VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

ROZHODOVÁNÍ WS1S POMOCÍ SYMBOLICKÝCH AUTOMATŮ

DECIDING WS1S WITH AUTOMATA

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. PAVEL BEDNÁŘ**
AUTHOR

VEDOUCÍ PRÁCE **doc. Mgr. LUKÁŠ HOLÍK, Ph.D.**
SUPERVISOR

BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání diplomové práce

144803

Ústav:	Ústav inteligentních systémů (UITS)
Student:	**Bednář Pavel, Bc.**
Program:	Informační technologie a umělá inteligence
Specializace:	Matematické metody
Název:	**Rozhodování WS1S pomocí symbolických automatů**
Kategorie:	Algoritmy a datové struktury
Akademický rok:	2022/23

Zadání:

Cílem práce je implementovat nástroj pro rozhodování logiky WS1S, reprodukovat a rozvinout implementační techniky nástroje MONA, a pokusit se o novou reprezentaci automatů, kde uzly rozhodovacích diagramů, použitých v nástroji MONA, jsou stavy automatu.

1. Nastudujte algoritmy pro rozhodování logiky WS1S.
2. Navrhněte reprezentaci automatů vycházející z nástroje MONA, kde ale BDD budou součástí automatu (uzly BDD budou stavy).
3. Na základě navržené reprezentace a nástroje MONA implementujte vlastní nástroj.
4. Pracujte na efektivitě nástroje.
5. Porovnejte efektivitu výsledného nástroje s nástrojem MONA.

Literatura:

1. Klarlund, N., Møller, A., Schwartzbach, M.I. (2001). MONA Implementation Secrets. In: Yu, S., Păun, A. (eds) Implementation and Application of Automata. CIAA 2000. Lecture Notes in Computer Science, vol 2088. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-44674-5_15
2. Fiedor, T., Holík, L., Janků, P., Lengál, O., Vojnar, T. (2017). Lazy Automata Techniques for WS1S. In: Legay, A., Margaria, T. (eds) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2017. Lecture Notes in Computer Science(), vol 10205. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-662-54577-5_24

Při obhajobě semestrální části projektu je požadováno:
1, 2, ideálně část 3.

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

Vedoucí práce:	**Holík Lukáš, doc. Mgr., Ph.D.**
Vedoucí ústavu:	Hanáček Petr, doc. Dr. Ing.
Datum zadání:	1.11.2022
Termín pro odevzdání:	17.5.2023
Datum schválení:	3.11.2022

Abstrakt

WS1S je druhořádová logika s jednoduchou syntaxí a sémantikou, nabízející velkou stručnost popisu a rozhodnutelnost pomocí konečných automatů. Bohužel složitost rozhodovací procedůry je *nonelementary*, což na jednu stranu evokuje problém pro praktické aplikace, ale zároveň to představuje určitý prostor pro různé heuristiky a optimalizace. Nejpoužívanější nástroj v této oblasti, Mona, představuje ukázku toho, že konečné automaty s velkými abecedami mohou pracovat efektivně, a to díky BDD kódování přechodů. V této práci představíme novou rozhodovací procedůru pro WS1S, která zkombinuje klasický přístup s Mona přístupem tak, že přechody reprezentované pomocí BDD integrujeme přímo do automatu. Tím dosáhneme efektivity BDD přechodů, ale zároveň flexiblity díky používání čistě jen automatů a navíc oproti BDD můžeme přeskočit více proměnných nebo pracovat s nedeterminismem. Experimenty ukazují, že jsme schopni v některých oblastech konkurovat nástroji Mona a také dokážeme obecně tvořit automaty s méně stavy než Mona.

Abstract

WS1S is second-order logic with simple syntax and semantics, offering great brevity and decidability using finite automata. Unfortunately, the complexity of the decision procedure is *nonelementary*, which may sounds like a problem for practical applications, but on the other hand it represents a certain space for various heuristics and optimizations. The most widely used tool in this field, Mona, is a demonstration that finite automata with large alphabets can work efficiently, thanks to BDD encoding of transitions. In this work, we present a novel decision procedure for WS1S that combines the classical approach with the Mona approach by integrating the transitions represented by BDD directly into the automaton. In this way, we achieve the efficiency of BDD transitions, but at the same time flexibility thanks to the use of automata only and in addition, compared to BDD, we can skip more variables or work with non-determinism. Experiments shows that we are able to compete with the Mona in some areas and we are also able to create automata with fewer states than Mona in general.

Klíčová slova

rozhodování WS1S, konečné automaty, binární rozhodovací diagramy

Keywords

deciding WS1S, finite automata, binary decision diagrams

Citace

BEDNÁŘ, Pavel. *Rozhodování WS1S pomocí symbolických automatů*. Brno, 2023. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce doc. Mgr. Lukáš Holík, Ph.D.

Rozhodování WS1S pomocí symbolických automatů

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením doc. Mgr. Lukáše Holíka, Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Pavel Bednář
17. května 2023

Poděkování

Chtěl bych poděkovat vedoucímu práce doc. Mgr. Lukáši Holíkovi, Ph.D. za odborné vedení, cenné rady a výborný přístup. Dále děkuji Ing. Tomáši Fiedorovi, Ph.D. za poskytnutou sadu WS1S formulí k evaluaci.

Obsah

Kapitola 1

Úvod

Logika obecně hraje podstatnou roli v mnoha oblastech informatiky. Například ve formální verifikaci, konkrétně v model checkingu specifikujeme vlastnosti modelovaného systému pomocí CTL* logiky, popřípadě nějaké její vhodné podmnožiny. Dále třeba v oblasti theorem provingu existuje několik automatizovaných rozhodovacích procedůr pro různorodé rozhodnutelné teorie. Velmi známé a hojně používané rozhodovací procedůry jsou SAT a SMT solvery. SAT (satisfiability) solvery mají na vstupu formuli ve výrokové logice a odpovídají na otázku, zda-li je tato formule splnitelná. SMT (satisfiability modulo theories) solvery, jak již název napovídá, rozšiřují SAT solvery pro různé prvořádové formule s rovností a atomy z různých prvořádových teorií (někdy se povolují i kvantifikátory). Existují ovšem i rozhodnutelné logiky s vyšším řádem, jako jsou WS1S, WS2S, S1S nebo MSO.

