii (atomický výrok, který je pravdivý nad všemi stavy).

Sémantika $[[\phi]]$ formule ϕ je množina všech běhů γ pro které je vlastnost zachována: $[[\phi]] = \{\gamma | \gamma \models \phi\}$. Sémantiky jsou induktivně definovány na struktuře formule následovně

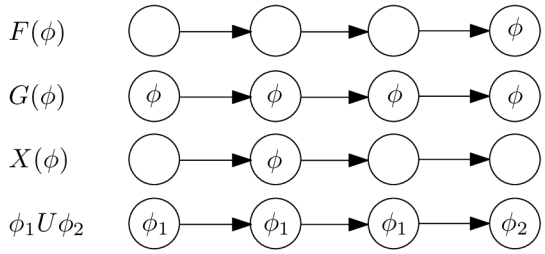
$$\begin{array}{lcl}
\gamma \models \top & & \\
\gamma \models p & \Leftrightarrow & p \in I(\gamma_0) \\
\gamma \models \neg\phi & \Leftrightarrow & \gamma \not\models \phi \\
\gamma \models \phi_1 \wedge \phi_2 & \Leftrightarrow & \gamma \models \phi_1 \wedge \gamma \models \phi_2 \\
\gamma \models X(\phi) & \Leftrightarrow & |\phi| > 1 \wedge \gamma^1 \models \phi \\
\gamma \models \phi_1 U \phi_2 & \Leftrightarrow & \exists k \in [|\gamma| - 1] : (\gamma^k \models \phi_2 \wedge \forall i \in [k-1] : \phi^1 \models \phi_1)
\end{array}$$

Každý běh γ je tautologií ($\top$) a každý běh γ zachovává nějaký atomický výrok, když první stav γ_0 zachovává. Následující negace a konjunkce jsou interpretovány obvyklým způsobem.

LTL také zavádí modální operátory X, který je interpretován jako „následně“, tzn. vyžaduje platnost vlastnosti ϕ v nějakém následujícím stavu a binární operátor $\phi_1 U \phi_2$ je interpretován „ϕ_1 dokud neplatí ϕ_2“. Formule platí, pokud existuje stav na cestě, kde je splněna vlastnost ϕ_2 a v každém předcházejícím stavu je splněna vlastnost ϕ_1.

Nad uvedenými základními operátory jsou definovány další dva, které jsou pouze přepisem uvedených, ale s výhodou použitelné.

$$\begin{array}{lcl}
F(\phi) & := & U(\top, \phi) \\
G(\phi) & := & \neg F(\neg\phi)
\end{array}$$



Obrázek 2.2: Příklad grafické interpretace operátorů LTL

Unární operátor $F(\phi)$ (interpretová jako „někdy v budoucnu platí“) se používá k určení, že vlastnost ϕ bude splněna v některém z následujících stavů a operátor $G(\phi)$ (interpretován jako „vždy“ nebo „globálně“) specifikuje, že vlastnost ϕ je splněna v každém následujícím stavu.

Temporální logika s větvením času (Computational Tree Logic, CTL)

CTL logika je jedna z prvních temporálních logik s větvením času. Vychází z LTL logiky kterou rozšiřuje o možnost větvení a zavádí existenční ($\exists$) a obecný ($\forall$) kvantifikátor cesty. Tyto kvantifikátory jsou užity vždy v konkrétním stavu k určení, zda všechny cesty nebo některá z cest vycházející z daného stavu splňuje danou vlastnost.

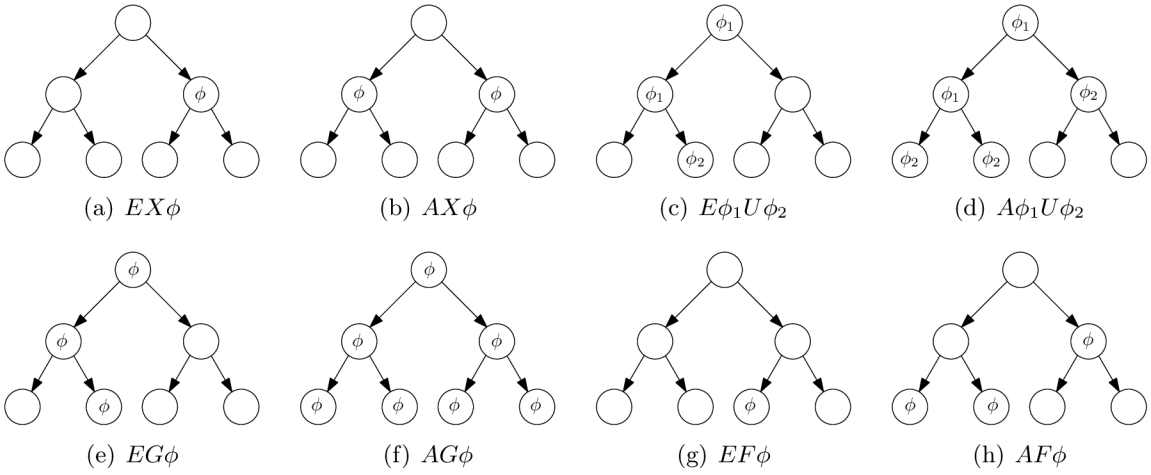
$$\phi ::= \top \mid p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid EX(\phi) \mid E\phi_1 U \phi_2 \mid A\phi_1 U \phi_2$$

Sémantika formulí označující podmnožinu stavů $s \in S$ (S je množina stavů Kripkeho struktury), pro které je formule splněna: $[[\phi]]^T = \{s \in S | s \models \phi\}$. Nechť Π_s^* označuje množinu všech běhů ze stavu s. Sémantiky jsou induktivně definovány na struktuře formule následovně

$$\begin{array}{lcl}
s \models \top & & \\
s \models p & \Leftrightarrow & p \in I(s) \\
s \models \neg\phi & \Leftrightarrow & s \not\models \phi \\
s \models \phi_1 \wedge \phi_2 & \Leftrightarrow & s \models \phi_1 \wedge s \models \phi_2 \\
s \models EX(\phi) & \Leftrightarrow & \exists s' \in S : (s, s') \in R \wedge s' \models \phi \\
s \models E\phi_1 U \phi_2 & \Leftrightarrow & \exists \gamma \in \Pi_s^* : \exists k \in [[\gamma]] : (\gamma^k \models \phi_2 \wedge \forall i \in [k-1] : \gamma^i \models \phi_1 \\
s \models A\phi_1 U \phi_2 & \Leftrightarrow & \forall \gamma \in \Pi_s^* : \exists k \in [[\gamma]] : (\gamma^k \models \phi_2 \wedge \forall i \in [k-1] : \gamma^i \models \phi_1
\end{array}$$

Každý stav s je tautologií ($\top$) a každý stav s splňuje výrok, když je výrok přiřazen stavu. Negace a konjunkce jsou definovány obvyklým způsobem. Modalita EX („následně ϕ“) intuitivně znamená, že existuje bezprostřední následník stav s' dosažitelný vykonáním jednoho přechodu, který splňuje ϕ. Modalita $E\phi_1 U \phi_2$ („existuje ϕ_1 dokud ϕ_2“) vyžaduje existenci takového běhu γ začínajícího ve stavu s, jež má prefix, jehož poslední stav splňuje ϕ_2 a veškeré ostatní stavy prefixu splňují ϕ_1. Modalita $A\phi_1 U \phi_2$ („všechna ϕ_1 dokud ϕ_2“) vyžaduje, aby veškeré běhy γ začínajícího ve stavu s a měly prefix, jehož poslední stav splňuje ϕ_2 a veškeré ostatní stavy prefixu splňují ϕ_1.

Nad uvedenými základními operátory jsou definovány další, které jsou pouze přepisem uvedených, ale s výhodou použitelné.



(a) $EX\phi$ (b) $AX\phi$ (c) $E\phi_1 U\phi_2$ (d) $A\phi_1 U\phi_2$

(e) $EG\phi$ (f) $AG\phi$ (g) $EF\phi$ (h) $AF\phi$

Obrázek 2.3: Grafická interpretace operátorů CTL

$$\begin{aligned} AX(\phi) &:= \neg EX(\neg\phi) \\ AF(\phi) &:= A\top U\phi \\ EF(\phi) &:= E\top U\phi \\ AG(\phi) &:= \neg EF(\neg\phi) \\ EG(\phi) &:= \neg AF(\neg\phi) \end{aligned}$$

Modalita $AX(\phi)$ („všechna následující ϕ“) vyžaduje, aby ϕ byla zachována ve všech následnících aktuálního stavu s. Modalita $AF(\phi)$ („někdy v budoucnu platí“) vyžaduje, aby ϕ bylo zachováno v některém stavu všech možných následujících běhů z aktuálního stavu. $EF(\phi)$ je existenční variantou předchozího - musí existovat nějaký běh ze současného stavu, který zachovává ϕ. $AG(\phi)$ („globálně pro všechny“) vyžaduje, aby ϕ bylo splněno ve všech stavech a ve všech možných cestách ze současného stavu. Poslední modalita $EG(\phi)$ („globálně existuje“) vyjadřuje, že musí existovat cesta ze současného stavu, který zachovává ϕ ve všech svých stavech.

2.1.2 Popis systému

K popisu systému, modelování, s používá Büchiho nebo Kripkeho automat, kde množina všech akceptujících stavů je množinou konfigurací. Takto jsou všechny běhy systému (modelu) akceptujícími běhy.

Obvykle je vybrána n-tice proměnných $V = (x_1, x_3, \ldots, x_n)$, společně s jejich doménami $D = (D_1, D_2, \ldots, D_n)$. Množina konfigurací Γ je pak $D_1 \times D_2 \times \ldots \times D_n$, podmnožinou kartézského součiny všech domén. Pro popis množiny konfigurací se používá predikátů nad proměnnými, například $x_3 = 5$ určuje množinu $D_1 \times D_2 \times \{5\} \times D_4 \times \ldots \times D_n$.

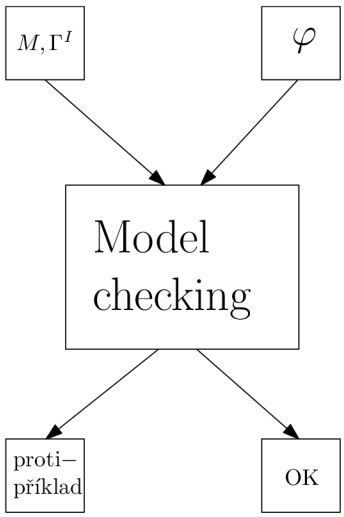
2.1.3 Model Checking (MC)

Klasický model checking lze aplikovat pouze na konečně stavové systémy (přesněji jejich modely). Hlavní využití této metody je při verifikaci hardwaru, jehož stavový prostor je ze své podstaty vždy konečný. Zjednodušeně lze říci, že se jedná o prohledávání stavového prostoru stav po stavu a u každého se kontroluje zachování nějaké vlastnosti. Rozšíření na nekonečně stavové systémy bude probráno v kapitole 2.2.

Je dána množina konfigurací Γ, model M nad Γ a temporální formule $\varphi \in PTL(\Gamma)$ nad Γ. Nyní hlavní problém kontroly modelů (a model checking problem) může být definován následovně

$$M, \Gamma^I \models \varphi$$

který označuje problém dosažitelnosti (reachability problem). V případě konečně stavových systémů (např. hardware), lze model redukovat na konečně stavový automat (či graf) a problém na prohledávaní grafu.



Obrázek 2.4: Konceptuální schéma model checkingu

Na obrázku 2.4 je znázorněné konceptuální schéma model checkingu. Funkce PTL označuje temporální výrokovou logiku (propositional temporal logic, kapitola 2.1.1), φ kontrolovanou vlastnost a Γ^I počáteční (výchozí) stav systému.

Jedním z hlavních problémů model checkingu je *problém stavové exploze (state explosion problem)*[19] (systém má příliš mnoho proměnných nebo je příliš nedeterministický a jeho stavový prostor je potom příliš velký). Tomuto problému je i dnes věnováno mnoho úsilí a existuje několik způsobů, jak explozi stavového prostoru předejít. Zde je pouze strohý výčet:

Abstrakce - zabývá se abstraktním popisem systému, který bude mnohem menší než jeho konkrétní model

Redukce částečným uspořádáním - poukazuje na to, že některé kroky systému se mohou provést současně a jejich pořadí neovlivní výsledný účinek. Tohoto se využívá při generování stavového prostoru. Užití má metoda převážně v paralelních systémech.

Kompozitní verifikace - využívá faktu, že se na systém můžeme dívat jako na kompozici menších podsystémů. Tyto podsystémy jsou ověřeny každý zvlášť a tím dochází k redukci původního stavového prostoru.

Mezi typické problémy řešené pomocí model checkingu patří *analýza dosažitelnosti (reachability analysis)*, *živost (liveness)*. První z nich, analýza dosažitelnosti představuje základní problém model checkingu a byla již částečně představena výše a druhá, živost, je analýzou dynamických systémů, kde dochází z různých důvodů k čekání některých procesů

na jiné. Ověření živosti zjišuje, zda se všechny procesy dostanou k běhu v konečném čase, či nejlépe stejně často.

2.2 Regulární model checking (RMC)

Regulární model checking představuje adaptaci výše uvedeného model checkingu pro verifikaci nekonečně stavových systémů. Vznik RMC je dán v současnosti stále vzrůstající důležitostí nekonečně stavových systémů a potřebou jejich formální analýzy.

2.2.1 Regulární relace

Regulární model checking je postaven na regulárních relacích[12], které budou použity pro reprezentování relace přechodů. Regulární relace jsou přijímány automaty podobným způsobem jako regulární množiny jsou přijímány konečným automatem.

Definice 4 ***(křížení, cross-product)*** *Nechť $\Sigma_1, \Sigma_2, \ldots, \Sigma_m$ jsou konečné abecedy. Pak pro slova $a_1^j, a_2^j, \ldots, a_n^j \in \Sigma_j^n$ stejné délky n pro j, $1 \leq j \leq m$ je jejich křížení (cross product[12])*

$$a_1^1.a_2^1.\ldots.a_n^1 \times a_1^2.a_2^2.\ldots, a_n^2 \times \ldots \times a_1^m.a_2^m.\ldots.a_n^m$$

definováno jako slovo

$$(a_1^1, a_2^1, \ldots, a_n^1).(a_1^2, a_2^2, \ldots, a_n^2) \ldots \times (a_1^m, a_2^m, \ldots, a_n^m)$$

nad abecedou $\Sigma_1 \times \Sigma_2 \times \ldots \times \Sigma_m$.

Jazyk obsahující pouze taková slova kóduje relace následujícím způsobem. V jazyce L nad abecedami $\Sigma_1 \times \Sigma_2 \times \ldots \times \Sigma_m$ označuje $[L]$ relace skládající z množiny n-tic $(w_1, w_2, \ldots, w_m)$ takových, že $w_1 \times w_2 \times \ldots \times w_m$ je v L.

Relace, které mohou být reprezentovány regulárním jazykem uvedeným způsobem jsou nazývány *regulární.*

Definice 5 ***(Regulární relace)*** *Nechť $\Sigma_1, \Sigma_2, \ldots, \Sigma_m$ jsou konečné abecedy. Relace $R \subseteq \Sigma_1^* \times \Sigma_2^* \times \ldots \times \Sigma_m^*$ arity m je regulární, když existuje regulární jazyk L nad $\Sigma_1, \Sigma_2, \ldots, \Sigma_m$ takový, že $[L] = R$.*

Definice 6 ***(Konkatenace, sjednocení a průnik regulárních relací)*** *Nechť R a R' jsou regulární relace stejné arity a nechť máme dva regulární jazyky L a L' takové, že $[L] = R$ a $L' = [R']$. Konkatenací $R.R'$ je konkatenace reg. jazyků $L.L'$, sjednocením $R \cup R'$ je $L \cup L'$ a průnikem $R \cap R'$ je $L \cap L'$. Regulární relace jsou uzavřené na sjednocení a průnik*[1]*.*

Pro dvě relace R arity m a R' arity m' nazýváme jejich *cross product* jako *zachovávající délku (length-preserving)* $R \overline{\times} R'$ a definujeme jej jako množinu n-tic $(w_1, w_2, \ldots, w_m, w'_1, w'_2, \ldots, w'_{m'})$ takových, že všechna slova $w_1, w_2, \ldots, w_m, w'_1, w'_2, \ldots, w'_{m'}$ jsou stejné délky a $(w_1, w_2, \ldots, w_m) \in R$ a $(w'_1, w'_2, \ldots, w'_{m'}) \in R'$

Zajímavým případem regulárních relací jsou binární regulární relace, které budeme dále používat k reprezentování relací přechodů našich problémů v RMC. *Kompozice* binárních

[1] Důkazy jsou triviální (prováděné nad ekvivalentními regulárními jazyky) a je možné je dohledat v [12]

regulárních relací je důležitá, protože nám bude modelovat běh (násobné přechody) systému. Konkrétněji, když R reprezentuje relaci přechodů jednoho kroku programu, tak kompozice $R \circ R$ reprezentuje změnu stavu systému ve dvou krocích, kde $\circ$ představuje binární operátor kompozice.

Tvrzení 1 *Nechť R a R' jsou binární regulární relace na Σ. Pak jejich složení $R \circ R'$ je regulární.*

Další velmi významnou věcí v kontextu RMC je *tranzitivní uzávěr (transitive closure)* R, značený R^+, a *reflexivní a tranzitivní uzávěr (reflexive and transitive closure)* značený R^*. Pokud nám R reprezentuje relaci přechodů mezi stavy, pak R^* reprezentuje přechody z jednoho stavu do všech ostatních, do kterých je možné se dostat v žádném (reflexivita) a více (trazitivita) krocích. Regulární relace nejsou uzavřené vůči těmto operacím.

Tvrzení 2 *Existuje taková regulární relace R, jejíž tranzitivní uzávěr R^* není regulární.*

Důkazy posledních dvou tvrzení je možné dohledat v [12].

Transducer

Regulární relace mohou být reprezentovány automatem nad dvojicemi znaků abecedy. Pojďme se na takové automaty podívat formálněji.

Nechť Σ je konečná abeceda a $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$. Konečný transducer τ nad Σ je pětice $(Q, \Sigma_\epsilon \times \Sigma_\epsilon, \delta, q_0, F)$, kde Q je konečná množina stavů, $\delta \subset Q \times \Sigma_\epsilon \times \Sigma_\epsilon \times Q$ je množina přechodů, $q_0 \in Q$ je počáteční stav a $F \subset Q$ množina koncových stavů. Konečný transducer je označován jako zachovávající délku (length-preserving), když $\delta \subset Q \times \Sigma \times \Sigma \times Q$.

Relace přechodů $\rightarrow \subset Q \times \Sigma^* \times \Sigma^* \times Q$ je definována jako nejmenší relace zachovající:

1. $q \overset{\epsilon,\epsilon}{\rightarrow} q$ pro každé $q \in Q$,
2. když $(q, a, b, q') \in \delta$, pak $q \overset{a,b}{\rightarrow} q'$,
3. když $q \overset{w,u}{\rightarrow} q'$ a $q' \overset{a,b}{\rightarrow} q''$, pak $q \overset{wa,ub}{\rightarrow} q''$.

Pak, v souladu s předchozí definicí je transducer τ určen množinou relací $\left\{(w,u) | \exists q' \in F : q_0 \overset{w,u}{\rightarrow} q'\right\}$. Relace je nazývána regulární, když může být určena nějakým transducerem. Pro množinu $L \subseteq \Sigma^*$ a relace $R \subseteq \Sigma^* \times \Sigma^*$, označuje $R(L)$ množinu $\{w \in \Sigma^* | \exists w' \in L : (w', w) \in R\}$.

Nechť $id \subseteq \Sigma^* \times \Sigma^*$ je relace rovnosti (identity relation) a $\circ$ složení (composition) relací. Pak definujeme rekursivně relace $\tau^0 = id, \tau^{i+1} = \tau \circ \tau^i$, a $\tau^* = \cup_{i=0}^{\infty} \tau^i$.

2.2.2 Regulární modely

Pro potřeby regulárního model checkingu je nutné mít vhodný model zkoumaného systému, který bude dovolovat kontrolu vlastností pomocí automatizovaných prostředků.

Definice 7 *Nechť Σ je nějaká abeceda, Γ^I je množina počátečních konfigurací a Γ^F množina akceptujících konfigurací. Regulárním modelem nad abecedou Σ je model $(\Gamma^I, \rightarrow, \Gamma^F)$ nad Σ^*, kde $\Gamma^I, \rightarrow$ a Γ^F jsou regulární.*

Čili, v regulárním modelu jsou množiny a relace regulární a tudíž mohou být reprezentovány pomocí konečných automatů, které budou použity níže uvedenými algoritmy pro verifikaci.

Jak bylo uvedeno v kapitole 2.1.1, pro transformaci nějakého systému na regulární model můžeme zvolit více reprezentací, kde každý stav takového systému představuje slovo nad nějakou abecedou. Množina počátečích stavů $Init$ je regulární množinou nad nějakou abecedou Σ a relace přechodů je regulární relací nad Σ. V souladu s definicí musíme být schopni najít reprezentaci, aby počáteční konfigurace systému (počáteční konfigurace stavů) byla regulární.

Například, předpokládejme, že potřebujeme reprezentovat dvě celočíselné proměnné x a y za použití abecedy $\Sigma = \{x, y\}$, a zvolíme reprezentaci stavu systému takovou, že $x = n$ a $y = m$ a ve tvaru slov $x_n y_m$. Taková reprezentace není vhodná, protože není možné reprezentovat množinu stavů $x = y$, protože množina $\{x^n y^m | n \geq 0\}$ není regulární. Nicméně můžeme zvolit jako reprezentaci dvou celočíselných proměnných dvě binární proměnné b_x a b_y s abecedou $\Sigma = \{true, false\} \times \{true, false\}$, kde *cross product* domén b_x a b_y a reprezentaci množiny stavů, kde $x = n$ a $y = m$ se slovy w takovými, že symbol na pozici i je $b_x = true$, pro $n = i$ a $b_y = true$ pro $m = i$, pak je množina reprezentující $x = y$ regulární a může být popsána například pomocí regulárního výrazu jako $(\neg b_x \wedge \neg b_y)^*.(b_x \wedge b_y).(\neg b_x \wedge \neg b_y)^*$.

Při hledání modelů narážíme na základní tři charakteristické třídy. Jsou to *parametrizované systémy*, které mají potenciálně neomezený počet procesů obvykle organizovaných ve stejném prostoru, dále *celočíselné proměnné* nad přirozenými čísly s neomezeným rozsahem a *fronty a zásobníky*, které představují například fronty mezi procesy či modelování komunikačních kanálů.

V následující části bude vysvětlena tvorba regulárních modelů uvedených základních tříd pro účely regulárního model checkingu.

Parametrizované systémy

Parametrizovanými systémy označujeme systémy parametrizované nějakým počtem procesů. Typicky se jedná o algoritmy, které jsou navrženy pro neomezený počet souběžně běžících procesů (synchronizační protokoly paralelních systémů). V tomto případě chceme zpravidla verifikovat nezávislost takových systémů na počtu souběžných procesů.

Předpokládáme, že všechny procesy jsou stejné (jejich běhy mohou nabývat stejné množiny stavů Q). Jako abecedu vezmeme množinu stavů každého procesu, např. $\Sigma = Q$. Pak každé slovo $a_1 a_2 \ldots a_n \in \Sigma^*$ představuje přepis, kde proces na pozici i je ve stavu a_i, pro $1 \leq n \leq n$.

Přechody uvnitř procesu nejsou nijak ovlivněné ostatními procesy a mohou být reprezentovány regulární relací $Id_{Q^*}.[(q, q')].Id_{Q^*}$, která představuje přechod procesu ze stavu q do stavu q'. Id_{Q^*} představuje identitu (regulární relaci $[(q_1, q_1), (q_2, q_2), \ldots, (q_n, q_n)]$, $pro\, n \geq 0$, $q_1 q_2 \ldots q_n \in Q^*$), která znamená, že systémy zůstávají ve stejném stavu (přechází do stejného stavu ve kterém byly před započetím přechodu).

Typickým příkladem takovýchto systémů je *přeposílání tokenu (token ring)*. Každý stav se může nalézt pouze ve dvou stavech, *má* nebo *nemá* token, kde *má* je reprezentováno stavem T a *nemá* stavem označeným N. Jako množinu konfigurací Γ máme tedy množinu $\{N, T\}^*$ a výchozím stavem takového multiprocesového systému bude například TN^*, první proces *má* token a ostatní *nemají*. Relace přechodů bude reprezentována regulární relací

$$Id_{N^*}.[(T,N).(N,T)].Id_{N^*} \cup Id_{N^*}.Id_{\{N,T\}}.Id_{N^*},$$

která se skládá ze dvou částí, kde první reprezentuje posun tokenu na pravého souseda (index $i+1$) a druhá „prázdný“ běh, kdy se stav multiprocesového systému nemění.

Pokud bychom chtěli modelovat také přenos tokenu z posledního procesu zpět na první (kruhová topologie), tak stačí pouze zavést

$$[(N,T)].Id_{N^*}.[(T,N)]$$

do relace přechodů.

Celočíselné proměnné

Ve většině systémů se pracuje s celočíselnými proměnnými $x_1, x_2, \ldots, x_n$. Pro modelování takových proměnných pomocí regulárních modelů přiřaďme každé takové proměnné x_1 boolean proměnnou b_i. Jako abecedu tudíž zvolme množinu $\Sigma = \{true, false\}^n$. Každý stav (hodnota celočíselné proměnné) je reprezentován slovem nad Σ tak, že odpovídající boolean proměnné b_i je $true$ na pozici j, když $x_i = j$ a na ostatních $false$.

Uvedená reprezentace dovoluje tvorbu různých omezujících podmínek nad celočíselnými proměnnými. Například, pro dvě proměnné x a y můžeme reprezentovat $x < y, x \leq y$ a $x = y + c$ pro nějakou konstantu c. Omezení $x = y + c$ můžeme zapsat regulární relací jako

$$[(y = y')^*] \cap [(\neg x' \wedge \neg y)^*).(\neg x' \wedge y).(\neg x' \wedge y).(\neg x' \wedge \neg y)^{c-1}.(x' \wedge \neg y).(\neg x' \wedge \neg y)^*]$$

Fronty

Mějme systém s frontou obsahující množinu zpráv M, kde M^* reprezentuje obsah fronty jako množinu konfigurací. Uvažujme *length-preserving* regulární relace. Musíme přidat zarovnávací symbol $\perp$ ošetřující změnu délky fronty. Symbol $\perp$ reprezentuje nevyužitou položku, která může být nahrazena nějakou novou zpravou přidávanou do fronty. Podobně, pokud vyjmeme zprávu z fronty, tak jej potřebujeme změnit zpět na $\perp$.

Rozeberme nyní konkrétněji reprezentaci systému, který se bude skládat z řídícího stavu a jedné FIFO fronty. Nechť Q označuje konečnou množinu řídích stavů a M označuje konečnou množinu zpráv ve frontě. Abecedou bude sjednocení těchto množin a přidáme navíc jeden symbol „výplně“ $\perp$ pro prázdný slot, tedy $\Sigma = Q \cup M \cup \{\perp\}$. Každý stav takového systému je tedy reprezentován slovem nad Σ ve tvaru $Q.\ \perp^*\ .M^*.\ \perp^*$, kde na první pozici je kontrolní stav a ve zbytku FIFO fronta. Symbol $\perp$, reprezentující prázdnou pozici, dovoluje růst a zkracování délky FIFO fronty.

Vložení zprávy $m \in M$ a pro kontrolní stav $q \in Q$ můžeme použít regulární relaci $[(q,q)].Id_{\perp^*}.Id_M.[(\perp,m)].Id_{\perp^*}$. Vyjmutí zprávy $m \in M$ ze systému o kontrolním stavu $q \in Q$ z poslední pozice FIFO fronty je reprezentováno regulární relací $[(q,q)].Id_{\perp^*}.[(m,\perp)].$ $.Id_{M^*}.Id_{\perp^*}$

Zásobníky

Poslední charakteristickou třídou jsou zásobníky, na které je možné se dívat podobným způsobem jako na fronty. Mějme systém jako v předchozím textu, který ovšem nebude obsahovat FIFO frontu, ale zásobník. Nechť Q označuje konečnou množinu řídících stavů a M označuje množinu zpráv na zásobníku. Abecedou Σ bude $Q \cup M \cup \perp$, kde symbol $\perp$ značí prázdnou pozici na zásobníku. Každý stav takového systému může být reprezentován slovem nad $Q.M^*.\ \perp^*$, kde první pozice nese kontrolní stav systémů a ve zbytku je reprezentace zásobníku.

Vložení prvku na vrchol zásobníku se zachováním kontrolního stavu je reprezentováno regulární relací $[(q,q)].Id_{M^*}.[(\perp,m)].Id_{\perp^*}$. Vyjmutí prvku ze zásobníku je reprezentováno regulární relací $[(q,q)].Id_{M^*}.[(m,\perp)].Id_{\perp^*}$.

2.2.3 Regulární model checking (RMC)

Při rozvoji klasického model checkingu (hlavní doména hardware) vznikla potřeba kontrolovat také nekonečně stavové systémy (převážně pro algoritmy/software). Mezi hlavní motivace patří také rostoucí zájem o nekonečně stavové systémy, mezi které patří neomezené zásobníkové systémy, FIFO paměti, systémy s čítači či parametrizované a dynamické víceprocesorové systémy.

Regulární model checking je kontrola určité vlastnosti nad regulárním modelem. Jako model je používán regulární model, pro modelování chování (běhu) se využívá regulárních kompozicí (regular composition), které jsou reprezentovány pomocí regulárních konečných transducerů nad nějakou abecedou Σ (problematika transducerů rozebrána v kapitole 2.2.1), regulární množinou $Init$ reprezentovanou pomocí konečného automatu a kontrolované vlastnosti Bad reprezentované opět pomocí konečného automatu.

Nyní, hlavní problém regulárního model checkingu může být definován jako

$$\tau^*(Init) \cap Prop \neq \oslash,$$

kde množina $\tau^*(Init)$ reprezentuje všechny dosažitelné konfigurace systému z počáteční konfigurace $Init$.

Jinou možností je výpočítat τ^* a následně kontrolovat pouze

$$\tau^* \cap \tau_{Bad} \neq \oslash,$$

kde τ_{Bad} popisuje chování chyby (kontrolované vlastnosti).

Hlavní problém první metody regulárního model checkingu je ve výpočtu množiny $\tau^*(Init)$, která je počítána iterativně

$$\tau^*(Init) = Init \cup \tau(Init) \cup \tau(\tau(Init)) \cup \ldots$$

a obvykle je výpočet nekonečný. U druhého přístupu je hlavní problém ve výpočtu τ^*, který je zpravidla výrazně složitější než výpočet množiny $\tau^*(Init)$ a dokonce existují případy kde $\tau^*(Init)$ je regulární, ale τ^* není.

2.3 Boolean Satisfiability (SAT) problem

SAT problém je pravděpodobně nejvíce zkoumaný kombinační/optimalizační/vyhledávací problém dnešní doby. Tento přístup k řešení problémů můžeme vidět v mnoha různých oborech jako jsou počítačové vědy, umělá inteligence nebo vývoj elektronických obvodů.

SAT problém je možné reprezentovat jako dvojici (X, C), kde $X = \{x_1, x_2, \ldots, x_n\}$ je konečná množina vázaných proměnných, které nabývají boolovských hodnot a $C = \{C_1, C_2, \ldots, C_m\}$, množina klauzulí obsahující omezující podmínky (budou rozvedené dále). Hlavní problém spočívá v nalezení takových hodnot proměnných X splňujících booleovskou funkci f skládajících se z *klauzulí* C nebo zjištění, že takové nastavení hodnot proměnných X neexistuje. SAT problém je jeden z hlavních NP-kompletních problémů[2].

Většina algoritmů hledajících řešení pracuje s problémy, jejichž funkce f je zadána v konjunktivní normální formě (CNF). CNF je tvar, kdy se logické funkce f skládá pouze z logických konjunkcí (AND funkcí) jedné nebo více *klauzulí (clauses)*, které se skládají pouze z logických disjunkcí (OR funkcí) atomických hodnot (*literálů*). Literál je základní atomická logická jednotka řešeného problému (instance proměnné nebo její komplement). Všechny boolovské funkce mohou být algoritmicky převedeny do CNF tvaru, jehož podoba je tedy

$$x_{11} \wedge (x_{21} \vee x_{22} \vee x_{23}) \wedge (x_{31} \vee x_{32}) \ldots$$

Důvod pro používání CNF pro reprezentaci problému je dán tím, že pro splnění celé boolovské funkce f musí být také každá klauzule splněna. V některých případech může být výhodné mít funkci f ve formě, kde každá klauzule obsahuje právě k literálů. SAT problém pracují s takovým zápisem f je označován jako *k-SAT* a typickými hodnotami k jsou 2 nebo 3.

$$(x_{11} \vee x_{12} \vee x_{13}) \wedge (x_{21} \vee x_{22} \vee x_{23}) \wedge (x_{31} \vee x_{32} \vee x_{33}) \ldots$$

Tento příklad ($k = 3$) je označován *3-SAT* a některé algoritmy mohou být více efektivní či přímo vytvářené pro práci s takovou formou[15].

2.3.1 Speciální případy SAT problémů

Existuje mnoho modifikací uvedeného SAT problému, které jsou vhodné na jistou doménu úloh. Jednou z nich je *kritický SAT (critical SAT)*[20], který má právě jednu nesplnitelnou klauzuli *clause* nebo *jedinečný SAT (unique SAT (USAT))*[17], kde existuje právě jedno přiřazení logických hodnot klauzulím, při kterém je funkce f splněna.

Mezi další zajímavé třídy SAT problémů patří například (N)HORNSAT[16]. (N)HORNSAT boolovská funkce f je popsána v CNF formátu a s omezeními, kde každá klauzule je disjunkcí literálů s nejvíce jednou negací literálu *(HORNSAT)* nebo nejvíce s jedním pozitivním literálem *(NHORNSAT)*.