Weak (množiny jsou konečné) monadic (relace jsou unární, tedy jsou množiny) Second-order (kvantifikování nad množinami) theory of 1 Successor (existuje jen jeden následník, tedy struktury jsou lineární) má velmi jednoduchou syntax a sémantiku, jedná se o variaci predikátové logiky s prvořádovými proměnnými, které představují přirozená čísla a druhořádovými proměnnými, které značí konečné množiny přirozených čísel s funkcí následníka a operátory porovnání [8]. Již před dlouhou dobou bylo objeveno, že WS1S je rozhodnutená pomocí konečných automatů [3]. Klíčová myšlenka je rekurzivně převádět každou podformuli hlavní formule na deterministický konečný automat, který reprezentuje množinu splnitelných interpretací. Automatové operace, jako jsou průnik nebo sjednocení, poté slouží k realizování booleovských spojek.

WS1S nabízí velkou stručnost, ovšem je to vykoupeno *nonelementary* složitostí, což se na první pohled může zdát jako podstatná překážka pro praktické aplikace. Je zde tedy velký prostor pro nové optimalizace a heuristiky, které přiblíží kompaktnost WS1S rychlosti ostatních reprezentací [7]. Takové techinky byly implementovány nástrojem Mona [11], jenž je aktuálně nejznámnější a nejúspěšnější rozhodovací procedůrou pro WS1S a WS2S. Nástroj Mona si díky jeho efektivitě našel využití v mnoha aplikacích z oblasti verifikace programů s komplexními dynamicky propojenými datovými strukturami [15, 13], řetězcových programů [17], parametrických systémů [2], distribuovaných systémů [12], hardwarové verifikace [1] nebo automatizované syntézy [16]. Mezi další nástroje patří Gaston [6], kde hlavní novinkou je testování prázdnosti jazyka během výpočtu formule a ořezávání stavového prostoru o části irelevantní pro test, jMosel [18] pro logiku M2L(str), rozhodovací procedůra implementovaná automatovou knihovnou z [4] používající symbolické konečné automaty nebo Koalgebraická rozhodovací procedůra [19] založená na testování ekvivalence dvojice formulí nalezením bisimulace mezi jejími deriváty.

V této práci představíme novou rozhodovací procedůru pro WS1S, která bude vycházet z úspěšného přístupu nástroje Mona, ale s využitím speciálně upravených automatů namísto BDD. Pro náš algoritmus použijeme flexibilitu z klasického přístupu odpovídajícímu běžnému automatu s abecedou bitových vektorů, které nahradíme integrováním efektivního BDD kódování bitových vektorů z nástroje Mona přímo do automatu. Získáme tak přímočarou reprezentaci, ve které manipulujeme pouze s automaty, odstraníme limity dané používáním BDD (problémová je např. reverzace, algoritmy simulace) a zároveň pracujeme s kompaktní abecedou o dvou symbolech $0, 1$.

Naše implementace je postavena na automatové knihovně Mata. Hlavní změna oproti klasickým automatům je ta, že místo běžných hran používáme skip hrany, jenž intuitivně drží informaci kolik proměnných přeskakujeme, což odpovídá přidružení proměnné vnitřním uzlům BDD. Konkrétněji, přidáváme ke každé hraně informaci o její délce, kde hrana délky 1 je speciální případ odpovídající běžným automatovým hranám, tedy do jazyka automatu přidá symbol přidružený k tomuto přechodu a hrana délky n, $n > 1$ znamená, že do jazyka bude přidán symbol přidružený k tomuto přechodu a navíc $n - 1$-krát všechny symboly abecedy. Výhoda je, že jsme schopni přeskočit i běžný stav automatu, pokud není koncový a všechny hrany z něj směřují do stejného stavu. Pro tento přístup jsme upravili automatové algoritmy jako je průnik, determinizace, komplement, projekce a minimalizace, aby vhodně pracovali s námi představenou reprezentací.

Následně jsme provedli porovnání s nástrojem Mona, a to po stránce výpočetního času a počtu generovaných stavů. Časové výsledky jsou negativně ovlivněny použitým Brzozowski algoritmem, který během minimalizace často stavově exploduje. Nabízíme i srovnání bez výpočetního času minimalizace, jenž indikuje, že kdybychom změnili minimalizační algoritmus, mohli bychom se dostat velmi blízko k výsledkům nástroje Mona. Dále jsme experimentálně ověřili, že jsme schopni vytvářet méně stavů (v kontextu nástroje Mona rozumíme stavem součet stavů automatu a BDD uzlů), což je zapříčiněno odlišnou reprezentací automatů a možností přeskakovat běžné stavy automatu.

Rozdělení kapitol v textu je následující. Kapitola 2 pojednává o teoretických základech formálních jazyků a automatů, WS1S a nástroji Mona. Poté kapitola 3 představuje navrhovaný algoritmus, na což navazuje kapitola 4, kde je popsána jeho implementace. Následují experimenty v kapitole 5 a posléze v kapitole 6 porovnání a zhodnocení navrhovaného algoritmu.

Kapitola 2

Automaty a WS1S logika

V této části popíšeme základní teoretické pojmy z teorie formálních jazyků a WS1S logiky nutné k dalšímu porozumnění problematiky.

2.1 Formální jazyky a automaty

V této sekci zmíníme základní definice spjaté s formálními jazyky a automaty, jako jsou abecedy, slova, jazyky a konečné automaty. Některé pojmy jsou zde definované pomocí rekurzivní definice. Všechny definice z této sekce jsou převzaty z [14].

2.1.1 Abecedy a slova

Prvním potřebných teoretickým aparátem je abeceda a slovo. Zde ukážeme jejich definice, příklady a vybrané operace.

Definice 2.1. Abeceda je konečná, neprázdná množina elementů, které se nazývají symboly.

Příklad 2.1. Například $\Sigma = \{a, b, c\}$ je abeceda.

Symbol je nejmenší jednotka, kterou uvažujeme. Symboly mohou být třeba písmena přirozeného jazyka, tedy klasické abecedy (např.: *a, b, c, ...*), nebo prvky binárního jazyka (*0, 1*), nebo také příkazy nějakého programovacího jazyka (např.: *int, id, =, +, {, }, ;, ...*).

Definice 2.2. Nechť Σ je abeceda. **Slovo** nad abecedou Σ definujeme následovně:

1. ϵ je slovo nad abecedou Σ.
2. Pokud x je slovo nad abecedou Σ a $a \in \Sigma$, pak xa je slovo nad abecedou Σ.

Příklad 2.2. Například ab je slovo nad abecedou $\Sigma = \{a, b, c\}$.

Slovo je tedy sekvence symbolů. V teorii formálních jazyků má slovo ještě druhý ekvivalentní pojem, a to řetězec. Prázdné slovo, nebo-li slovo obsahující žádný symbol, je značeno ϵ. Na řetězcích jsou definovány operace jako je třeba konkatenace, iterace, reverzace, podslovo, prefix, suffix.

Definice 2.3. Nechť x je slovo nad abecedou Σ. **Délka slova** x, $|x|$, je definována takto:

- Pokud $x = \epsilon$, pak $|x| = 0$.

- Pokud $x = a_1...a_n$ pro nějaké $n \geq 1$, kde $a_i \in \Sigma$ pro všechna $i = 1, ..., n$, pak $|x| = n$.

Příklad 2.3. Například $|ab| = 2$, $|\epsilon| = 0$.

Neformálně řečeno, délka slova je rovna počtu jeho symbolů.

Definice 2.4. Nechť x a y jsou dvě slova nad abecedou Σ. Pak xy je **konkatenace** x a y.

Příklad 2.4. Například konkatenace slov ab a ac je $abac$.

Dále pro každé slovo platí, že $x\epsilon = \epsilon x = x$. To znamená, že ϵ je neutrální prvek vůči konkatenaci.

Definice 2.5. Nechť x je slovo nad abecedou Σ. Pro $i \geq 0$ je i-tá **iterace** x, značeno jako x^i, definována následujícím způsobem:

- $x^0 = \epsilon$.
- $x^i = xx^{i-1}$, pro $i \geq 1$.

Příklad 2.5. Například druhá iterace slova ab, ab^2, odpovídá slovu $abab$.

Lze vypozorovat, že pro každé slovo x platí vztah $x^i x^j = x^j x^i = x^{i+j}$, kde $i \geq 0$ a $j \geq 0$.

2.1.2 Jazyky

Dalším logickým krokem, poté co máme abecedu a slova, jsou jazyky, tvořené právě z těchto slov. Uvažujme abecedu Σ. Nechť Σ^* značí množinu všech slov nad abecedou Σ. Dále nechť $\Sigma^+ = \Sigma^* \setminus \{\epsilon\}$, nebo-li Σ^+ je množina všech neprázných slov nad abecedou Σ.

Definice 2.6. Nechť Σ je abeceda a $L \subseteq \Sigma^*$. Pak L je **jazyk** nad Σ.

Příklad 2.6. Například $L = \{ab, ac\}$ je jazyk na abecedou $\Sigma = \{a, b, c\}$.

Výše zmíněná definice utváří pojem jazyk nad Σ jako množinu slov nad Σ. Toto odpovídá jak přirozeným jazykům, tak i programovacím jazykům. Z definice plyne, že $\emptyset$ a $\{\epsilon\}$ jsou jazyky nad libovolnou abecedou, ovšem platí, že $\emptyset \neq \{\epsilon\}$, protože $\emptyset$ neobsahuje žádný element a $\{\epsilon\}$ obsahuje jeden element, ϵ. Množina všech jazyků nad abecedou Σ, nebo-li všechny podmnožiny množiny všech slov nad abecedou Σ, se značí 2^{Σ^*}. Na jazycích můžeme také definovat několik různých operací, jako je sjednocení, průnik, rozdíl, komplement, konkatenace, reverzace, iterace, uzávěr, pozitivní uzávěr, prefix a suffix. Množinové operace se dají přímo aplikovat také na jazyky s tím, že mají ekvivalentní předpis. Zde uvedeme výčet zmíněných operací.

Definice 2.7. Nechť L je jazyk nad abecedou Σ. **Komplement** L, $\overline{L}$ je definovaný jako $\overline{L} = \Sigma^* \setminus L$.

Příklad 2.7. Například $L = \{\epsilon, a, aa\}$ a $\Sigma = \{a\}$, pak $\overline{L} = \{a^i \mid a \in \Sigma \wedge i \geq 3\}$.

Kromě množinových operací a komplementu můžeme na jazyky rozšířit i operace nad slovy. Některé vybrané budou dále představeny.

Definice 2.8. Nechť L_1 a L_2 jsou dva jazyky. **Konkatenace** L_1 a L_2, L_1L_2, je definována následovně: $L_1L_2 = \{xy \mid x \in L_1 \wedge y \in L_2\}$.

Příklad 2.8. Například $L_1 = \{a, ab\}$ a $L_2 = \{b, ab\}$, pak $L_1L_2 = \{ab, aab, abb, abab\}$.

Ze zmíněné definice plynou dva důsledky, a to $L\{\epsilon\} = \{\epsilon\}L = L$, $L\emptyset = \emptyset L = \emptyset$.

Definice 2.9. Nechť L je jazyk. Pro $i \geq 0$, i-tá **mocnina** L, L_i, je definovaná jako:

- $L^0 = \epsilon$.
- $L^i = LL^{i-1}$, pro $i \geq 1$.

Příklad 2.9. Například druhá iterace jazyka $\{a, b\}$, $\{a, b\}^2$, odpovídá jazyku $\{aa, ab, ba, bb\}$.

Modely jazyků reprezentují výpočet popsaný danými jazyky. Jeden jazyk lze obecně popsat pomocí více modelů. O takových modelech pak hovoříme jako o *ekvivalentních modelech*. Abychom mohli tyto modely efektivně prozkoumávat, spojujeme je do tříd podle jejich vyjadřovací síly. Tyto třídy jsou poté zkoumány, z čehož nám vyplynou množiny jazyků patřících do těchto tříd. Třídy modelů mají *stejnou vyjadřovací sílu*, pokud charakterizují stejnou množinu jazyků.

2.1.3 Konečné automaty

Nyní když máme pojem jazyk, tak je potřeba mít nějaký model, který daný jazyk popisuje. Konečné jazyky mohou být popsány množinovým výčtem, ovšem nekonečné jazyky ne. V této práci budou diskutovány pouze regulární jazyky, pro které je konečný automat (vedle regulárních výrazů) jeden ze dvou fundamentálních modelů popisu. Regulární výrazy a konečné automaty jsou ekvivalentní, tzn. jsou vzájemně převoditelné. V praxi jsou modely pro regulární jazyky využívány v mnoha softwarových aplikacích, například lexikální analyzátor překladačů je implementován pomocí konečných automatů, nebo vyhledávát řetězce v textu lze pomocí sestavení příslušných regulárních výrazů.