2.3.2 Rozšíření SAT

SAT problém se stává komplikovanějším zavedeme-li kvantifikátory boolovských proměnných. Příkladem takové funkce může být

$$\forall x \exists y \exists z (x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z)$$

[2]SAT problém byl prvním známým NP-kompletním problém s důkazem nalezeným v roce 1971[7]

Pokud použijeme pouze existenční ($\exists$) kvantifikátor, tak se jedná o problém ekvivalentní s předchozími uvedenými. Pokud ovšem zavedeme jen obecný ($\forall$) kvantifikátor, tak dostaneme třídu *co-NP-kompletních* problémů. Třída problémů s oběma kvantifikátory se nazývá *problém kvantifikovaných booleovských problémů (QBF)*, který lze řešit jako *PSPACE-kompletní.*

2.3.3 Existující algoritmy pro řešení SAT

Existují dvě základní kategorie algoritmů - *kompletní* a *nekompletní*. Hlavní rozdíl spočívá v tom, že kompletní algoritmy při řešení neuspokojitelných boolovských funkcí f vrátí odpověď „ UNSAT - neřešitelné“, u kterých nekompletní algoritmy nikdy neskončí (nevrátí žádnou odpověď v konečném čase).

Mezi kompletní algoritmy patří *pravdivostní tabulka, prohledávání do hloubky s návratem (backtracking), rezoluce, rekursivní učení, ...* a mezi nekompletní patří *lokální vyhledávání (GSAT, Random walk, ...), genetické algoritmy, ...*

2.3.4 Existující implementace SAT problémů (SAT solvers)

Pravděpdobně nejlepší implementací SAT problému je *zChaff*[3], který byl vyvinut na Princeton universitě a je založen na prohledávání do hloubky s návratem (backtracing) algoritmu s mnoha optimalizacemi (samotný algoritmus je nazýván *chaff*[10]). zChaff napsán v jazyce C++ and pro nekomerční použití je zdarma.

Další zajímavou implementací v jazyce C++ je *MINISAT*, která je také založena na algoritmu *chaff*. Velkou výhodou MINISATu je implementace na cca 600 řádků (bez komentářů) a výborné rozhraní pro zakomponování do dalších projektů.

2.4 Algoritmus Angluin $\mathcal{L}^*$

Angluin $\mathcal{L}^*$ algoritmus[1, 3] je aktivní algoritmus vyvinutý v roce 1987 pro učení regulárních množin $\mathcal{L} \subseteq \Sigma^*$, založený na konstrukci minimálního DFA $\mathcal{A}$, kde $\mathcal{L}(\mathcal{A}) = \mathcal{L}$. Pro svoji činnost požaduje *učitele*, který je obvykle nazýván *minimální dostačující učitel (Minimally Adequate Teacher - MAT)*, který poskytuje nápovědu, odpověď na dotazy členství (membership queries) nějakého řetězce ve výsledné regulární množině a *dotazy ekvivalence (equivalence queries)*, zda hypotéza (DFA $\mathcal{H}$ extrahovaný z algoritmu) je odpovídající výslednou regulární množinou $\mathcal{L}(\mathcal{H}) = \mathcal{L}$. V případě, že učitel odpoví „ne“ na dotaz ekvivalence, je vrácen protipříklad $w \in \mathcal{L} \backslash \mathcal{L}(\mathcal{H})$ nebo $w \in \mathcal{L}(\mathcal{H}) \backslash \mathcal{L}$.

Hlavní myšlenka algoritmu Angluin $\mathcal{L}^*$ je v systematickém prohledávání hledané regulární množiny (na počátku není známo nic o výsledné reg. množině) a vytváření DFA s minimálním počtem stavů k odvození výsledné regulární množiny. Pokud je odvozený DFA $\mathcal{H}$ chybný ($\mathcal{L}(\mathcal{H}) \neq \mathcal{L}$), je vrácen řetězec (např. nejkratší délky), který je následně použit jako korekce pro další iteraci učení.

Algoritmus udržuje prefixově uzavřenou (prifix-closed) množinu $\mathcal{S} \subseteq \Sigma^*$ reprezentující možné stavy výsledného DFA, množinu $\mathcal{S}.\Sigma$ pro přechodovou funkci obsahující řetězce z množiny $\mathcal{S}$ konkatenované se znaky vstupní abecedy Σ a suffixově uzavřenou (suffix-closed) množinu $\mathcal{E} \subseteq \Sigma^*$ označovanou jako *experimety (experimets)* k rozlišování stavů. Výše uvedené množiny jsou spojené přes *tabulku pozorování (observation table - $\mathcal{OT}$)*, která může

[3]ZChaff získal v soutěži SAT 2004 Competition, ocenění jako nejlepší solver v kategorii průmyslových solverů. V roce 2002 byl také oceněn jako nejlepší kompletní solver, a to jak pro průmyslové, tak pro umělé problémy.

být znázorněna jako dvoudimenzionální pole s řádky jako prvky množiny $s \in \mathcal{S}.\Sigma$ a sloupci jako prvky množiny $e \in \mathcal{E}$, kde pro řádek s a sloupec e je pole tabulky $\mathcal{OT}$ značeno $\mathcal{OT}(s.e)$.

Definice 8 ***(Observation table)*** *Observation table $\mathcal{OT}$ nad nějakou abecedou Σ je trojice $OT = (\mathcal{S}, \mathcal{E}, \mathcal{T})$, kde*

- $\mathcal{S} \subseteq \Sigma^*$ *neprázdná konečná prefixově uzavřená množina,*
- $\mathcal{E} \subseteq \Sigma^*$ *neprázdná konečná prefixově uzavřená množina a*
- $\mathcal{T} : ((\mathcal{S} \cup \mathcal{S}.\Sigma) \times \mathcal{E}) \to 0, 1$ *je funkce určující náležitost řetězce do výsledného DFA $\mathcal{A}$*

Pro každé $s \in (\mathcal{S} \cup \mathcal{S}.\Sigma)$ je dán řádek tabulky $row(s)$, který je mapován funkcí $f : \mathcal{E} \to \{0, 1\}$ definovanou jako $f(e) = \mathcal{T}(s.e)$.

Definice 9 ***(Uzavřená observation table)*** *Observation table $\mathcal{OT}$ je nazývána uzavřenou, když pro každé $t \in S.\Sigma$ existuje nějaké $s \in S$, takové že $row(t) = row(s)$.*

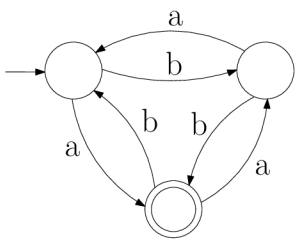
Definice 10 ***(Konzistentní observation table)*** *Observation table $\mathcal{OT}$ je nazývána konzistentní, když pro každé různá s_1, $s_2 \in \mathcal{S}$ taková, že $row(s_1) = row(s_2)$ platí $row(s_1.a) = row(s_2.a)$ pro všechna $a \in \Sigma$.*

Když $\mathcal{OT}$ je *uzavřená*, *konzistentní* je možné extrahovat výsledný DFA $\mathcal{H} = (Q, \delta, \Sigma, q_0, F)$ následovně:

- $Q = \{row(s) | s \in \mathcal{S}\}$
- $q_0 = row(\epsilon)$
- $F = \{row(s) | s \in \mathcal{S} \wedge \mathcal{OT}(s) = 1\}$
- $\delta(row(s), a) = row(s.a)$

Získaný DFA $\mathcal{H}$ je pomoci *dotazu ekvivalence* zkontrolován na $\mathcal{L}(\mathcal{H}) = \mathcal{L}$ a v případě nesouladu opraven v dalších iteracích algoritmu.

S/E	ϵ	a
ϵ	0	1
a	1	0
b	0	0
aa	0	0
ab	0	1
ba	0	1
bb	1	0



Obrázek 2.5: Příklad $\mathcal{OT}$ tabulky a odpovídající DFA $\mathcal{H}$

Angluin algoritmus (algoritmus 1) končí v polynomiálním čase[3] s výstupním DFA $\mathcal{H}$ reprezentujícím hledanou regulární množinu $\mathcal{L}$.

Algoritmus 1 Angluin $\mathcal{L}^*$

Inicializace $\mathcal{S}$ and $\mathcal{E}$ to $\{\epsilon\}$
Dotazy na členství pro ϵ and každé $a \in \Sigma$
Vytvoření počáteční $\mathcal{OT}$ tabulky $(\mathcal{S}, \mathcal{E}, \mathcal{T})$
repeat
  while $(\mathcal{S}, \mathcal{E}, \mathcal{T})$ není uzavřené nebo konsistentní **do**
    if $(\mathcal{S}, \mathcal{E}, \mathcal{T})$ není konzistentní **then**
      najdi s_1 and s_2 in S, $a \in \Sigma$ and $\epsilon \in \mathcal{E}$ takové, že
      $row(s_1) = row(s_2)$ and $\mathcal{T}(s_1.a, e) \neq \mathcal{T}(s_2.a, e)$
      přidej $a.e$ do E
      rozšiř $\mathcal{OT}$ na $(\mathcal{S} \cup \mathcal{S}.\Sigma).\mathcal{E}$ použitím dotazů členství
    end if
    if $(S, E, \mathcal{T})$ není uzavřen **then**
      find $s_1 \in S$, $a \in \Sigma$ takové, že
      $row(s_1.a)$ je různé od $row(s)$ pro každé $s \in \mathcal{S}$
      přidej $s_1.a$ do $\mathcal{S}$
      rozšiř $\mathcal{OT}$ na $(\mathcal{S} \cup \mathcal{S}.\Sigma).\mathcal{E}$ použitím dotazů členství
    end if
  end while
  Když $(\mathcal{S}, \mathcal{E}, \mathcal{T})$ je uzavřené a konsistentní, $\mathcal{H} = \mathcal{H}(\mathcal{S}, \mathcal{E}, \mathcal{T})$
  Extrahuj DFA $\mathcal{H}$
  Učitel položí dotaz *ekvivalence* na $\mathcal{H}$
  if Učitel vrátil protipříklad t **then**
    přidej t a všechny jeho prefixy do $\mathcal{S}$
    rozšiř $\mathcal{OT}$ na $(\mathcal{S} \cup \mathcal{S}.\Sigma).\mathcal{E}$ použitím dotazů členství
  end if
until Učitel odpoví „ano“ na dotaz ekvivalence $(\mathcal{L}(\mathcal{H}) = \mathcal{L}\ )$
vrať $\mathcal{H}$

2.5 Algoritmus Biermann

Algoritmus Biermann[5, 8] je algoritmus pro učení (odvození) DFA $\mathcal{A}$ ze vzorků (samples) reg. množiny.

Formální definice algoritmu Biermann vypadá následovně. Vzorky jsou množina řetězců, které jsou buď akceptovány, označeny symbolem „+“ nebo odmítnuty, značeny symbolem „−“. Z technických důvodů je vhodné pracovat s prefixově uzavřenou (prefix-closed) množinou vzorků.

Pro množinu vzorků, která není prefixově uzavřena (prefix-closed) zavádíme hodnotu *možná (maybe)*, označovanou symbolem „?“. Formálně, *vzorek* je funkce $0 : \Sigma^* \to \{+, -, ?\}$, která je definována pro u, pokud je definována pro nějaké ua. Pro řetězec u vzorek O udává, zda by u měl být *přijat*, *odmítnut* nebo *na něm nezáleží*. Pro řetězce u a u' říkáme, že O *nesouhlasí* na u a u', když $O(u) \neq ?$, $O(u') \neq ?$ a $O(u) \neq O(u')$.

Automat $\mathcal{A}$ je v souladu se vzorky O, když pro každé O definované na u platí $O(u) = +$ implikuje $u \in L(A)$ nebo $O(u) = -$ implikuje $u \notin L(A)$. Daný vzorek O a DFA $\mathcal{A}$ jsou v souladu s O, pak S_u označuje stav dosažitelný v A čtením u. Pokud ještě nemáme definován DFA A, můžeme pracovat s S_u jako s proměnnou nad stavy automatu $\mathcal{A}$ a odvodit omezující podmínky pro takové proměnné. Přesněji, nechť $CSP(O)$ označuje následující množinu

rovnic

$$\begin{array}{lll} & \{S_u \neq S_{u'} & \mid \quad O \text{ nesouhlasí na } u \text{ a } u'\} \quad \text{(C1)} \\ \cup & \{S_u = S_{u'} \Rightarrow S_{ua} = S_{u'a} & \mid \quad a \in \Sigma, ua, u'a \in \mathcal{D}(0)\} \quad \text{(C2)} \end{array}$$

Nechť doména $\mathcal{D}(CSP(O))$ určuje množinu proměnných S_u použitou v omezujících podmínkách. Řešení $CSP(O)$ je zobrazení $\Gamma : \mathcal{D}(CSP(O)) \to IN$ splňující uvedené rovnice. Množina $CSP(O)$ je řešitelná nad $[N]$ právě když existuje řešení na rozsahu $[N]$. Z daného vyplývá, že každé řešení CSP problému nad přirozenými čísly může být převedeno na automat odpovídající vzorkům O.

Tvrzení 3 ***(Učení jako CSP,[5])*** *Pro nějaké vzorky O existuje DFA $\mathcal{A}$ s N stavy v souladu s O právě když $CSP(O)$ je řešitelné nad $[N]$.*

2.5.1 Omezení stavového prostoru CSP

Obecně je možné najít minimální DFA k odpovídajícímu CSP problému postupným zkoušením hledání automatu s N, kde $N \geq 0$. Nicméně je možné najít zjednodušení CSP. Nechť máme bijekci $i : [N] \to [N]$, kterou nazveme *přejmenování* a řekněme, že Γ a Γ' jsou *modulo ekvivalentní* $\Gamma = i \circ \Gamma'$.

Tvrzení 4 ***(Nezávislost na pojmenování)*** *Pro vzorky O je $\Gamma : \mathcal{D}(CSP(O)) \to [N]$ řešení $CSP(O)$, když každé přejmenování $i : [N] \to [N], i \circ \Gamma$ je řešením $CSP(O)$.*

Uvedené tvrzení je použito při omezování stavového prostoru řešení. Ke každé proměnné reprezentující stav přiřadíme různá čísla, která budou následně reprezentovat různé stavy.

Definice 11 ***(Jistě různé)*** *S_u a $S_{u'}$ jsou* jistě různé proměnné, *když existuje nějaké $v \in \Sigma^*$ takové, že $O(uv) \neq O(u'v)$ a $O(uv), O(u'v) \neq ?$. Jinak říkáme, že S_u a $S_{u'}$ jsou podobné.*

CSP problém s M jistě různými proměnnými potřebuje nejméně M různých stavů, které společně s tvrzením 3 dává

Tvrzení 5 ***(Omezení stavového prostoru zespodu)*** *Nechť M je počet jistě různých proměnných, pak CSP(O) nemá řešení pro všechna $[N], N < M$.*

Pro řešení $CSP(O)$ je možné *jistě různé proměnné* (v souladu s tvrzením 11) mapovat na různé stavy výsledného automatu.

Tvrzení 6 ***(Mapování jistě různých proměnných na stavy)*** *Nechť $S_{u_1}, \ldots, S_{u_M}$ je M jistě různých proměnných, pak je CSP(O) řešitelné, právě když $CSP(O) \cup \{S_{u_i} = i | i \in [M]\}$ je řešitelné.*

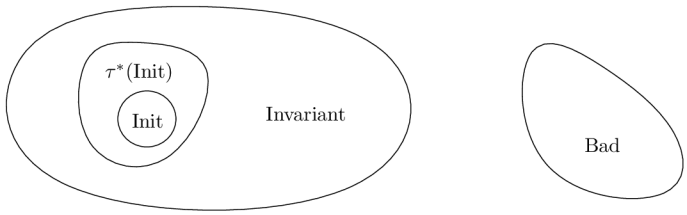
Kapitola 3

Nový algoritmus regulárního model checkingu

Kapitola popisuje nový algoritmus regulárního model checkingu, založený na algoritmu Angluin L^*, jehož myšlenka vznikla za spolupráce členů výzkumné skupiny Tomáše Vojnara (FIT VUT, Brno), Petra Habermehla (LIAFA, Paris) a Martina Leuckera (TU, München).

3.1 Teoretický rozbor

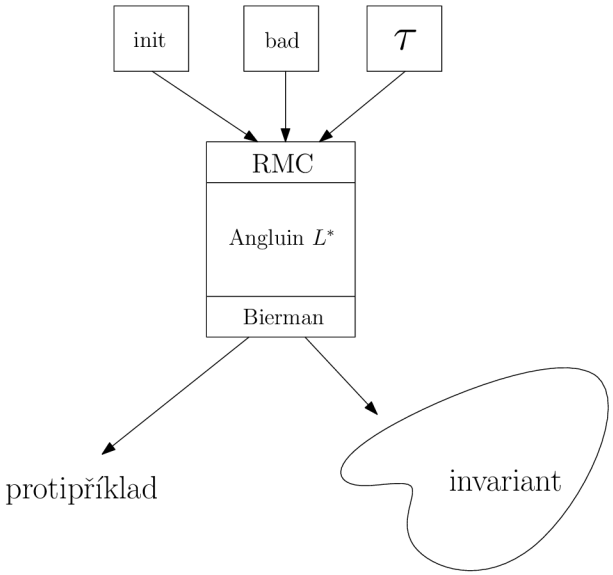
Hlavní myšlenka algoritmu je v hledání invariantu, který obsahuje všechny dosažitelné stavy systémů, ale není v rozporu s kontrolovanou (verifikovanou) vlastností. Problém je ilustrován na obrázku 3.2, kde množina $Init$ představuje počáteční konfiguraci systému, Bad označuje kontrolovanou vlastnost, $\tau^*(Init)$ množinu všech dosažitelných stavů systému z počáteční konfigurace a Inv invariant splňující výše uvedené.