Definice 2.10. Konečný automat je pětice $M = (Q, \Sigma, R, s, F)$, kde

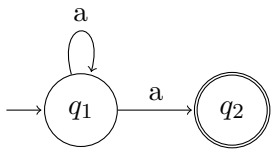
- Q je konečná množina stavů
- Σ je vstupní abeceda taková, že $Q \cap \Sigma = \emptyset$
- $R \subseteq Q(\Sigma \cup \{\epsilon\}) \times Q$ je relace, nazývající se *konečná množina pravidel*
- $s \in Q$ je počáteční stav
- $F \subseteq Q$ je množina koncových stavů.

Příklad 2.10. Například $M_1 = (\{q_1, q_2\}, \{a\}, \{(q_1a, q_1), (q_1a, a_2)\}, q_1, \{q_2\})$ přijímá jazyk $L = \{a^i \mid i \geq 1\}$. Vizualizace M je na obrázku 2.1.

Pravidlo $(pa, q) \in R$, kde $p, q \in Q$ a $a \in \Sigma \cup \{\epsilon\}$ se dle konvencí většinou píše jako $pa \vdash q$. Konečný automat může být vizualizován pomocí tabulkové nebo grafické reprezentace. Tabulková reprezentace je ve formě tabulky, kde řádky jsou prvky z $\Sigma \cup \epsilon$ a sloupce prvky z Q. Ukázka tvorby takové reprezentace je na obrázku 2.2a. Grafická reprezentace je více názornější z pohledu postupu samotného výpočtu, uzly jsou označeny prvky z Q a hrany jsou označeny prvky z $\Sigma \cup \epsilon$. Pravidla pro tento druh reprezentace jsou ukázána na obrázcích 2.2b a 2.2c. Konečný automat tedy obsahuje vstupní pásku, čtecí hlavu a konečné stavové řízení. Výpočet probíhá tak, že čte vstupní pásku a na základě ní mění svůj stav. Obecně ale pro změnu stavu nemusí číst žádný vstupní symbol a z jedné konfigurace může přejít do vícero

	a
q_1	$\{q_1, q_2\}$
q_2	$\emptyset$

(a) Tabulková reprezentace.



(b) Grafická reprezentace.

Obrázek 2.1: Tabulková a grafická reprezentace konečného automatu M_1.

konfigurací, tedy pracuje nedeterministicky. Tato vlastnost je z matematického pohledu hezká, nicméně v praxi se špatně implementuje, proto později zavedeme restrikce, které tuto vlastnost odstraní.

Definice 2.11. Nechť $M = (Q, \Sigma, R, s, F)$ je konečný automat. **Konfigurace** M je χ takové, že $\chi \in Q\Sigma^*$.

Příklad 2.11. Například konfigurace M_1 je q_1aa.

Význam této definice je, že konfigurace je aktuální stav automatu a dosud nepřečtená část vstupního řetězce. Konfigurace představuje uložitelný stav, nebo-li nějaký popis konkrétního automatu a vstupního řetězce takový, že můžeme kdykoliv přestat v přijímání vstupu, uložit konfiguraci a následně až budeme chtít pokračovat, tak z ní obnovíme proces přijímání vstupu právě tam, kde byl přerušen.

Definice 2.12. Nechť $M = (Q, \Sigma, R, s, F)$ je konečný automat a $r = (pa, q) \in R$ pravidlo. Pokud pay je konfigurace M, kde $p \in Q$, $a \in \Sigma \cup \epsilon$, $y \in \Sigma^*$, pak M může provést **přechod** z pay do qy za použití r, značeno jako $pay \vdash qy$ $[r]$ nebo zjednodušeně $pay \vdash qy$.

Příklad 2.12. Například M_1 z konfigurace q_1aa může provést přechod $q_1aa \vdash q_1a$ [1], nebo přechod $q_1aa \vdash q_2a$ [2].

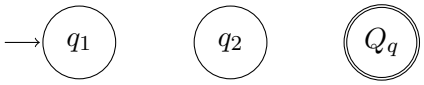
Jeden přechod představuje jeden výpočetní krok automatu. Toto nyní rozšíříme, abychom mohli provádět několik přechodů po sobě a tedy několik výpočetních kroků automatu.

Definice 2.13 (Sekvence přechodů 1/2). Nechť $M = (Q, \Sigma, R, s, F)$ je konečný automat.

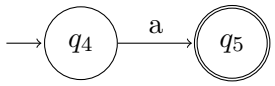
- Nechť χ je konfigurace M. M provede nula přechodů z χ do χ, zapisujeme $\chi \vdash^0 \chi$ $[\epsilon]$.

	...	a	...	ϵ
q_1				
...				
q_2		$t(q_2, a)$		
...				
$\underline{q_n}$				

(a) $q_1 \in Q$ je počáteční stav, koncové stavy jsou podtržené (zde $q_n \in F$), ostatní jsou běžné stavy (zde $q_2 \in Q$) a $t(p, a) = \{q \mid (pa, q) \in R\}$.



(b) $q_1 \in Q$ označuje počáteční stav, $q_2 \in Q$ označuje běžný stav a $q_3 \in F$ označuje koncový stav.



(c) $(q_4a, q_5) \in R$ označuje pravidlo $q_4a \vdash q_5$.

Obrázek 2.2: Pravidla pro vytvoření obecné tabulkové a grafické reprezentace konečného automatu $M = (Q, \Sigma, R, s, F)$.

- Nechť existuje sekvence konfigurací $\chi_0, ..., \chi_n$ pro nějaké $n \geq 1$ takové, že $\chi_{i-1} \vdash \chi_i$ $[r_i]$, kde $r_i \in R$, $i = 1, ..., n$, což znamená:

$$\begin{aligned} \chi_0 &\vdash \chi_1 \ [r_1] \\ &\vdash \chi_2 \ [r_2] \\ &\vdots \\ &\vdash \chi_n \ [r_n], \end{aligned}$$

  pak M provede n přechodů z χ_0 do χ_n podle $[r_1, ..., r_n]$, zapisujeme jako $\chi_0 \vdash^n \chi_n$ $[r_i ... r_n]$.

Definice 2.13 (Sekvence přechodů 2/2). Nechť $M = (Q, \Sigma, R, s, F)$ je konečný automat a χ, χ' dvě konfigurace M.

- Pokud existuje $n \geq 1$ takové, že $\chi \vdash^n \chi'$ $[\rho]$ v M, pak $\chi \vdash^+ \chi'$ $[\rho]$ (*tranzitivní uzávěr* $\vdash$).
- Pokud existuje $n \geq 0$ takové, že $\chi \vdash^n \chi'$ $[\rho]$ v M, pak $\chi \vdash^* \chi'$ $[\rho]$ (*tranzitivní a reflexivní uzávěr* $\vdash$).

Příklad 2.13. Například v M_1 lze provést sekvenci přechodů $q_1aa \vdash^2 q_2$ [12].

Podobně jako u přechodů lze zápis zjednodušit a psát $\chi \vdash^n \chi'$ místo $\chi \vdash^n \chi'$ $[\rho]$, $\chi \vdash^+ \chi'$ místo $\chi \vdash^+ \chi'$ $[\rho]$ a $\chi \vdash^* \chi'$ místo $\chi \vdash^* \chi'$ $[\rho]$. Všimněme si, že ρ, zvané *pravidlové slovo* odpovídající $\chi \vdash^n \chi'$ v M, specifikuje sekvenci n pravidel podle kterých M provede sekvenci přechodů $\chi \vdash^n \chi'$. Pravidla jsou dle konvence symbolizovány číslem pořadí jejich uvedení v definici automatu, nikoliv explicitním vypsáním. Tím pádem pravidlová slova můžeme interpretovat jako obyčejná slova nad abecedou přirozených čísel, nebo-li číselným označením pravidel a uplatňovat na ně dříve definované operace. Tímto principem lze sekvence pravidel efektivně zapisovat, například n použití pravidla 1 zjednodušit do předpisu: $\chi_0 \vdash^n \chi_n$ $[\rho]$, kde $\rho = \{1\}^n$.

Definice 2.14. Nechť $M = (Q, \Sigma, R, s, F)$ je konečný automat. **Jazyk přijímaný** M, $L(M)$, je definován: $L(M) = \{w \mid w \in \Sigma^* \wedge sw \vdash^* f \text{ v } M \text{ pro nějaké } f \in F\}$.

Příklad 2.14. Například $L(M_1) = \{a^i \mid a \in \Sigma \wedge i \geq 1\}$.

Neformálně řečeno, jazyk přijímaný konečným automatem jsou všechny řetězce takové, že jsou celé přečteny pomocí sekvencí přechodů a skončí v nějakém koncovém stavu.

2.2 WS1S a její souvislost s konečnými automaty

Již v roce 1960 bylo objeveno a dokázáno [3], že *Weak Second-order theory of 1 Successor* (nyní WS1S) odpovídá třídě *regulárních jazyků*, což znamená, že může být více přirozenou alternativou pro *regulární výrazy* a tedy lze převést na konečný automat. Představení WS1S a detailnější vysvětlení transformačního procesu, čerpané z [10], bude představeno v následujících podsekcích.

2.2.1 Syntax WS1S

WS1S umožňuje kvantifikování nad druhořádovými proměnnými, které mohou nabývat hodnot konečných podmnožin $\mathbb{N}_0$. Před samotným vysvětlením syntaxe WS1S je potřeba provést úpravy nutné k pozdějšímu výkladu:

- Prvořádové termy se zakódují jako druhořádové termy, protože na prvořádové termy můžeme nahlížet jako na jednoprvkové množiny. Booleovské proměnné lze zakódovat mnoha způsoby, není definována nějaká univerzální metoda. V podsekci 2.3.1 bude představena jedna konkrétní.
- Komplexní druhořádové termy se upraví do formy, kde obsahují navíc nové proměnné. Například formule $A = (B \cup C) \cap D$ se upraví do tvaru $\exists V : A = V \cap D \wedge V = B \cup C$.
- Podformule jsou přepsány, aby obsahovaly méně druhů operací. To znamená, že máme nějaké základní operace a všechny ostatní se na ně převadí. Například $\forall A : \phi$ se přepíše na $\neg(\exists A : \neg\phi)$.

Pro celou logiku WS1S nám stačí několik základních operací. Konkrétně libovolná formule ϕ se může skládat z těchto podformulí:

$$\begin{array}{ccc} \neg\phi' & \phi' \wedge \phi'' & \exists P_i : \phi' \\ P_i \subseteq P_j & P_i = P_j \setminus P_k & P_i = P_j + 1 \end{array}$$

kde P_i, P_j, P_k jsou množiny a ϕ', ϕ'' jsou formule. Praktický i konceptuální problém představují prvořádové proměnné, přesněji jejich negace. Jak již bylo představeno, tak tyto proměnné jsou převedeny na jednoprvkové druhořádové proměnné, ovšem samotný převod neřeší sémantiku operace negace. Pro ilustraci nechť formule $\phi : p = 0$, kde p je prvořádová proměnná je převedena uvedeným způsobem na formuli $\phi' : P = \{0\} \wedge singleton(P)$, kde P je druhořádová proměnná a $singleton(P)$ značí, že P je jednoprvková množina. Pokud bychom na ϕ' aplikovali negaci, chtěli bychom dostat negaci $\neg\phi$, která odpovídá $\neg(P = \{0\}) \wedge singleton(P)$, ale místo ní dostáváme $\neg(P = \{0\} \wedge singleton(P))$. Tento problém se řeší tak, že ke každému automatu A reprezentujícímu podformuli ϕ a každé volné proměnné P_i, se musí přidat konjunkce s automatem reprezentujícím vlastnost jednoprvkové množiny pro P_i.

2.2.2 Sémantika WS1S

Nechť ϕ_0 je hlavní formule, w je řetězec nad abecedou $\mathbb{B}^k$, kde $\mathbb{B} = \{0, 1\}$ a k je počet proměnných v ϕ_0. Každé proměnné P_i je přiřazeno číslo i z intervalu $1, 2, ..., k$. Řetězec w určuje interpretaci $w(P_i)$ proměnné P_i definovanou jako konečnou množinu $\{j \mid j\text{-tý bit v } P_i\text{-tém řádku je } 1\}$. Například formuli $\phi_0 \equiv \exists C : A = B \setminus C$, kde proměnným A, B, C jsou přiřazeny postupně indexy $1, 2, 3$, může interpretovat tento konkrétní řetězec w:

	0	1	2	3
A	1	0	1	0
B	0	0	1	0
C	1	1	0	0

Tento řetězec interpretuje všechny proměnné včetně C, ikdyž se jedná o vázanou proměnnou. Jazyk odpovídající formuli ϕ je nezávislý na řádku odpovídajícímu proměnné C

v tom smyslu, že změna w v C nijak neovlivní příslušnost w do jazyka automatu. Všimněme si, že přidání nebo odebrání suffixu $(0^k)^*$ k w je také nezávislé na zmíněné příslušnosti. Řekneme, že w je *minimální*, pokud neexistuje neprázdný suffix $(0^k)^*$, kde k je počet proměnných ve formuli.