Obrázek 3.1: Vztahy mezi množinami *init*, $\tau^*(init)$, *inv* a *bad*

Poznamenejme, že počáteční konfigurace *init* a kontrolovaná vlastnost *bad* budou reprezentovány konečnými automaty nad abecedou Σ a chování systému bude reprezentováno konečným length-preserving transducerem *trs* nad stejnou abecedou Σ. Jazyky přijímané automaty *init* a *bad* budeme označovat stejnými názvy, ale s velkými počátečními písmeny ($Init$ a Bad) a regulární relace reprezentována trasducerem *trs* bude označena symbolem τ.

Jádrem celého algoritmu je algoritmus Angluin L^* (představen v kapitole 2.4), který ke své činnosti potřebuje učitele, nazýván *minimální adekvátní učitel*, pro zodpovídání *dotazů na členství (membership queries)* ve hledaném invariantu. Připomeňme, že činnost algoritmu je založena na systematickém zkoumání členství daných řetězců (jsou generovány algoritmem Angluin) hledaného invariantu a vytváření DFA s minimálním počtem stavů, který přijímá jazyk invariantu.



Obrázek 3.2: Blokové schéma metody

Aloritmus je navržen na verifikaci systémů, jejichž přechodová funkce může být modelována pomocí length-preserving transducerů. Proto budeme moci některé operace omezit na provádění nad množinami, které budou podmnožinami původních o délce, která nás zajímá.

Například, pokud nás bude zajímat dosažitelnost konfigurace reprezentované slovem w, tak se můžeme omezit pouze na počáteční konfigurace stejné délky, tzn. odpovědí na otázku dosažitelnosti bude $\tau^*(\{w'|w' \in Init \wedge |w'| = |w|\}) \cap \{w\} \neq \oslash$

3.1.1 Minimální adekvátní učitel (MAT)

Jak již bylo řečeno, minimální adekvátní učitel slouží k odpovídání dotazů na členství řetězce w v jazyce hledaného invariantu. Pojďme se podívat na případy, které mohou nastat

1. $w \in \tau^*(Init)$
2. $w \notin \tau^*(Init)$
3. $\tau^*(w) \cap Bad \neq \oslash$
4. $\tau^*(w) \cap Bad = \oslash$

Nejdříve se podívejme na slova w, která splňují právě jednu z uvedených vlastností. Případ (1) znamená, že konfigurace reprezentovaná slovem w je dosažitelná z výchozí konfigurace sekvencí přechodů (počet přechodů ≥ 0). Tento případ říká, že řetězec reprezentující tuto konfiguraci musí být ve výsledném invariantu. Případ (3) říká, že je z této konfigurace možné se sekvencí přechodů (počet přechodů ≥ 0) dostat do konfliktu s Bad. Taková konfigurace tudíž nesmí být obsažena ve výsledném invariantu. Dalším případem jsou konfigurace, která spadají pod (2) nebo (4). Takové konfigurace nejsou dosažitelné z $Init$ nebo není možné se z nich dostat nějakou sekvencí do Bad, takže máme možnost volby o členství těchto řetězců.

Podobně jsou analyzovány i kombinace uvedených možností, kdy například současná platnost (1) a (3) signalizuje dosažení chybového stavu ze stavu dosažitelného z počáteční konfigurace, což signalizuje porušení kontrolované vlastnosti.

Psudoalgoritmus MAT respektující výše uvedené je znázorněn na algoritmu 3.1.1.

Algoritmus 2 Minimální adekvátní učitel

1: $ChoiceValue \Leftarrow$ konstanta určující příslušnost neurčitých řetězců
   $w \Leftarrow$ slovo generované Angluinem L^*
2: $L_1 = \tau^*(\{w' | w' \in Init \wedge |w'| = |w|\})$
3: **if** $L_1 \cap Bad \neq \oslash$ **then**
4: **exit** „porušena verifikovaná vlastnost“
5: **else if** $w \in L_1$ **then**
6: **return** $w/1$
7: **else if** $\tau^*(\{w\}) \cap Bad \neq \oslash$ **then**
8: **return** $w/0$ {w by neměl být v Inv}
9: **else**
10: **return** $w/ChoiceValue$**{**můžeme zvolit, zda vložit či nevložit w do Inv**}**
11: **end if**

Důležité je poznamenat, že dotaz na členství je prováděn pro všechny řetězce, které jsou zkoumány algoritmem Angluin a časová náročnost MAT bude základním faktorem ovlivňujícím výslednou výkonnost celého algoritmu.

3.1.2 Kontrola vlastností invariantu

Podobně jako u analýzy učitele se podíváme na vlastnosti invariantu (dotaz ekvivalence). Korektní invariant musí splňovat následující dvě vlastnosti

1. $Init \subseteq Inv$ a
2. $\tau(Inv) \subseteq Inv$,
3. $Inv \cap Bad = \oslash$,

kde první říká, že Inv musí obsahovat množinu $Init$, druhá zajišťuje uzavřenost vůči operaci „přechod mezi stavy“ a zajišťuje tak, že se není možné dostat do žádné konfigurace mimo invariant (a tudíž do konfigurace, která by nebyla prověřena) a poslední, třetí, že Inv nesmí být v rozporu s Bad (musí mít prázdný průnik).

Dojde-li k porušení některé z uvedených podmínek, je vrácen libovolný protipříklad (např. nejkratší z nich), který bude v následujícím kroku již ošetřen algoritmem Angluin.

Pseudo-algoritmus kontroly invariantu je znázorněn na algoritmu 3.

3.1.3 Výsledný algoritmus regulárního model checkingu

Připomeňme, že algoritmus Angluin začíná zkoumáním prázdného řetězce ε a systematicky prohledává prostor ze spodu, než jsou splněné výše uvedené podmínky na invariant.

Při analýze výše uvedených algoritmů jsme zjistili, že zkoumaný řetězec w

1. musí být obsažen ve výsledném invariantu,
2. nesmí být obsažen v invariantu či
3. jeho členství není určeno.

Algoritmus 3 Kontrola vlastností extrahovaného invariantu

1: $ChoiceValue \Leftarrow$ konstanta určující příslušnost neurčitých řetězců
2: **if** $Init \not\subseteq Inv$ **then**
3: **return** $(w_R \in Init \setminus Inv)/1$
4: **else if** $\tau(Inv) \not\subseteq Inv$ **then**
5: $w \in \{w' | w' \in Inv \wedge \tau(w') \notin Inv\}$ {např. nejkratší délky}
6: **if** $Bad \cap \tau^*(\{w\}) \neq \oslash$ **then**
7: **if** $w \in \tau^*(\{w' | w' \in Init \wedge |w'| = |w|\}$ **then exit** „porušena verif. vlastnost“
8: **else return** $w/0$ {w by neměl být v Inv}
9: **else**
10: **if** $w \in \tau^*(\{w' | w' \in Init \wedge |w'| = |w|\}$ **then return** $w/1$ {w by měl být v Inv}
11: **else return** $w/ChoiceValue$ {můžeme zvolit, zda vložit či nevložit w do Inv}
12: **end if**
13: **end if**

První a druhá možnost je zřejmá, řetězec je jednoznačně mapován na hodnotu „1“ nebo „0“, ale u třetí možnosti je nutné se rozhodnout. Situaci budeme řešit manuálním nastavením mapování takových řetězců, které poté budou buď všechny obsaženy v invariantu (mapovány na „1“) nebo nebudou (mapovány na „0“).

Zamysleme se nyní nad významem takového nastavení detailněji. Budeme-li všechny neurčité (3) řetězce mapovat na hodnotu „0“ (nebudou obsaženy v invariantu), tak bude invariant odvozen pouze od množiny $\{w | w \in \tau^*(Init) \wedge |w| \leq k\}$, pro $k \geq 0$ což si může v některých případech vyžádat více iterací (či dokonce učící proces nemusí skončit v konečném čase) algoritmu Angluin k odvození korektního invariantu.

Budeme-li ovšem všechny neurčité řetězce mapovat na hodnotu „1“ (budou obsaženy v invariantu), bude k odvození invariantu použita množina $\{w | w \in \tau^*(Init) \wedge |w| \leq k\}$, pro $k \geq 0$ a další řetězce (potenciálně všechny, které nejsou v rozporu s Bad, $w \notin Bad$). Učící algoritmus odvozování z takového množiny má zaručené ukončení v konečném čase (buď pozitivní nebo negativní).

Výslednou činností je již jen systematické prohledávání prostoru dle algoritmu Angluin, sestávající se z opakovaného provádění dotazů na členství (MAT), kontroly konzistence a uzavřenosti $\mathcal{OT}$ (včetně případných úprav), extrakce invariantu DFA $\mathcal{H}$ zakončené kontrolou na ekvivalenci s hledaným jazykem a případné zpracování chybně naučených řetězců.

Výsledný algoritmus verifikace lze shrnout s odkazem na algoritmus Angluin jednoduše do algoritmu 4.

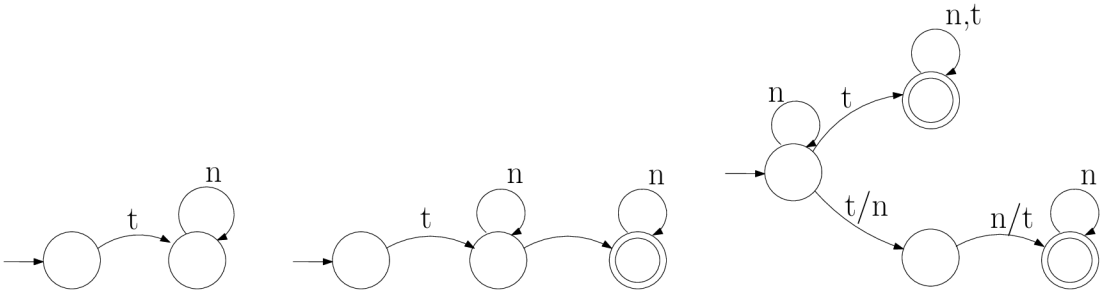
3.2 Demonstrace algoritmu metody

V této části bude provedena detailní demonstrace činnosti algoritmu na známém a jednoduchém problému *posílání tokenu (token passing)*, ve kterém figuruje neomezený počet procesů řazených v lineárním poli (případně kruhovém) a kde index procesu v poli je současně jeho ID. Každý proces se může nalézat ve dvou stavech - vlastní nebo nevlastní token. Operace, které můžou v každém kroku provádět, jsou předání svého tokenu procesu po jeho pravici (ID procesu + 1) nebo ponechání si jej do následujícího kroku.

Zaměřme se na verifikaci vlastnosti (dosažitelnosti konfigurace systému), zda lze docílit stavu, kdy dva procesy vlastní token současně (chybný stav systému).

Algoritmus 4 Celkový algoritmus nové metody formální verifikace

1: $Init \Leftarrow$ regulární množina počátečních konfigurací *init*
   $Bad \Leftarrow$ regulární množina chybných konfigurací *bad*
   $\tau \Leftarrow$ length-preserving transducer *trs*
2: **begin**
3: $stav \Leftarrow$ Angluin($Init$, Bad, τ, Inv)
4: **if** $stav = OK$ **then**
5: **return(„verif. podmínka zachována“,** Inv**)**
6: **else**
7: **return(„verif. podmínka porušena“,** Inv**)** {Inv obsahuje protipříklad}
8: **end if**
9: **end**



(a) init (b) bad (c) τ

Obrázek 3.3: Vstupní konečné automaty a transducer *init*, *bad* a *tau*

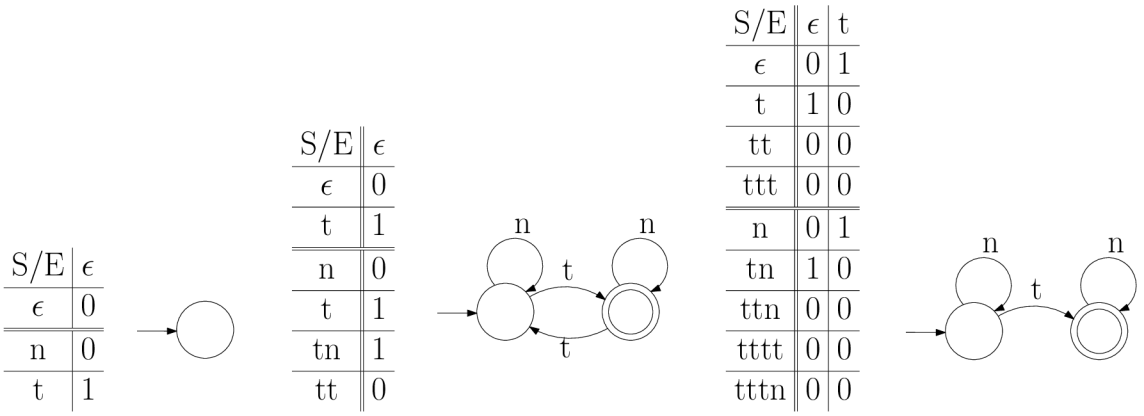
Modely systému pro účely formální verifikace jsou znázorněny na obrázku 3.3, kde počáteční konfigurace *init* (a) a verifikovaná vlastnost *bad* (b) jsou reprezentované konečnými automaty a chování systému je reprezentované transducerem τ (c). Transducer je upraven tak, aby při aplikaci zachoval i původní konfiguraci systému.

Pro první demonstraci budeme mít nastavenou konstantu pro neurčité řetězce na „0“, což znamená, že neurčité řetězce nebudou obsaženy v hledaném invariantu. Simulace běhu metody je zaznamenána na obrázku 3.4, kde je vždy znázorněna observation tabulka $\mathcal{OT}$ a příslušný konečný automat reprezentující invariant pro každou iteraci algoritmu Angluin.

Vysvětlení k jednotlivým krokům z obrázku 3.4

(a) incializace algoritmu angluin - $Init \not\subseteq Inv \Rightarrow$ protipříklad „t“
(b) 1. krok - $Bad \cap Inv \neq \oslash \Rightarrow$ protipříklad „ttt“
(c) 2. krok - invariant nalezen

Podívejme se na operace, které jsme museli vykonat. V inicializaci jsme museli položit 3 dotazy MAT, extrahovat automat z $\mathcal{OT}$ tabulky a položit jeden dotaz ekvivalence. V druhém kroce další 3 dotazy MAT, extrahovat automat a opět jeden dotaz ekvivalence a v posledním kroce 4 dotazy MAT, $\mathcal{OT}$ tabulka není konzistentní $\Rightarrow$ dalších 9 dotazů MAT, extrahovat automat a závěrečný dotaz ekvivalence. Celkem jsme tedy potřebovali $3+3+4+9 = 19$ dotazů MAT a 3 dotazy ekvivalence. Tyto hodnoty jsou hlavním měřítkem určujícím výslednou náročnost verifikace.



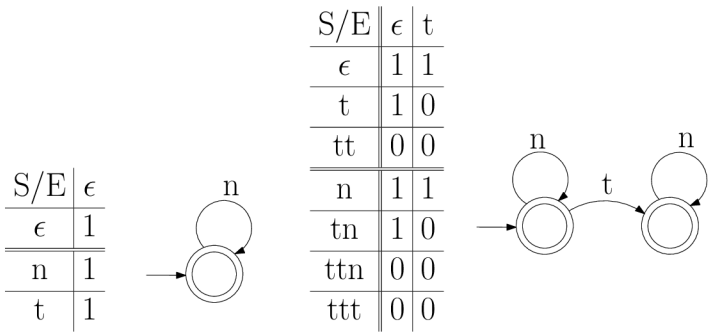
(a) Inicializace (b) 1. krok (c) 2. krok

Obrázek 3.4: Simulace algoritmu verifikace s mapováním neurčitých řetězců na „0“

V druhé demonstraci budeme mít nastavenou konstantu pro neurčité řetězce na „1“, což znamená, že neurčité řetězce budou obsaženy v hledaném invariantu. Simulace běhu je zaznamenána na obrázku 3.5.

Vysvětlení k jednotlivým krokům z obrázku 3.5

(a) incializace algoritmu angluin - $Bad \cap Inv \neq \oslash \Rightarrow$ protipříklad „tt“
(b) 2. krok - invariant nalezen



(a) Inicializace (b) 1. krok

Obrázek 3.5: Simulace algoritmu verifikace s mapováním neurčitých řetězců na „1“

Podívejme se také na operace, které jsme museli vykonat. V inicializaci jsme museli položit 3 dotazy MAT, extrahovat automat z $\mathcal{OT}$ tabulky a položit jeden dotaz ekvivalence. V druhém (posledním) kroce další 4 dotazy MAT, $\mathcal{OT}$ tabulka není konzistentní $\Rightarrow$ dalších 7 dotazů MAT, extrahovat automat a závěrečný dotaz ekvivalence. Celkem jsme tedy potřebovali $3 + 4 + 7 = 14$ dotazů MAT a 2 dotazy ekvivalence.