Sémantika formule ϕ může být nyní definovaná induktivně vzhledem k interpretaci w, kde $w \vDash \phi$, pokud interpretace w znamená, že ϕ je *true*:

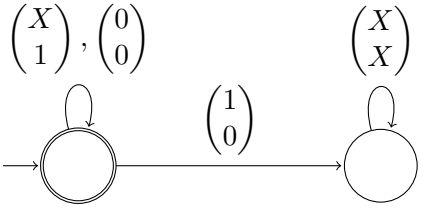
$w \vDash \neg\phi'$	právě tehdy, když	$w \nvDash \phi'$
$w \vDash \phi' \wedge \phi''$	právě tehdy, když	$w \vDash \phi' \wedge w \vDash \phi''$
$w \vDash \exists P_i : \phi'$	právě tehdy, když	$\exists$ konečná $M \subseteq \mathbb{N} : w[P_i \mapsto M] \vDash \phi'$
$w \vDash P_i \subseteq P_j$	právě tehdy, když	$w(P_i) \subseteq w(P_j)$
$w \vDash P_i = P_j \setminus P_k$	právě tehdy, když	$w(P_i) = w(P_j) \setminus w(P_k)$
$w \vDash P_i = P_j + 1$	právě tehdy, když	$w(P_i) = \{j + 1 \mid j \in w(P_j)\}$

kde $w[P_i \mapsto M]$ značí nejkratší řetezec w', který interpretuje všechny proměnné $P_j, j \neq i$ stejně jako w, ale P_i interpretuje jako M.

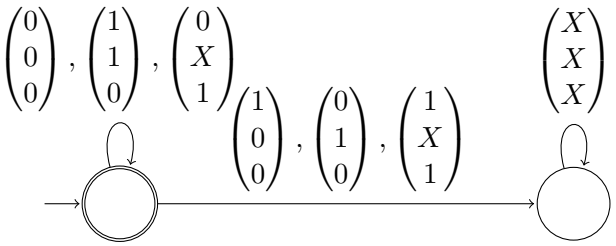
2.2.3 Převod formule na automat

Klíčová myšlenka je rekurzivně převádět každou podformuli hlavní formule ϕ_0 na deterministický konečný automat, který reprezentuje množinu splnitelných interpretací. Formule ϕ představuje jazyk $\mathcal{L}(\phi) = \{w \mid w \vDash \phi\}$, jenž odpovídá všem řetezcům interpetujícím splnitelná přiřazení proměnných ve ϕ. Jak již bylo zmíněno, každá formule sestává z libovolné kombinace šesti základních podformulí. V následující části ukáži jak každou takovou podformuli transformovat na deterministický konečný automat A takový, že $\mathcal{L}(A) = \mathcal{L}(\phi)$. Z tohoto trvzení také plyne, že $\mathcal{L}(\phi)$ je regulární jazyk. Transformace vypadá následovně:

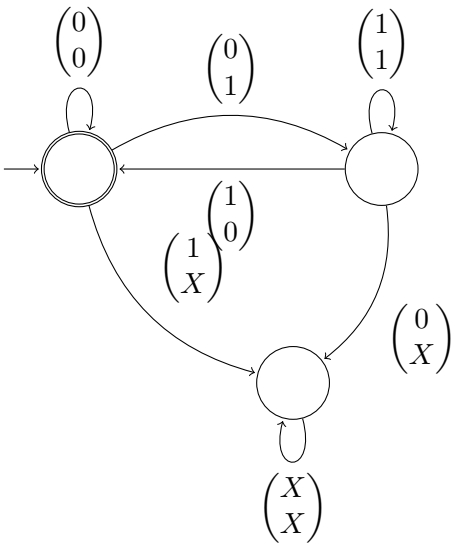
- $\phi = P_1 \subseteq P_2$:



- $\phi = P_1 = P_2 \setminus P_3$:



- $\phi = P_1 = P_2 + 1$:



- $\phi = \neg\phi'$: Negace formule odpovídá komplementu automatu, platí tedy, že $\mathcal{L}(\neg\phi') = \complement\mathcal{L}(\phi') = \complement\mathcal{L}(A') = \mathcal{L}(\complement A')$, kde A' je automat takový, že $\mathcal{L}(\phi') = \mathcal{L}(A')$ a $\complement$ je operace komplementu. Pokud je automat úplný a deterministický, lze tuto operaci provést pouhým prohozením koncových a nekoncových stavů.

- $\phi = \phi' \wedge \phi'$: Konjunkci formulí realizujeme pomocí průniku jazyků, formálně $\mathcal{L}(\phi' \wedge \phi'') = \mathcal{L}(\phi') \cap \mathcal{L}(\phi'')$. Cílový automat A je produkt automatů $A' \times A''$, kde $\mathcal{L}(\phi') = \mathcal{L}(A'), \mathcal{L}(\phi'') = \mathcal{L}(A'')$. Stavový prostor výsledného automatu se dá zoptimalizovat tak, že z nově vytvořených stavů budeme uvažovat pouze dosažitelné.

- $\phi = \exists P_i : \phi'$: Intuitivně je výsledný automat A shodný s automatem A' pro ϕ', ale smaže hodnotu na P_i-tém řádku. Výsledkem této operace může být nedeterministický automat A, tudíž je potřeba ho determinizovat a upravit tak, aby každé $w \in \mathcal{L}(A)$ bylo minimální, což znamená, aby neobsahovalo suffix tvaru 0^+, který je zbytečný.