Porovnáním hodnot obou simulací vidíme, že nám stačilo v druhé simulaci (zahrnování neznámých řetězců do invariantu) o 5 dotazů MAT a jeden dotaz ekvivalence méně. Tento rozdíl se výrazně prohlubuje u složitějších verifikačních problémů a může způsobit výrazné rozdíly v časové náročnosti hledání invariantu. Je ovšem vhodné vyzkoušet obě možnosti nastavení.

Kapitola 4

Optimalizace

4.1 Bierman algoritmus

V návaznosti na algoritmus Angluin máme dánu množinu řetězců, které jsou akceptovány DFA $\mathcal{A}$ a množinu, která je odmítnuta DFA $\mathcal{A}$. Algoritmus dokáže výsledný DFA pouze odvodit z takových množin a neexistuje možnost dotazů na členství (učitele) jako tomu je u algoritmu Angluin. Množina příjímaných a odmítnutaných řetězců je nazývána *vzorky (samples)*.

V uvedené verzi (kapitola 2.4) obsahuje angluinova $\mathcal{OT}$ tabulka pouze hodnoty „0“ a „1“. Navázání na algoritmus Bierman nám umožňuje zavést další stav $\mathcal{OT}$ tabulky, identifikován znakem „?“ o stejném významu jako v algoritmu Biermann - na řetězci nezáleží. Algoritmus Bierman, jak byl definován dříve (kapitola 2.5), pracuje nad vzorky O, které jsou definovány funkcí $0 : \Sigma^* \to \{+, -, ?\}$ určujícím jejich členství ve výsledném DFA $\mathcal{A}$. Pro naše účely relaci přeznačíme na $0 : \Sigma^* \to \{1, 0, ?\}$, kde symbol „1“ indikuje přijetí vzorku (řetězce) ve výsledném DFA $\mathcal{A}$, symbol „0“ odmítnutí a symbol „?“ na řetězci nezáleží.

V naší metodě formální verifikace budeme biermanův algoritmus používat pro extrakci konečného automatu z $\mathcal{OT}$ tabulky.

4.1.1 Nalezení jistě různých stavů

Z algoritmu Angluin máme dánu tabulku $\mathcal{OT} : (U \cap U.\Sigma) \times V \to \{1, 0, ?\}$, ve které jsou dva stavy u a u' *jistě různé*, když $\mathcal{OT}(u)$ a $\mathcal{OT}(u')$ jsou *jistě různé* (definice 11).

4.1.2 Převod CSP na SAT problém

V předchozí části byla rozvedena konstrukce CSP problému a možnosti jeho zefektivnění. Nyní je nutné daný CSP problém efektivně vyřešit. Řešení je postaveno na technikách představených v [8, 13].

V návaznosti na možnosti řešení SAT problémů automatizovanými prostředky budeme převádět uvedený CSP problém na ekvivalentní SAT problém (Boolean Satisfaction Problem) v konjunktivní normální formě (conjunctive normal form, CNF). Čili potřebujeme reprezentovat uvedené omezující podmínky v termech, které budou odpovídat přiřazení hodnot z $[N]$ v CNF formě. Přesněji, do CNF klausulí budeme převádět následující 4 případy:

1. $S_u \in [N]$
2. $S_u \neq S_{u'}$
3. $S_u = S_{u'} \Rightarrow S_{ua} = S_{u'a}$
4. $S_u = i$, pro nějaké $i \in [N]$

Pro kódování uvedených podmínek do SAT máme dvě možnosti - *binární* a *unární kódování*. První z nich, *binární kódování*, umožňuje kódování podmínek do menšího počtu klausulí než *unární kódování*, ale řešení výsledného SAT problému vyžaduje více času[8] a proto pro se budeme dále zabývat pouze *unárním kódováním*. Více informací o *binárním kódování* lze nalézt v [8].

Unární kódování SAT

Cílem je najít předpis pro kódování čtyř výše uvedených možností omezujících podmínek. První možnost (1) říká, že nějaký stav $[N]$ automatu je dosažitelný proměnnou S_u. Pro každou proměnnou S_u alokujeme N logických proměnných $S_u^1, S_u^2, \ldots, S_u^N$, které reprezentují dosažení n-tého stavu. Požadavek $S_u^j = 1$ (j-tý stav je dosažený proměnnou S_u) implikuje $\bigwedge_{k \neq j} S_u^k = 0$ (maximálně jeden stav automatu přijímá řetězec reprezentovaný proměnnou S_u). Uvedenou implikaci pomocí boolovských zákonů převedeme na konjunktivní normální tvar, která společně s podmínkou, že každé proměnné S_u musí být přidělen stav DFA, má tvar

$$(S_u^1 \vee S_u^2 \vee \ldots \vee S_u^N) \wedge \bigwedge_{k \neq j} (\neg S_u^j \vee \neg S_u^k),\ pro\ 1 \leq j \leq N$$

čili pro kódování každého takového pravidla je zapotřebí N^2 klausulí.

Kódování $S_u \neq S_{u'}$ omezení (2) vypadá

$$(\neg S_u^1 \vee \neg S_{u'}^1) \wedge (\neg S_u^2 \vee \neg S_{u'}^2) \wedge \ldots \wedge (\neg S_u^N \vee \neg S_{u'}^N)$$

čili pro kódování těchto omezení potřebujeme N klausulí.

Omezení $S_u = S_{u'} \Rightarrow S_{ua} = S_{u'a}$ (3) je kódováno

$$\begin{gathered}
(\neg S_u^1 \vee \neg S_{u'}^1 \vee S_{ua}^1 \vee \neg S_{u'a}^1) \wedge \\
\vdots \\
(\neg S_u^1 \vee \neg S_{u'}^1 \vee S_{ua}^N \vee \neg S_{u'a}^N) \wedge \\
(\neg S_u^2 \vee \neg S_{u'}^2 \vee S_{ua}^1 \vee \neg S_{u'a}^1) \wedge \\
\vdots \\
(\neg S_u^2 \vee \neg S_{u'}^2 \vee S_{ua}^N \vee \neg S_{u'a}^N) \wedge \\
\vdots \\
(\neg S_u^N \vee \neg S_{u'}^N \vee S_{ua}^N \vee \neg S_{u'a}^N)
\end{gathered}$$

čili pro každý takovýto případ potřebujeme N^2 klusulí.

Omezení (4) říká, že každou *jistě různou* proměnnou S_u z $\mathcal{OT}$ tabulky je možné přímo mapovat na konkrétní stav. Kódování se provede přídáním klauzule $S_u^i \vee \bigvee_{k \neq i} \neg S_u^k, pro\ 1, \leq k \leq N$, kde i označuje stav přijímající S_u.

Nechť n je počet vzorků $\mathcal{D}(O)$ a N počet stavů výsledného DFA $\mathcal{A}$, tak unární kódování CSP podmínek má $\mathcal{O}(n^2N^2)$ klausulí o $\mathcal{O}(nN)$ literálů.

4.1.3 Demonstrace metody s optimalizací Biermanova algoritmu

V této části bude provedena detailní demonstrace algoritmu verifikace s rozšířením o algoritmus Bierman. Specifikace verifikovaného systému i kontrolované vlastnosti je shodné se specifikacemi z kapitoly 3.2.

Simulace běhu je zaznamenána na obrázku 4.1, kde je vždy znázorněna $\mathcal{OT}$ tabulka a z ní extrahovaný konečný automat reprezentující invariant pro každou iteraci metody. Jedná se o grafickou interpretaci průběhu metody s využitím MiniSAT solveru pro řešení SAT problémů. Pokud bylo možné více ohodnocení pro klausuli, byla vybrána právě jedna dle algoritmu solveru MiniSAT.

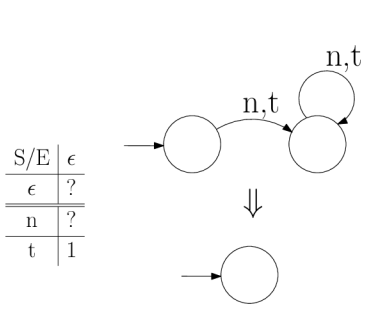
Vysvětlení k jednotlivým krokům z obrázku 4.1

(a) incializace metody - $Init \not\subseteq Inv \Rightarrow$ protipříklad „t“
(b) 1. krok - $\tau(Inv) \not\subseteq Inv \Rightarrow$ protipříklad „nt“
(c) 2. krok - $Bad \cap Inv \neq \oslash \Rightarrow$ protipříklad „ttt“
(d) 3. krok - $Init \not\subseteq Inv \Rightarrow$ protipříklad „tn“
(e) 4. krok - $Bad \cap Inv \neq \oslash \Rightarrow$ protipříklad „ttnt“
(f) 5. krok - invariant nalezen

Podívejme se na operace, které jsme museli vykonat. S každou extrakcí automatu musíme identifikovat 4 druhy omezujících podmínek z $\mathcal{OT}$ tabulky, převést na SAT problém a ten vyřešit. Počet operací v jednotlivých krocích je tedy

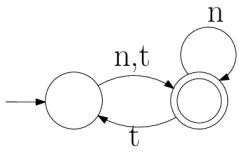
(a) incializace metody - 3 dotazy MAT, extrakce automatu, dotaz ekvivalence
(b) 1. krok - 3 dotazy MAT, extrakce automatu, dotaz ekvivalence
(c) 2. krok - 4 dotazy MAT, extrakce automatu, dotaz ekvivalence
(d) 3. krok - 5 dotazů MAT, extrakce automatu, dotaz ekvivalence
(e) 4. krok - 2 dotazy MAT, extrakce automatu, dotaz ekvivalence
(f) 5. krok - 5 dotazů MAT, extrakce automatu, dotaz ekvivalence

Celkem jsme tedy potřebovali $3 + 3 + 4 + 5 + 2 + 5 = 22$ dotazů MAT a 6 dotazů ekvivalence. Porovnáme-li se simulací z kapitole 3.2, tak vidíme, že jsme v případě využití algoritmu Bierman potřebovali více kroků k nalezení invariantu. Je to dáno tím, že algoritmus pouze splňuje požadavky a hledá libovolné ohodnocení nejmenšího počtu stavů automatu takové, aby klasule byla splněna. V případě, že by se jednalo o složitější problém (extrahovaný automat by měl více stavů), tak by se tolik neprojevil vliv „náhodnosti“ ohodnocení proměnných klauzule a celé řešení by konvertovalo rychleji ke korektnímu invariantu.



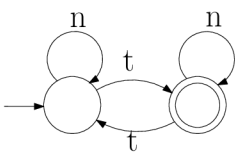
S/E	ϵ
ϵ	?
n	?
t	1

(a) Inicializace



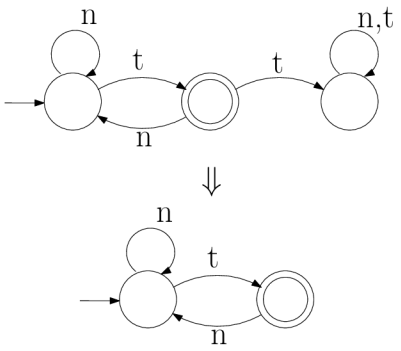
S/E	ϵ
ϵ	?
t	1
n	?
tn	1
tt	0

(b) 1. krok



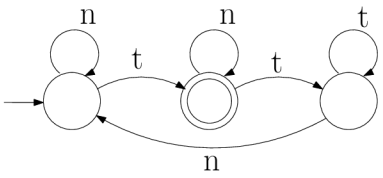
S/E	ϵ
ϵ	?
t	1
n	?
nt	1
nn	?
tn	1
tt	0
ntn	1
ntt	0

(c) 2. krok



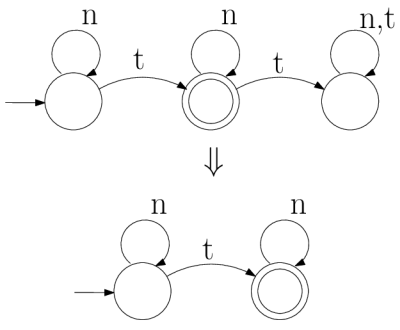
S/E	ϵ
ϵ	?
t	1
n	?
nt	1
tt	0
ttt	0
nn	?
tn	1
ntn	1
ntt	0
ttn	0
tttt	0
tttn	0

(d) 3. krok



S/E	ϵ
ϵ	?
t	1
n	?
tn	1
nt	1
tt	0
ttt	0
nn	?
ntn	1
ntt	0
tnt	0
tnn	1
ttn	0
tttt	0
tttn	0

(e) 4. krok



S/E	ϵ
ϵ	?
t	1
n	?
tn	1
nt	1
tt	0
ttn	0
ttt	0
ttnt	0
nn	?
ntn	1
ntt	0
tnt	0
tnn	1
ttn	0
tttt	0
tttn	0
ttntn	0
ttntt	0

(f) 5. krok

Obrázek 4.1: Demonstrace činnosti algoritmu s využitím Biermanova algoritmu

4.2 Angluin algoritmus s redukovanou tabulkou

Další optimalizací je zavedení redukované tabulky algoritmu Angluin, které zvýší rychlost výpočtu minimalizací počtu výpočtů trasitivních uzávěrů nad jednotlivými řetězci, což je z časového hlediska nejnáročnější operace celé uvedené verifikace.

Algoritmus Angluin, tak jak byl uveden ve svoji původni podobě, obsahuje zpravidla více řádků observation tabulky než je následně stavů ve výsledném DFA. Z toho vyplývá, že více řádků reprezentuje jeden stav. Algoritmus představený v této kapitole je založený na algoritmu *Reduced Observation Table Algorithm*, autoři Rivest a Schapire[14] a se snaží tuto vlastnost minimalizovat.

Většina poznámek k algoritmu Angluin se týká právě rozlehlosti observation tabulky a můžou být přímo vyřešeny pomocí redukované tabulky. Redukovaná observation tabulka $\mathcal{ROT}$ je stejného tvaru jako standardní angluinova tabulka $\mathcal{OT}$, tzn. $\mathcal{ROT} = (\mathcal{S}_R, \mathcal{E}_R, \mathcal{T}_R)$, kde $\mathcal{S}_R, \mathcal{E}_R, \mathcal{T}_R$ jsou stejného významu jako u $\mathcal{OT}$. Rozdíly jsou až v učícím algoritmu, který udržuje konečnou kolekci průzkumů (observations), které jsou uchovávány v redukované observation tabulce $\mathcal{ROT}$. Tabulka je definována následovně

Definice 12 ***(Redukovaná observation tabulka)*** *Redukovaná observation tabulka je trojice $\mathcal{ROT} = (\mathcal{S}_R, \mathcal{E}_R, \mathcal{T}_R)$ nad abecedou Σ, kde*

- $\mathcal{S}_R, \mathcal{E}_R \subseteq \Sigma^*$ *jsou neprázdné konečné množiny,*
- $\mathcal{T}_R : ((\mathcal{S}_R \cup \mathcal{S}_R.\Sigma) \times \mathcal{E}_R) \to \{1, 0\}$ *je funkce udávající příslušnost slova do výsledného DFA,*
- $se = s'e'$ *implikuje* $\mathcal{T}_R(s, e) = \mathcal{T}_R(s', e')$ *pro* $s, s' \in (\mathcal{S}_R \cup \mathcal{S}_R.\Sigma)$ *a všechna* $e, e' \in \mathcal{E}_R$
- $row(s) = row(s')$ *implikuje* $s = s'$ *pro všechna* $s, s' \in \mathcal{S}_R$.

Čili *redukovaná observation tabulka* se liší od *observation tabulky* dvěma způsoby. Jedním je, že množina $\mathcal{S}_R$ nemusí být nutně prefixově uzavřená a druhý, že každý řádek se objeví jednou v horní části tabulky ($row(s)$ takové, že $s \in \mathcal{S}_R$). To znamená, že žádné dva řádky z horní části $\mathcal{ROT}$ se nemůžou mapovat na stejné stavy ve výsledném DFA. V kontextu původního Angluin algoritmu to znamená, že se tabulka nemůže dostat do stavu *nekonzistence*.

Definice *uzavřené* $\mathcal{ROT}$ je stejná jako v původním algoritmu Angluin a z každé *uzavřené* $\mathcal{ROT}$ je možné extrahovat DFA $\mathcal{H}$ stejným způsobem jako v původním Angluin algoritmu.

Zpracování protipříkladu

Nejdříve si zavedeme jazyk U, který bude obsahovat pouze řetězce, které jsou v angluinově T_R tabulce označeny tak, aby byly přijaty výsledným DFA $\mathcal{H}$. Formálněji

Definice 13 ***(Jazyk U)*** *Jazyk U odpovídá observation tabulce T_R, když všechna slova označená „1“ jsou v jazyce U a když všechna slova označená „0“ nejsou v U.*

Pak, pokud je dotazem *ekvivalence* vrácen protipříklad $t = u_i v_i$ délky m, je nutné najít bod zlomu i takový, že $s_i v_i \in U \Leftrightarrow s_{i+1} v_{i+1} \notin U$, kde $s_i = \delta(row(\varepsilon), u_i)$ jsou stavy navštívené přijímáním t v automatu a v_i je korespondující suffix t. Existence takového i je zaručena, jelikož $s_0 v_0 \in U \Leftrightarrow s_m v_m \notin U$, takže sekvenčním průchodem jej najdeme za použití maximálně m dotazů *členství*. Další možností je binární vyhledávání, které najde bod zlomu s $\log m$ dotazy.

Redukovaná tabulka zůstává malá, jelikož není nutné přidávat všechny prefixy protipříkladů. To ale ovšem znamená, že nový automat může stále špatně klasifikovat předchozí protipříklady či protipříklady můžou být vráceny dotazem *ekvivalence* několikrát.

Algoritmus 5 Angluin s redukovanou observation tabulkou

Inicializace $\mathcal{S}$ and $\mathcal{E}$ to $\{\epsilon\}$
Dotazy na členství pro ϵ and každé $a \in \Sigma$
Vytvoření počáteční $\mathcal{OT}$ tabulky $(\mathcal{S}, \mathcal{E}, \mathcal{OT})$
repeat
  while $(\mathcal{S}_R, \mathcal{E}_R, \mathcal{T}_R)$ není uzavřené **do**
    najdi $s \in \mathcal{S}_R$ a $a \in \Sigma$ takové, že
      $row(s.a)$ je různé od $row(s')$ pro každé $s' \in \mathcal{S}_R$
      přidej $s.a$ do S
      rozšiř $\mathcal{S}_R$ na $(\mathcal{S}_R \cup \mathcal{S}_R.\Sigma).\mathcal{E}_R$ použitím dotazů členství
  end while
  Když $(\mathcal{S}_R, \mathcal{E}_R, \mathcal{T}_R)$ je uzavřené, tak $\mathcal{H} = \mathcal{H}(\mathcal{S}_R, \mathcal{E}_R, \mathcal{T}_R)$
  Učitel položí dotaz *ekvivalence* na $\mathcal{H}$
  if Učitel vrátil protipříklad t **then**
    nechť t je $u_i v_i$ délky m, kde $u_0 = v_m = \varepsilon$, $v_0 = u_m = t$
      a $t = u_i a_i v_{i+1}$, pro $i < m$
      Najdi takové i (bod zlomu) pro které platí $s_i a_i v_{i+1} \in U \Leftrightarrow s_{i+1} v_{i+1} \notin U$
    přidej v_{i+1} a všechny jeho prefixy do $\mathcal{E}_R$
    rozšiř $\mathcal{T}_R$ na $(\mathcal{S}_R \cup \mathcal{S}_R.\Sigma).\mathcal{E}_R$ použitím dotazů členství
  end if
until Učitel odpoví „ano“ na dotaz ekvivalence $(\mathcal{L}(\mathcal{H}) = \mathcal{L}\ )$
vrať $\mathcal{H}$

Algoritmus učení $\mathcal{ROT}$ je podobný algoritmu učení $\mathcal{OT}$. Rozdíly jsou pouze v tom, že horní část tabulky $\mathcal{ROT}$ může obsahovat pouze unikátní řádky a tudíž není nutné kontrolovat konzistenci. Výrazná změna se ovšem týká zpracování protipříkladu, kde se hledá bod zlomu (breakpoint) a přidává se jen jeden řádek do horní části tabulky $\mathcal{ROT}$ a tabulka $\mathcal{ROT}$ nemusí být prefixově uzavřena.

4.3 Možnosti výpočtů transitivních uzávěrů τ

Dá se říci, že podstata regulárního model checkingu spočívá ve výpočtu transitivních uzávěrů (případně v tom, jak tyto výpočty zefektivnit či přímo obejít). I v naší metodě je výpočet transitivních uzávěrů klíčovou částí, která nejvíce ovlivňuje celkový výkon metody.

V algoritmu Angluin se převážně jedná o dotazy, zda nějaká konfigurace w je dosažitelná z množiny $Init$, či zda je z konfigurace reprezentované řetězcem w dosažitelná nějaká konfigurace[1] z množiny Bad.

Jelikož je metoda designovaná pro práci s length-preserving transducery, lze výpočet $\tau^*(Init)$ a následný dotaz $w \in^? \tau^*(Init)$ redukovat na dotaz $w \in^? \tau^*(\{w' | w' \in Init \wedge |w'| = |w|\})$. U výpočtu $\tau^*(\{w\})$ a následného $\tau^*(\{w\}) \cap Bad =^? \oslash$ takové zefektivnění nenajdeme.

[1] Pod pojmem konfigurace zde budeme uvažovat řetězec nad Σ, reprezentující danou konfiguraci.

Další možností je předpočítat obecný τ^*, kde ovšem narážíme na dva problémy. První je, že výpočet τ^* je často složitější než $\tau^*(Init)$ a druhý, že τ^* nemusí být regulární. Výhodou (při zachování uvedeného algoritmu) je, že složením (kompozicí) obecného τ^* a množiny konfigurací w lze získat množinu všech dosažitelných konfigurací $range(w \circ \tau^*)$ z konfigurace w za využití zlomku systémových zdrojů oproti výpočtu $\tau^*(w)$.

Další optimalizací, v případě length-preserving transducerů, je možnost „ořezání“ transduceru τ na požadovanou délku τ^N a tím značně snížit komplikovanost výpočtu transitivního uzávěru $(\tau^N)^*$. „Ořezání“ je možné docílit kompozicí s množinou všech slov požadované délky nad abecedou, čili $\tau^N = \Sigma^N \circ \tau$. Takto „ořezaný“ transducer je možné použít i při výpočtech z předcházejícího odstavce.

Kapitola 5

Implementace

V této kapitole provedu rozbor implementace uvedené metody v jazyce Prolog spolu s ostatními použitými technikami použitými při implementaci, kterými jsou knihovna pro práci s konečnými automaty FSA, SAT solver a použité rozhraní mezi jazyky Prolog a C++.

Dlouhou dobu byla metoda implementována v prostředí YAP[1] Prologu, který se ukázal jako velmi vhodný pro implementaci z důvodu jeho minimalizace při zachování dostatečně širokých možností jazyka Prolog a s kvalitní dokumentací. V posledních měsících jsem ovšem musel přejít na SWI Prolog z důvodu zavedení optimalizace pomocí Biermanova algoritmu, které ke své činnosti vyžaduje SAT solver, za který byla zvolena výborná implementace MiniSAT. MiniSAT napsána v jazyce C++[2] a proto nebylo možné prostředí YAP prologu dále používat.

Jako prostředek pro práci s konečnými automaty byl zvolen FSA toolkit, který umožňuje poměrně komfortní práci s konečnými automaty v jazyce Prolog.

5.1 YAP prolog

YAP velmi rychlá implementace jazyka Prolog vyvinutá na LIACC/Universidade do Porto a COPPE Sistemas/UFRJ. Jádro je založeno na WAM (Warren Abstract Machine), s několika optimalizacemi zvyšujícími výkon. YAP je vyvíjen od roku 1985. YAP implementuje většinu z ISO-Prolog standardu a je podporován knihovnou FSA (kapitola 5.3).

Pod YAP Prologem je funkční první varianta verifikační metody (bez biermanovy optimalizace). Mezi možnosti, které nám YAP Prolog poskytuje patří I/O proudy, sockety, moduly, výjimky, debugger, C-rozhraní, dynamický kód, interní database a další... Důvodem, který mi znemožnil další používání této implementace ovšem je, že neexistuje žádné rozhraní do jazyka C++.

Pro zvýšení výkonnosti je vhodné spouštět YAP s parametry definující velikost paměťových alokací, např. příkazem „`yap -h 300000 -s 300000 -t 300000`“.

[1] Yet Another Prolog - minimalistická rychlá implementace, http://www.ncc.up.pt/~vsc/Yap/

[2] Existuje i varianta napsaná v jazyce ANSI C, ale nejedná se o oficiální a ověřenou implementaci.

5.2 SWI Prolog

V současnosti je SWI-Prolog dle počtu stažení nejrozšířenější implementací Prologu. Krom samotného prostředí Prologu nabízí standardně XPCE toolkit, který slouží k tvorbě grafických rozhraní v jazyce Prolog. Společně s XPCE toolkitem je vyvíjen od roku 1987 a dle oficiálních stránek tvrdí, že vývoj je řízen převážně požadavky aplikací.

Krom splnění první části ISO standardu nabízí také poměrně dobrou kompatibilitu s implementacemi YAP, GNU-Prolog a Ciao. Mezi standardně poskytované funkce patří C rozhraní, zpracování unicode znaků, neomezený rozsah celých a reálných čísel (postaven na knihovně GMP), neomezená délka atomických proměnných a další... Další výhodou je existence mnoha balíčků, které výrazně rozšiřují možnosti SWI Prologu. Mezi nejzajímavější balíčky patří

PlUnit - framework pro podporu testování
clib - TCP/IP sockety, Procesy, CGI, MIME,
cpp - C++ rozhraní
JPL - Java rozhraní
SGML - zpracování SGML, HTML, XML
http - podpora HTTP protokolu
ssl - podpora SSL zabezpečené komunikace (OpenSSL)

Pro naše účely je nejzajímavější balík **cpp** zprostředkující rozhraní mezi SWI Prologem a jazykem C++ (kapitola 5.2.1).

V prostředí implementace SWI Prologu je implementována naše verifikační metoda s rozšířením o Biermanův algoritmus.

Pro zvýšení výkonnosti je vhodné spouštět SWI s parametry definující velikost paměťových alokací, např. příkazem „`pl -G0 -T0 -A0 -L0 -O`“ (nejsou nastavené žádné limity).

5.2.1 C++ rozhraní

V této části rozebereme detailněji možnosti balíčku **cpp** zprostředkující rozhraní do jazyka C++. Jedná se o produktivní a velmi dobře navržené rozhraní.

Zdrojový kód 1 Motivační příklad využití C++ rozhraní

```
PREDICATE(hello, 1)
{ cout << „Hello \ << (char *)A1 << endl;
  return TRUE;
}

PREDICATE(add, 3)
{ return A3 = (long)A1 + (long)A2;
}
```

Na motivačním příkladu 1 je ukázáno vytvoření predikátu `hello(+W)`, které vypíše řetězec `Hello <W>` na standardní výstup a predikátu `add(X,Y,Ret)`, který slouží k unifikaci termu `Ret` s hodnotou `X+Y`.

Pro správnou funkci je nutné importovat hlavičkový soubor s deklaracemi funkcí rozhraní `SWI-cpp.h` případně `SWI-Prolog.h` (v závislosti na systému).

Pro práci s termy je v C++ připravena kolekce tříd, které spravují jednotlivé typy termů. Mezi ně patří

PlTerm - základní term v Prologu. Poskytuje konstruktor a operátory pro konverze do C++ datový typů a jejich kontrolu,
PlString - potomek `PlTerm` s konstruktory pro vytváření Prologovských řetězců,
PlCodeList - potomek `PlTerm` s konstruktory pro vytváření seznamů ASCII hodnot,
PlCharList - potomek `PlTerm` s konstruktory pro vytváření seznamů jednoznakových atomů,
PlCompound - potomek `PlTerm` s konstruktory pro vytváření složených termů,
PlTail - potomek `PlTerm` s konstruktory pro tvorbu a čtení prologovských seznamů,
PlAtom - umožňuje velmi rychlou manipulaci s prologovými atomy na jejich interní úrovni,

a další, jejichž popis lze nalézt v oficiálním manuálu. Nejzajímavější je bázová třída `PlTerm` zajištují základní operace nad termy a definující konstruktory pro vytvoření termů ze základních typů C++ (const char *, long, double, void *).

Pro naše účely bude podstatná třída `PlTail` (třída pro manipulaci se seznamy), jelikož např. abeceda je reprezentována seznamem atomů reprezentujících vždy jeden znak nebo třída `PlCompound`, která umožňuje reprezentovat složený term pomocí C++ typu `string`, do kterého bude konstruován generovaný automat metody Bierman.

Pro tvorbu predikátů je předpřipraveno makro `PREDICATE(nazev, pocet_arg)` pomocí kterého je možné v prostředí jazyka C++ nadefinovat nový predikát s daným počtem argumentů, které jsou přístupné z těla predikátu pomocí maker `A<n>`, kde $1 \leq n \leq$ `pocet_arg`.

Zdrojový kód 2 Příklad fragmentu kódu vypisující seznam uložený v termu `A1` na `stdout`

```
PlTail tail(A1);
PlTerm e;

while(tail.next(e))
  std::cout << (char *)e << std::endl;
```

Překlad .cpp souborů a import do SWI

Aby byly vytvořené predikáty dostupné z prostředí SWI prologu je nutné z námi vytvořeného `.cpp` souboru vytvořit sdílenou knihovnu `.so`. Uvedenou operaci lze provést pomocí připravené utility `plld` voláním:

Zdrojový kód 3 Překlad `*.cpp` na sdílenou knihovnu `.so`

```
plld -shared -o library_of_predicates.so source_files.cpp
```

Pokud máme korektně vytvořenou knihovnu s predikáty v jazyce C++ můžeme ji importovat příkazem:

Zdrojový kód 4 Import `.so` knihovny do SWI

```
load_foreign_library('library_name').
```

Další informace o rozhraní SWI Prolog a jazyka C++ lze nalézt v online manulálu [21].

5.3 Finite State Automata Toolbox (FSA)

Finite State Automata Toolbox (FSA) je knihovna predikátů pro práci s regulárními výrazy, konečnými automaty a transducery implementovaných v jazyce Prolog. Mezi základní funkce patří vytvoření automatu z regulárního výrazu, kompozice konečných automatů i konečných transducerů, determinizace konečných automatů i konečných transducerů, tvorba doplňků a průniků apod. Mezi dalšími nabízí i GUI pro práci s konečnými automaty a transducery, které je ovšem vázáno na SICStus Prolog[3]. Zajímavou funkcí je také konverze automatu na odpovídající program v jazyce C. Mezi podporované implementace Prologu patří YAP, SICStus a SWI Prolog.

Snadnost použití FSA toolkitu je ilustrována na příkladu, který demonstruje vytvoření konečného automatu L podle regulárního výrazu a dále vytvoření transduceru na základně regulárního výrazu a přijetí řetězce „ababac“. Poznamenejme, že veškeré exportované predikáty knihovny FSA začínají prefixem „`fsa_`“.

Zdrojový kód 5 Motivační příklad použití FSA

```
| ?- use_module(fsa_library).
...
...
| ?- fsa_regex_atom_compile('[a*,b^,{d,e}]',L).
L = fa(r(fsa_preds),3,[0],[1],[trans(0,a,0),trans(0,b,2),
    trans(0,d,1),trans(0,e,1),trans(2,d,1),trans(2,e,1)],[]) ?
yes
| ?- fsa_regex_transduces('{a:b,? -a}*',„ababac\,L), atom_codes(Atom,L).
L = [98,98,98,98,98,99],
Atom = bbbbbc ?
yes
| ?-
```

5.3.1 Nastavení pomocí globálních proměnných

FSA knihovna umožňuje konfiguraci defaultních nastavení jednotlivých funkcí pomocí tzv. *globálních proměnných*. Možnosti manipulace s globálními proměnnými jsou pomocí následujících predikátů:

[3] Je ovšem nabízena binární varianta FSA, která umožňuje spuštění grafické nádstavby bez instalace SICStus Prologu.

`fsa_global_set(+Key,?Val)` - nastavení globální proměnné `Key` na hodnotu `Val`
`fsa_global_get(+Key,?Val)` - unifikace globální proměnné `Key` s proměnnou `Val`
`fsa_global_list(-List)` - získání seznamu globálních proměnných. V případě že není zadán parametr, tak je seznam vytisknut na std. výstup.

Zvýšení výkonu knihovny

FSA využívá cachování mezivýsledků jednotlivých operací. Nastavení cachování je přístupné pomocí globálních proměnných a jejím vypnutím je možné pro jisté druhy výpočtů zvýšit výkonnost. Pro naše účely se ukázalo vhodnější mít cachování vypnuté, což lze provést pomocí následujících predikátů

```
fsa_global_set(determinize_preds_cache,off).
fsa_global_set(cleanup_list_cache,off).
```

5.3.2 Abeceda

Velmi často diskutovanou vlastností FSA toolkitu je možnost práce s univerzálním symbolem abecedy „?“, který zastupuje právě jeden znak nad libovolnou abecedou. Symbol „?“ je možné s výhodou používat i ve výrazech definující množinu znaků, např. výraz „?-a“ označuje libovolný symbol abecedy vyjma symbol „a“. Výše uvedené vlastnosti jsou dostupné v režimu `fsa_preds`.

Druhým režimem je `fsa_frozen`, který výše uvedené nepovoluje. Defaultním režimem knihovny FSA je režim `fsa_preds` a nastavení je uložené v globální proměnné `pred_module`.

5.3.3 Reprezentace automatů v FSA

Knihovna FSA poskytuje množinu predikátů umožňující práci s automaty. Mezi podporované automaty patří

- konečné automaty
- vážené automaty (weighted recognizers)
- transducery
- vážené transducery (weighted transducers)

Struktura reprezentující automat

Při práci s FSA potřebujeme občas pracovat přímo s datovou strukturou reprezentující konečný automat, případně transducer. Umožní nám to generovat si vlastní automaty jisté struktury a podobně. V našem algoritmu například generujeme automaty přijímající všechna slova konkrétní délky či v algoritmu Bierman je nutné výsledný automat vytvořit na základě získaných údajů z SAT solveru.