2.3 Nástroj Mona

Jedná se o nástroj implementující rozhodovací procedůru pro logiky WS1S a WS2S. WS2S, *Weak monadic Second-order theory of 2 Succesors*, je generalizace logiky WS1S, kde elementy jsou generovány dvěma následníky, namísto jednoho. Interpretace WS1S odpovídá řetězcům, zatím co WS2S stromům, tím pádem Mona pro její rozhodování používá speciální typ automatů, zvaný *Guided Tree Automata*. V případě WS1S pak Mona vrací jako výsledek minimální deterministický konečný automat, jehož jazyk je množina řetězců, které interpretují globální proměnné tak, že konjuknce všech formulí ve vstupním programu platí. Mona dále analyzuje automat a vrátí nejkratší protipříklad řetězce pomocí hledání nejkratší sekvence přechodů z počátečního do zamítajícího stavu a nejkratší splnitelný příklad řetězce analogicky s tím rozdílem, že hledá koncový stav. Jde o velmi efektivní nástroj na rozhodování těchto logik, a to i přes jejich *non-elementary* složitost. Během vývoje Mony bylo objeveno mnoho technik, ze kterých nyní těží efektivita nástroje. Mezi hlavní techniky patří *BDD reprezentace automatů*, *dagifikace formulí* a *restrikce formulí*. Podrobně tyto

techniky popíšeme v následujících podsekcích. Všechny informace z této sekce jsou čerpány z [10, 11, 5, 8, 9].

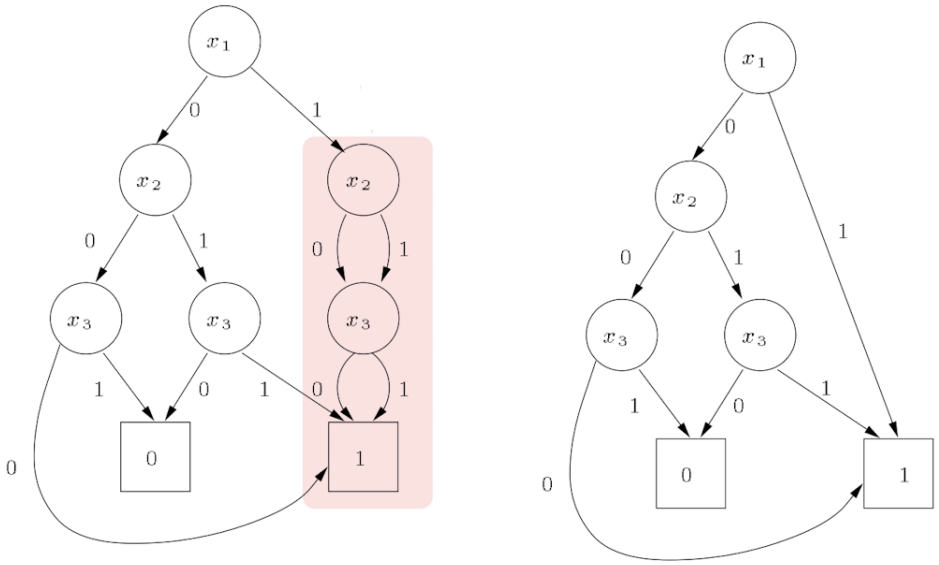
2.3.1 Reprezentace automatů

Klasická reprezentace automatu odpovídá tomu, že máme přechod pro každý symbol. V našem případě by to znamenalo, že pokud je v dané formuli n proměnných, tak je potřeba pro jeden stav, v nejhorším případě, 2^n přechodů, abychom pokryli všechna ohodnocení proměnných ve formuli. Tento počet přechodů už je poměrně vysoký a má za následek zpomalení výpočtu přechodové relace, což vyústilo v integraci BDD reprezentace přechodů automatů. BDD, *Binary Decision Diagrams*, jsou speciální druh deterministického automatu, který přijímá konečná slova. Další jejich výhoda je, že jdou minimalizovat v lineárním čase. Umožňují tedy výraznou kompresi symbolické informace a zrychlují výpočet. Tento přístup se často označuje jako *symbolický*.

BDD. BDD jsou často používány pro výpočet pravdivostní hodnoty funkce v booleovské logice. Uvažme například booleovskou funkci $\Phi(x_1, x_2, x_3) \equiv x_1 \vee (x_2 \Leftrightarrow x_2)$, kde $x1$, $x2$, $x3$ jsou proměnné. BDD reprezentace Φ je orientovaný acyklický graf, jehož vnitřní uzly jsou označeny názvy proměnných a listové uzly reprezentují hodnotu funkce Φ. Reprezentace je závislá na pořadí proměnných. Různá uspořádání proměnných znamenají různě velké a tím padém i odlišně rychlé BDD. Nicméně určit seřazení takové, aby bylo použito minimální množství uzlů je NP-úplný problém. Mona proměnné řadí tak, že všechny explixitně deklarované proměnné jsou řazeny vzestupně dle výskytu deklarace a implicitně deklarované proměnné, což jsou ty, které se vytvoří automaticky pro potřeby tvorby automatů (například při zpracovávání formule $a < b + 1$ je potřeba vytvořit novou proměnou a formuli přepsat do tvaru $\exists x : a < x \wedge x = b + 1$), se vkládají za explicitně deklarované. Na obrázku 2.3 můžeme vidět příslušné BDD pro funkci Φ s upořádáním proměnných x_1, x_2, x_3. Pro zjistění hodnoty funkce se zadanými parametry postupujeme následovně. Začneme v kořenovém uzlu, podíváme se jakou proměnnou představuje, zjistíme hodnotu této proměnné a podle tohoto ohodnocení se přesuneme do dalšího uzlu. Toto opakujeme dokud se nedostaneme do listového uzlu (označen čtverečkem), který obsahuje hodnotu výsledku funkce.

Můžeme si všimnout, že zde existuje jistá podobnost s konečnými automaty. BDD mají ovšem navíc možnost *komprimovat cesty*, což je užitečné v případě, že bychom měli posloupnost stavů pro které platí, že z nich vedou přechody přes všechny symboly do stejného stavu. V kontextu naší úlohy to znamená, že několik proměnných po sobě může mít libovolnou hodnotu, tedy je zbytečné tyto jejich stavy uvažovat, protože nám nepřináší žadnou novou informaci.

Operace na BDD. Díky tomu, že BDD jsou grafy, tak je můžeme efektivně ukládádat v počítači. Složitost booleovských operací na BDD, jako jsou $\phi \vee \psi$ a $\phi \wedge \psi$, kde BDD reprezentace formulí ϕ, ψ má velikost n, respektive m, je $O(n \cdot m)$. Projekce, nebo-li formule $\exists x : \phi$ má lineární složitost, negaci, $\neg\phi$, lze provézt dokonce v konstantním čase. Vytvořit BDD můžeme pomocí minimálních deterministických konečných automatů a aplikace komprese cest. Jazyk tohoto automatu jsou všechna splnitelná přiřazení proměnných, zakódované jako řetězce o délce počtu proměnných, kde pozice v řetězci odpovídá uspořádání proměnných.