Každý automat je reprezentován termem v jazyce Prolog, jehož struktura je následující:

```
fa(Symboly, Stavy, Počáteční stavy, Koncové stavy, Přechody, Skoky)
```

Symboly - je term r(Sig) (konečné automaty) nebo t(SigD,SigR) pro transducery.
Stavy - celé číslo udávající počet stavů automatu.
Počáteční stavy - seřazený seznam celých čísel stavů určující počáteční stavy.
Koncové stavy -seřazený seznam celých čísel stavů určující koncové stavy.

Přechody - seřazený seznam trojic „trans(A,B,C)“, kde A a C jsou celá čísla určující zdrojový a cílový stav a B je symbol.

Skoky - seřazený seznam dvojic „jump(A,B)“, kde A a B jsou celá čísla určující zdrojový a cílový stav ε-přechodů.

Pro zvýšení komfortu zápisu je možné v režimu `fsa_preds` přechod mezi stavy uskutečnitelný přes více znaků abecedy zapsat, místo vyjmenování přechodu pro každý znak, pomocí `in([seznam symbolů abecedy])` či jeho negaci `not_in([seznam symbolů abecedy])`

Na příkladu je ukázán konečný automat odpovídající regulárnímu výrazu „b.b$^+$“.

Zdrojový kód 6 Příklad jednoduchého konečného automatu FSA

```
fa(r(fsa_preds),3,[0],[1],[trans(0,b,2),trans(1,b,1),trans(2,b,1)],[])
```

Pro definice transducerů je možné zvolit zápis přechodu pomocí `trans(0,$@(in([b,c,w]))/ $@(in([b,c,w])),1),`, který definuje přepis symbolů na odpovídajících si pozicích (tzn. v uvedeném jsou kódovány tři možnosti `trans(0,b/b,1)`, `trans(0,c/c,1)` a `trans(0,w/w,1)`). Další možností zápisu je pomocí termu `in([symboly])`, např. `trans(4,in([c00,c10])/e00,5)`.

Zdrojový kód 7 Příklad jednoduchého transduceru FSA

```
fa(t(fsa_preds,fsa_preds),2,[0],[1],
[trans(0,w/c,1),
 trans(0,$@(in([b,c,w]))/ $@(in([b,c,w])),1),
 trans(0,in([c,w])/b,1)],
[]).
```

Export automatu do souboru

Knihovna FSA umožňuje také exportovat zpracovávané automaty do souborů, se kterými je pak možné pracovat pomocí dalších nástrojů (např. grafické rozhraní spustitelné příkazem `fsa -tk`). Pro uložení a načtení automatu do souboru slouží predikáty `fsa_write_file( [+Format,]+File,+Fa)` pro zápis a `fsa_read_file([+Format,]+File,?Fa)` pro čtení. První, nepovinný parametr, slouží k specifikaci formátu automatu. V případě, že není zadaná je bráno defaultní nastavení uložené v globální proměnné `read` nebo `write`.

5.3.4 Možnosti práce s automaty

Je podporována většina základních operátorů regulárních výrazů pro konečné automaty i transducery. Mezi základní funkce patří

- konkatenace, sjednocení, doplněk, rozdíl a průnik
- kompozice, cross-produkt, domain a range, inverze *(transducery)*
- determinizace, minimalizace, odstranění ε-přechodů, operátory regulárních výrazů
- přijetí řetězce automatem, aplikace transduceru na řetězec, generování řetězců přijímaných automatem, generování dvojic řetězců u transducerů

Zdrojový kód 8 Obsah souboru definující konečný automat

```
%% Automatically generated by FSA Utilities.
%% For more info, cf. http://www.let.rug.nl/~vannoord/Fsa/

fa(
%begin sigma and symbols
r(fsa_preds),
%end sigma and symbols
2, % number of states
[ % begin start states
0
], % end start states
[ % begin final states
0,
1
], % end final states
[ % begin transitions
trans(0, b, 0),
trans(0, in([c, w]), 1),
trans(1, w, 1)
], % end transitions
[]). % jumps
```

Základním predikátem zprostředkujícím práci s predikáty je `fsa_regex_compile(+Term,-Fa)`, kde `Term` obsahuje regulární výraz. Predikát kompiluje výraz do termu `Fa` obsahující výsledný automat. Pro konstrukci regulárních výrazů je možné použít zabudované operátory regulárních výrazů. Mezi nejzajímavější z nich patří

fa(`Fa`) - říká, že `Fa` je Term obsahující definici automatu
file(`X`) - říká, že `X` obsahuje název souboru s definicí automatu

union(`E1`,`E2`) - sjednocení, `E1` ∪ `E2`
difference(`E1`,`E2`) - rozdíl, `E1` \ `E2`
concat(`E1`,`E2`) - spojení, `E1`.`E2`
intersect(`E1`,`E2`) - průnik, `E1` ∩ `E2`, `E1` a `E2` musejí být konečné automaty, výsledkem je minimální automat
complement(`E`) - doplněk, ¬`E`

minimize(`E`) - minimalizace `E`
determinize(`E`) - determinizace `E`

compose(`E1`,`E2`) - složení dvou transducerů `E1` (dvojice (A,B)) a `E2` (dvojice (B,C)), výsledkem je minimální automat (transducer) nad dvojicemi (A,C).
range(`E`) - vrací množinu B takových, že dvojice (A,B) jsou v `E`
domain(`E`) - vrací množinu A takových, že dvojice (A,B) jsou v `E`

Na následujícím příkladu 9 je demonstrováno získání rozdílu `E` dvou automatů, `A`\`B`, či zapsáno pomocí průniku `A`∩¬`B`.

Zdrojový kód 9 Příklad práce s automaty pomocí operátorů

```
fsa_regex_compile(difference(fa(A),fa(B)),E)
fsa_regex_compile(intersect(fa(A),complement(fa(B))),E)
```

5.3.5 Další funkce FSA knihovny

Krom práce s automaty nabízí knihovna FSA také dalších 8 ucelých balíků funkcí pro zjednodušení práce se složitějšími datovými strukturami. Konvence pojmenování funkcí v balících je vždy odvozena od jména balíku, který tvoří prefix každé funkce. Jmenovitě se jedná o balíky

fsa_array - neaktualizovatelné velmi efektivní pole,
fsa_m_array - „mutable“ pole,
fsa_u_array - aktualizovatelné pole,
fsa_hash - neaktualizovatelná hash datová struktura,
fsa_m_hash - „mutable“ hash datová struktura,
fsa_u_hash - aktualizovatelná hash datová struktura,
set_bbbtree - Vyvážený binární strom (množina) a
map_bbbtree - Vyvážený binární strom (mapa).

Funkce interně používají stromové struktury pro urychlení přístupu k uloženým položkám. Je vhodné je používat pouze při zpracování velkého množství dat. Pro malé objemy může být režie průchodu stormovou strukturou příliš vysoká.

5.4 MiniSAT solver

MiniSat je minimalistická implementace open-source SAT solveru, vyvinutá pro zjednodušení práce se SAT problémy a pro podporu využívání SAT k řešení problémů. MiniSAT je vyvíjen od roku 2003 a od počátku je velmi zdařilá a často používaná. Cela implementace je velmi malá, čítá kolem 600 řádků kódu (počítáno bez komentářů a prázdných řádků). MiniSAT je poskytován jako třída napsaná v jazyce C++ poskytující rozhraní znázorněné na zdrojovém kódu 10.

Pro standardní použití je možné solver použít následujícím modelovým způsobem. Každá boolean proměnná klauzule se musí nejdříve „zavést“ volaním metody `newVar()`. S pomocí těchto proměnných jsou klazule vytvářeny a přídávany voláním `addClause()`, jež má za parametr `vec<Lit>`, kde `Lit` představuje třídu literálu (boolean proměnné), který je výtvářen pomocí konstruktorů `Lit(var)` nebo `~Lit(var)` (negace). Metoda `addClause()` je schopná detekovat při přidávání triviální konflikty jako klauzule $\{x\}$ a $\{\neg x\}$ a indikuje konflikty návratovou hodnotou `FALSE`. V případě, že došlo ke konfliktu, tak je solver v nedefinovaném stavu a nesmí být použit znovu. Pokud k uvedenému konfliktu nedošlo, tak je hledání řešení spuštěno metodou `solve()`, jež může mít za parametr ohodnocení známých literálů. Tato metoda vrací `FALSE` pokud problém není řešitelný (*unsatisfiable*) nebo `TRUE` řešení existuje (*satisfiable*). Pokud řešení existuje tak je dostupné ve veřejné vlastnosti solveru `vec<Lit> model`.

Zdrojový kód 10 Rozhraní třídy implementující MiniSAT solver

```
class Solver {
    var       newVar    ()
    bool      addClause  (Vec<lit>) literals)
    bool      add...     (...) /*další omezení*/
    bool      simplyfyDB ()
    bool      solve      (Vec<lit> assumptions)
    vec(bool) model      /*pokud je problém řešitelný, tak vektor
                           obsahuje řešení*/
    vec<Lit>  conflict   /*pokud je problém neřešitelný, tak vector
                           obsahuje konfliktní literály*/
}
```

Dále existuje metoda `simplifyDB()`, která může být použita před voláním `solve()` ke zjednodušení interní množiny omezujících podmínek (často nazývána *constraint database*).

Aktuálně jsou k dispozici dvě verze - stabilní a zaběhlá verze 1.14 a nová verze 2[4]. Pro naše účely jsem se rozhodl použít dobře otestovanou verzi 1.14.

Více o použití MiniSATu lze najít v dokumentaci [11].

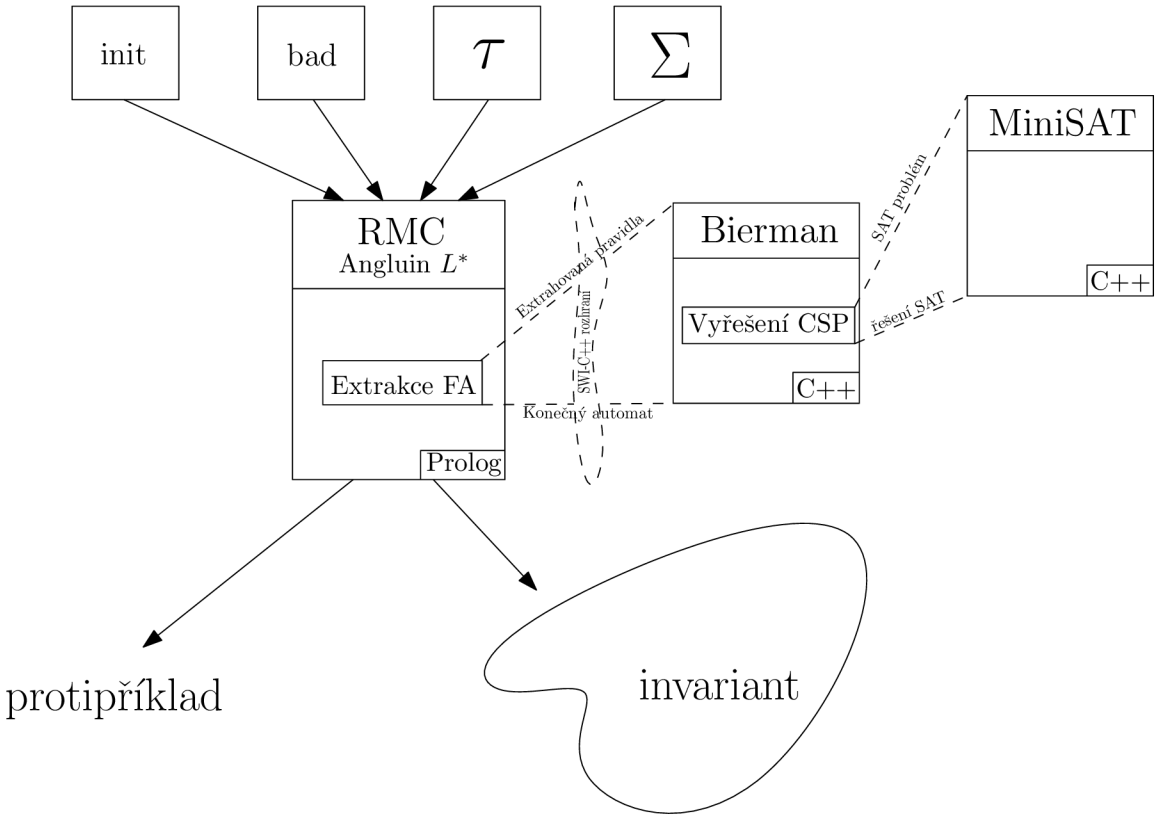
5.5 Popis implementace metody formální verifikace

Implementace metody je provedena v rozsahu základní verze (algoritmus Angluin L^*) a s rozšířením o biermanův algoritmus. V této části se budeme zabývat výhradně druhé, komplexnější variantě, jelikož základní verze je v ní téměř přímo obsažena.

Na obrázku 5.1 je vidět vztah jednotlivých částí implementace a způsob jejich propojení.

Celá implementace se skládá ze dvou zdrojových souborů `angluin.pl`, `bierman.cpp` a adresáře `minisat`, který obsahuje zdrojové soubory MiniSAT solveru. Jelikož je přiložen `Makefile` stačí pro zkompilování `*.cpp` zdrojových souborů zadat příkaz `make` v kořenovém adresáři metody. Spuštění metody verifikace je rozděleno do dvou kroků, dvou příkazů jazyka prolog. Prvním je příkaz `consult(solver)`, pro načtení konfiguračního souburu (více kapitola 5.5.3) a druhým `consult(angluin([OT,Inv]))`, která spouští vlastní metodu. S termem `OT` je univikována observation tabulka a s termem `Inv` automat nalezeného invariantu (pokud je nalezen). Výstupem je soubor `inv_final`, který obsahuje automat Fa, kde $L(Fa) = Inv$ v případě, že byl invariant nalezen. Pokud došlo k porušení verifikované podmínky je příslušný protipříklad uložen do souboru `bad_string` opět ve formě automatu jej přijímají.

[4] Verze 2 je datována 8.12.2006 a není k ní, v době psaní této práce, stále publikována žádná dokumentace.



Obrázek 5.1: Konceptuální schéma implementované metody formální verifikace

5.5.1 Angluin L^* algoritmus

Hlavní část metody je založena na algoritmu Angluin. Jádrem algoritmu Angluin je observation tabulka $\mathcal{OT}$, která je v Prologu reprezentována pěticí `[OTTable, S, E, ELen, SSigma]`, kde

OTTable - `map_bbbtree`, observation tabulka,
S - `set_bbbtree`, množina vzorků $\mathcal{S}$,
E - `fsa_u_array`, množina experimentů $\mathcal{E}$,
ELen - pocet prvků v `E` a
SSigma - `set_bbbtree`, množina vzorků $\mathcal{S}.\Sigma$.

Množiny `S` a `SSigma` spojené v jednu tvoří řádky $\mathcal{OT}$ tabulky. Datový typ $\mathcal{OT}$ tabulky je tvořen výhradně pomocí prostředků poskytovaných knihovnou FSA (kapitola 5.3.5), které by měly zajistit vyšší rychlost v přístupu jednotlivým prvkům.

Dále již pouze vyjmenuji stěžejní predikáty vždy s krátkym popisem.

ot_mat(+W, -RetVal) - MAT, minimální adekvátní učitel
ot_consistent([+OTTable,+S,+EArr,+ELen, +SSigma],-AE) - `AE` obsahuje prvek, který je nutné přidat do množiny experimentů $\mathcal{E}$
ot_closed([+OTTable,+S,+EArr,+ELen, +SSigma],-S1A) - `S1A` obsahuje prvek, který je nutné přidat do množiny vzorků $\mathcal{S}$

angluin_do(+OT,-OTNew) - jedna iterace algoritmu - konzistence a uzavřenost, extrakce Fa, dotaz ekvivalence a případné volání další iterace
angluin_do_while(+OT,-OTNew) - zajišťuje konzistenci a uzavřenost tabulky
chckInv(+Inv, -WRet, -Value,+S) - kontrola nalezeneho invariantu, `Inv` je konečný automat, `WRet` obsahuje případný protipříklad s příslušností danou hodnotou `Value`, `S` je množina $\mathcal{S}$ z tabulky $\mathcal{OT}$

5.5.2 Biermanův algoritmus

Biermanův algoritmus je implementován v jazyce C++ s využitím knihovny MiniSAT pro řešení SAT. Implementace se skládá ze tří částí, kde první je hledání CSP pravidel v jazyce Prolog, druhou je definice rozhraní mezi C++ a SWI Prologem a poslední vyřešení CSP (převod na SAT problém a vyřešení pomocí MiniSAT solveru).

Jako rozhraní z prostřední SWI Prologu je nadefinována následující množna predikátů

bierman_init - inicializace (vytvoření instance třídy) solveru,
bierman_delete - smazání (instance třídy) solveru,
bierman_addConstraintType2(S_u,$S_{u'}$) - omezení $S_u \neq S_{u'}$ (2),
bierman_addConstraintType3(S_u,$S_{u'}$,S_{ua},$S_{u'a}$) - omezení $S_u = S_{u'} \Rightarrow S_{ua} = S_{u'a}$(3),
bierman_addConstraintType4(S_u) - omezení $S_u = i$, pro nějaké $i \in [N]$ (4),
bierman_setSigma(Sigma) - nastavení abecedy (abeceda je předávána jako seznam),
bierman_print - vytiskne nastavení bierman solveru,
bierman_solve(Fa) - spustí samotné řešení,
bierman_addFinalState(S_u) - přidá přijímané vzorky.

Pro správnou funkci by měly být predikáty volány v pořadí

1. `bierman_init/0`
2. `bierman_setSigma/2`
3. `bierman_addFinalState/1`, `bierman_addConstraintType2/2`, `bierman_addConstraintType3/4`, `bierman_addConstraintType4/1`
4. `bierman_solve/1`
5. `bierman_delete/0`

Poslední, třetí částí, je řešení SAT problému implementované pomocí C++ třídy s veřejným rozhraním zobrazeným ve zdrojovém kódu 12. Z metod je nejzajímavější metoda `solve()`, která se stará o vytváření klazule v CNF tvaru a následný pokus o její vyřešení pomocí MiniSAT solveru a konstrukci FSA z nalezeného řešení. Pokud není klauzule řešitelná, tak vygeneruje novou klauzuli pro hledaný DFA s o jedna větším počtem stavů a zkusí ji vyřešit znovu.

5.5.3 Konfigurační soubor `solver.pl`

Jednotlivé verifikační problémy jsou uložené vždy do samostatného adresaře. Ke každému problému (v jeho adresáři) je přiložen soubor `solver.pl`, který obsahuje nastavení pro každého z nich (zavedení predikátů `alph(-X)` `init(-X)`, `bad(-X)` a `tau(-X)` do databáze) . Ukázka obsahu tohoto souboru je vidět v zdrojovém kódu 11 (nastavení verifikace algoritmu Bakery).

Zdrojový kód 11 Obsah souboru `solver.pl` verif. problému Bakery

```
:- reconsult('../angluin.pl').
:- assertz(alph([b,c,w])). %alphabet Sigma
:- fsa_read_file('init',L), assertz(init(L)). %počáteční konfigurace - FSA
:- fsa_read_file('bad',L), assertz(bad(L)). %testovaná vlastnost - FSA
:- fsa_read_file('trs',L), assertz(tau(L)). %chování - transducer
```

5.5.4 Problémy s implementací

Při implementaci jsem se setkal s četnými problémy, které se v první fázi týkaly samotného programování v jazyce Prolog, se kterým se mi napracovalo příliš dobře. Tyto problémy se ovšem po nějaké době podařilo překonat a vyvstaly problémy nové, týkající se převážně knihovny FSA ve spojení s prostředím jazyka Prolog. Knihovna FSA je napsána velmi dobře, ale přeci jen je vidět, že ještě není zcela odladěna a jsou chvíle kdy se chová dosti nepředvídatelně. Při práci s jednoduchými (malými) automaty je vše v pořádku, ale v případě větších (např. 300 a více stavů) automatů se nemusí chovat zcela korektně. Jedná se například o operátor `cleanup()` či pokud se v některých výrazech použije či nepožije minimalizace... Posledně zmíněné vlastnosti knihovny FSA způsobovaly nelogické pády celého řešení, které dokázalo značně pozdržet celý vývoj.

Zdrojový kód 12 Veřejné rozhraní třídy implementující řešení CSP omezení

```
class CBierman{
  public:
    /*nese vyslednou definici automatu kompatibilní s knihovnou FSA*/
    stringstream fa;

    CBierman();
    CBierman(int);
    ~CBierman();

    /*zkonstruuje SAT klauzuli v CNF formě, vyřeší a zkonstruuje
      automat jako řetězec v proměnné fa*/
    void solve();

    /*přidá proměnnou*/
    void addVariable(PlTerm);

    /*nastavi abecedu použitou konečnými automaty*/
    void setSigma(PlTerm);

    /*přidá stav do množiny koncových stavů*/
    void addFinalState(PlTerm);

    /*přidání CSP podmínky typu 2*/
    void addConstraintType2(PlTerm, PlTerm);

    /*přidání CSP podmínky typu 2*/
    void addConstraintType3(PlTerm, PlTerm, PlTerm, PlTerm);

    /*přidání CSP podmínky typu 4*/
    void addConstraintType4(PlTerm);

    /*vytiskne stav solveru na stdout*/
    void printState();
};
```

Kapitola 6

Experimety

V první části této kapitoly budou představeny typické problémy regulárního model checkingu. V druhé části budou rozebrány experimenty, prováděné nad některými z uvedených problémů a také bude zmíněn rozdíl ve výkonnosti vlivem optimalizace zavedením Biermannova algoritmu.

6.1 Bakery algoritmus (Bakery Algorithm)

V Bakery algoritmu pro výlučný přístup (mutual exclusion) existuje současně neomezený počet procesů čekajících na získání „tiketu“ pro vstup do kritcké sekce (KS). Každý proces, který chce vstoupit do kritické sekce získá tiket, jehož číslo je nejvyšší číslo ostatních čekajících tiketů plus jedna. Když proces má nejmenší číslo z ostatních čekajících procesů, může vstoupit do kritické sekce, pokud z ní předchozí (jehož tiket byl o jedno menší) vystoupí a později při opouštění kritické sekce zahodí svůj tiket.

6.2 Szymanski algoritmus (Szymanski's Algorithm)

V Szymanski algoritmu pro výlučný přístup existuje opět neomezený počet procesů organizovaný v lineárním poli, kde index v poli značí ID procesu. Každý proces obsahuje dvě boolovské proměnné a jeden kontrolní stav, označující v jakém stavu vůči KS se nachází.

Algoritmus 6 Szymanski algoritmus

1: **await** $\forall j : j \neq i : \neg s[j]$
2: $w[i], s[i] \Leftarrow true, true$
3: **if** $\exists j : j \neq i : (pc[j] \neq 1) \wedge (pc[j] \neq 2)$ **then**
4: $\quad s[i] \Leftarrow false$; goto 8
5: **else**
6: $\quad w[i] \Leftarrow false$; goto 11
7: **end if**
8: **if await** $\exists j : j \neq i : s[j] \wedge \neg w[j]$ **then**
9: $\quad w[i], s[i] \Leftarrow false, true$
10: **end if**
11: **await** $\forall j : j \neq i : \neg w[j]$
12: **await** $\forall j : j < i : \neg s[j] \vee \neg w[j]$
13: $s[i] \Leftarrow false$; goto 1

V algoritmu kontrolní proměnná (pc) nese informaci o stavu procesů tak, že je v ní uloženo číslo řádku, ve kterém se proces aktuálně nachází a proměnná i nese ID procesu. Podíváme-li se na algoritmus, tak můžeme vidět, že proces může být ve čtyřech kategorických stavech. Nejdříve v inicializační části (řádky 1 a 2), pak se provede analýza okolních procesů (řádky 3 až 7), čekání (8-12) a dále pak již následuje jen vstup do kritické sekce.

6.3 Dijkstrův algoritmus (Dijkstra's Algorithm)

Dijkstrův algoritmus pro zajištění výlučeného přístupu neomezeného počtu procesů, řazených v lineárním poli, kde index v poli je ID procesu. Každý proces má přiřazený $flag$, který signalizuje jeho schopnost vstoupit do kritické sekce.

Algoritmus 7 Dijsktrův algoritmus

```
1:  flag[i] ⇐ 1
2:  if p ≠ i then
3:     await flag[p] = 0 then
4:        p ⇐ i
5:  end if
6:  flag[i] ⇐ 2
7:  if ∃j ≠ i : flag[j] = 2 then
8:     goto 1
9:  end if
10: flag[i] ⇐ 0
11: goto 1
```

V algoritmu značí proměnná p ID procesu aktuálně nacházejícího se v kritické sekci a $flag$ je pole indexované ID procesů a nabývajících hodnot $0, 1, 2$, kde hodnota 0 signalizuje, že proces nečeká na vstup ani není v KS, 1 znamená, že proces čeká na vstup do KS a 2, že se nachází v KS.

6.4 Výsledky

V této části shrnu výsledky z experimentů a podíváme se na komplikace, které vyvstaly se složitějšími problémy. Uvedené časy jsou měřeny na počítači s konfigurací 2GHz@1.2GB RAM, Linux.

Prvním problémem, na kterém byla implementace zkoušena, byla verifikace algoritmu Bakery. Časové náročnosti jsou znázorněné v tabulce 6.1. Z tabulky je zajímavá časová náročnost metody s optimalizací Bierman, která zabrala nejvíce času. Je to způsobeno tím, že bylo stanoveno málo CSP podmínek (většina řetězců byla klasifikována jako „?“) a tak odvozený automat (s jistou náhodností ohodnocení, protože SAT problém byl uspokojen při větším množství hodnot jednotlivých literálů) nebyl odvozen „ideálně“.

metoda	Prolog	čas (s)
Angluin/0	YAP	3.468
Angluin/1	YAP	0.684
Bierman	SWI	5.64

Tabulka 6.1: Časové statistiky verifikace algoritmu Bakery

Při experimentech s Dijsktrovým a Szymanskiho algoritmem dochází k chybě při výpočtu predikátu `fsa_regex_compile(mb(intersection(fa(CInv), fa(TauInv))), TauInvISCInv)`, který slouží k následnému zjištění, zda $\tau(Inv) \subseteq^{?} Inv$ (převodem na $\neg Inv \cap \tau(Inv) =^{?} \oslash$). Predikát `fsa_regex_compile/2` pochází z FSA knihovny a zajímavým aspektem je, že pokud si zpracovávané automaty (`CInv` a `TauInv`) uložím samostatně a poté zkusim operaci provést samostatně mimo běh metody, tak se vše provede korektně. Bohužel se mi nepovedlo uvedenou chybu (jedná se o přetečení zásobníku) odstranit a tudíž není implementace na verifikaci těchto problémů funkční.

metoda	Prolog	čas (min)
Angluin/0	YAP	48+
Angluin/1	YAP	53+
Bierman	SWI	28+

Tabulka 6.2: Časové statistiky verifikace algoritmu Szymanski

V tabulce 6.2 jsou uvedeny naměřené časy strávené řešením verifikace Szymanskiho algoritmu, než dojde k pádu implementace metody. Symbol „+“ znamená, že verifikace skončila s chybou v uvedeném čase.

metoda	Prolog	čas (min)
Angluin/0	YAP	41+
Angluin/1	YAP	33+
Bierman	SWI	31+

Tabulka 6.3: Časové statistiky verifikace Dijkstrova algoritmu

V tabulce 6.3 jsou uvedeny naměřené časy strávené řešením verifikace Dijkstrova algoritmu, než dojde k pádu implementace metody.

Podíváme-li se do tabulek (přestože nevidíme konečné časy), tak dosažené časy před pádem jsou příliš vysoké[1]. Hlavní problém této metody je v počítání příliš vysokého počtu tranzitivních uzávěrů (konkrétně se jedná o počet $|\mathcal{S}|*|\mathcal{E}|$, kde $\mathcal{S}, \mathcal{E}$ jsou množiny z $\mathcal{OT}$), což u testovaných algoritmů znamená řádově stovky až tisíce výpočtů tranzitvních uzávěrů. Pro představu např. výpočet tranzitivního uzávěru pro řetězec délky 4 u algoritmu Szymanski zabere i 40s a s delšími řetězci narůstá.

[1] V porovnání s konkurenčními metodami, např. [9]

Kapitola 7

Závěr

Práce se zabývá regulárním model checkingem a je v ní představena nová metoda formální verifikace založená na odvozování regulárních jazyků, která slouží k dokázování správnosti nekonečně stavových systémů.

V prvních kapitolách je představen úvod do teorie modelování systémů a (regulárního) model checkingu. Tyto kapitoly mohou složit jako úvodní text do problematik model checkingu v českém jazyce, kterých není napsáno mnoho. V dalších kapitolách je představena nová metoda regulárního model checkingu, založená na algoritmech Angluin a Bierman a provedena její implementace.

Jádrem celé metody je algoritmus Angluin, který při implementaci ukázal i slabé stránky metody. Mezi ně patří hlavně definice učitele, který musí pro každý zkoumaný řetězec spočítat tranzitivní uzávěr a právě tyto výpočty tranzitivních uzávěrů způsobují příliš velkou časovou náročnost celé metody. Možností je zavedení dalších optimalizací (např. algoritmus Angluin s redukovanou observation tabulkou), které by mohly tuto příliš velkou časovou náročnost snížit. Druhou stránkou je optimalizace za pomoci algoritmu Bierman, která za jistých okolností (v tabulce $\mathcal{OT}$ se vyskytuje příliš mnoho neklasifikovaných řetězců) může způsobit vyšší počet iterací metody a tím i nezanedbatelně zvýšit výsledný čas potřebný pro verifikaci.

Byly provedeny dvě implementace, kde první je tvořena pouze algoritmem Angluin a druhá optimalizovaná algoritmem Bierman. První varianta umožňuje nastavení konstanty pro neklasifikované řetězce určující, zda tyto řetězce budeme zahrnovat nebo nezahrnovat do hledaného invariantu. Předpokládalo se, že implementace s optimalizací bude rychlejší než bez ní, ale experimenty ukázaly, že není možné říct, která varianta metody vykazuje vždy nejlepší výsledky. Pro každý verifikovaný problém je tedy vhodné vyzkoušet veškeré varianty a zjistit, která daný problém řeší nejrychleji.

Implementace je napsána převážně v jazyce Prolog a postavena na funkcích z knihovny FSA, která umožňuje jednoduchou práci s konečnými automaty. Knihovna je napsána velmi dobře (přestože se jedná o práci jediného tvůrce), ale jelikož není masově používána, není zcela odladěna. Výhodou ovšem je, že autor je přístupný a hlášené chyby se snaží rychle opravovat[1]. Implementace není zcela dokončena, což je způsobeno částečně problémy s

[1] V rámci práce byla nahlášena chyba, která byla téměř okamžitě opravena a vydána nová verze knihovny FSA (ver. fsa6-301).

knihovnou FSA a také několika záludnými chybami v implementaci, které velmi pozdržely vývoj.

V současné době implementace funguje korektně pouze na jednoduché problémy a bylo by vhodné v rámci dalšího vývoje implementaci doladit pro verifikaci i komplikovaných problémů. Dále lze prověřit vliv dalších optimalizací na výslednou časovou náročnost řešení. Jedná se převážně o implementaci algoritmu Angluin s redukovanou tabulkou[14] a zavedení učení přes diskriminační stromy[6].

Literatura

[1] Vardhan Abhay. *Learning To Verify Systems.* 2006.

[2] Smrčka Aleš. Formální verifikace. http://www.fit.vutbr.cz/ smrcka/fav/guide/.cs, 2007.

[3] Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[4] Raffelt Harald Berg Therese. Model checking. In Broy Manfred [6], pages 557–603.

[5] Feldman J.: Biermann A. On the synthesis of finite state machines from samples of their behavior. *IEEE Transactions on Computers*, 21(6):592–597, 1972.

[6] Katoen Joost-Pieter Leucker Martin Pretschner Alexander Broy Manfred, Jonsson Bengt, editor. *Model-Based Testing of Reactive Systems, Advanced Lectures [The volume is the outcome of a research seminar that was held in Schloss Dagstuhl in January 2004]*, volume 3472 of *Lecture Notes in Computer Science*. Springer, 2005.

[7] Stephen A. Cook. The complexity of theorem-proving procedures, 1971.

[8] Olga Grinchtein, Martin Leucker, and Nir Piterman. Inferring network invariants automatically. In *Proceedings of the 3rd International Joint Conference on Automated Reasoning (IJCAR'06)*, volume 4130 of *Lecture Notes in Artificial Itelligence*, September 2006.

[9] P. Habermehl and T. Vojnar. Regular model checking using inference of regular languages. http://citeseer.ist.psu.edu/habermehl04regular.html, 2004.

[10] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient (sat) solver. http://citeseer.ist.psu.edu/moskewicz01chaff.html, 2001.

[11] Niklas Sörensson Niklas Een. An extensible sat-solver [ver 1.2].

[12] Marcus Nilsson. Regular model checking, 2000.

[13] Arlindo L. Oliveira and Joao P. Marques Silva. Efficient algorithms for the inference of minimum size DFAs. *Machine Learning*, 44(1/2):93–119, 2001.

[14] Schapire R.E. Rivest R.L. Inference of finite automata using homing sequences. *Information and Computation*, pages 299–347, 1993.

[15] Rodosek. A new approach on solving 3-satisfiability. In *AISMC: International Conference on Artificial Intelligence and Symbolic Mathematical Computing*, pages 197–212, 1996.

[16] S. K Shukla. HORNSAT, model checking, verification, and games. http://citeseer.ist.psu.edu/shukla95hornsat.html, 28 1995.

[17] Rasmus Aslak Kjaer Thomas Christensen, Morten Gade. Optimal scheduling using sat-solving and bounded model checking. http://www.1-1link.dk/1.asp?dat4-ps=&link=13432, 2004.

[18] B. A. Trakhtenbrot and Y. A. Barzdin. *Finite Automata: Behavior and Synthesis.* North-Holland, 1973.

[19] Antti Valmari. The state explosion problem. In *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets*, pages 429–528, London, UK, 1998. Springer-Verlag.

[20] Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics.* Elsevier and MIT Press, 1990.

[21] Jan Wielemaker. A c++ interface to swi-prolog. http://www.swi-prolog.org/packages/pl2cpp.html.