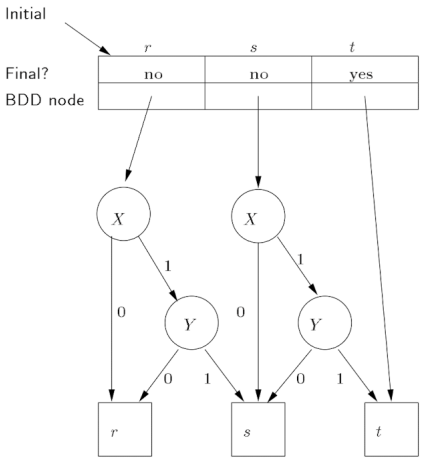
(a) BDD před kompresí cest.

(b) BDD po kompresi cest.

Obrázek 2.3: Ukázka možného BDD pro booleovskou funkci $\Phi(x_1, x_2, x_3) \equiv x_1 \vee (x_2 \Leftrightarrow x_2)$.

BDD reprezentace automatu. Jak už jsme zmínili, použití klasické reprezentace automatu vede k exponenciální velikosti abecedy, vzhledem k počtu proměnných ve formuli, tedy každý stav může mít až exponenciálně přechodů, což je značně neefektivní. Nejprve uvažme BDD reprezentaci automatu, při které jedno BDD kóduje množinu $\{(r, a, s) \mid r \to^a s\}$, kde r, s jsou stavy a $a \in \mathbb{B}^k$ je symbol, což odpovídá celému automatu. Taková reprezentace vyžaduje zvolit vhodné kódování stavů. Výše uvedený přístup je také používán v symbolickém model checkingu. Například Kripkeho struktura $M = (S, I, R, L)$ nad atomickými pozorováními $AP = \{p, q, r\}$, kde $S = \{s_1, s_2, s_3\}$ je konečná množina stavů, $I \subseteq S = \{s_1\}$ je množina počátečních stavů, $R \subseteq S \times S = \{(s_1, s_2), (s_2, s_3), (s_1, s_3), (s_3, s_3)\}$ je přechodová relace a $L : 2^{AP}$ je označovací funkce. Pokud máme n stavů, pak počet proměnných v BDD na zakódování těchto stavů je $m = \lceil \log_2 n \rceil$. V našem příkladě tedy potřebujeme dvě proměnnné v_1, v_2. Formule $\phi_S(v_1, v_2) = \neg v_1 \vee (v_1 \wedge \neg v_2)$ reprezentuje množinu stavů S. Toto zakódování říká, že stav s_1, s_2, s_3 odpovídá postupně $\phi_S(00)$, $\phi_S(01)$, $\phi_S(10)$. Dále každé atomické pozorování má vlastní BDD, které představuje funkci, jenž nám dává informaci v jakých stavech dané pozorování platí. Bohužel výše popsaná reprezentace není vhodná pro některé základní automatové operace, které v rozhodovací procedůře potřebujeme, jako je minimalizace nebo determinizace. Samotná reprezentace není kanonická, protože neexistují žádná pravidla pro zakódování stavů. Vidíme, že reprezentace pomocí jednoho BDD není ideální pro automatové operace, tím pádem budeme chtít jinou datovou strukturu, která bude podporovat efektivní automatové operace.

SMTBDD. Výše uvedený problém budeme řešit pomocí *shared, multi-terminal BDD*. Rozdíl oproti klasickým je, že listy nejsou booleovské 0,1, ale odpovídají stavům automatu. Každý stav je asociován k jednomu BDD, ovšem jednotlivá BDD spolu navzájem sdílejí uzly. Například formule $\exists p, q : p \neq q \wedge p \in X \cap Y \wedge q \in X \cap Y$ (neformálně, průnik X a Y obsahuje více než jeden element), je reprezentována automatem M na obrázku 2.4,



Obrázek 2.4: Mona automat pro formuli $\exists p, q : p \neq q \wedge p \in X \cap Y \wedge q \in X \cap Y$.

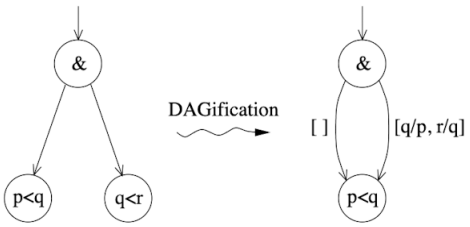
kde je každý stav, r, s, t, definován pomocí informace jestli je koncový a ukazatele na svůj počáteční BDD uzel, určující jeho přechodovou relaci. Datová struktura ještě obsahuje jeden ukazatel na počáteční stav celého automatu. Jazyk, který automat přijíma je $L(M) = (\binom{0}{0} + \binom{0}{1} + \binom{1}{0})^*\binom{1}{1}(\binom{0}{0} + \binom{0}{1} + \binom{1}{0})^*\binom{1}{1}(\binom{0}{0} + \binom{0}{1} + \binom{1}{0} + \binom{1}{1})^*$.

Interpretace řetězců. Tato část pojednává o tom, jak mapujeme řetězce automatu na přiřazení hodnot proměnných v logice. Jako příklad uvažme WS1S formuli $\phi \equiv P = Q$ a interpretaci $P = \{1, 3\}$, $Q = \{2, 3\}$. Tato interpretace je reprezentována řetězcem nad abecedou $\mathbb{B}^2$:

	-1	0	1	2	3	4	5
P	0	0	1	0	1	0	0
Q	0	0	0	1	1	0	0

Zde první řádek odpovídá přirozeným číslům, kromě první pozice, která je vyhrazena pro booleovské proměnné, jenž budou vysvětleny v zápětí. Druhý řádek odpovídá prezenci hodnot z prvního řádku v proměnné P. Třetí řádek je analogický k druhému, akorát představuje proměnnou Q. Pokud označíme tento řetězec jako w, můžeme psát, že $w \nvDash \phi$. Všimněme si, že existuje mnoho dalších řetezců, které popisují tuto interpretaci, protože když přidáme neomezený počet symbolu $\binom{0}{0}$, interpretace zůstane totožná. Obecně pro k proměnných formule ϕ definuje jazyk $L(\phi) = \{w \mid w \vDash \phi\}$ nad abecedou $\mathbb{B}^k$.

Přirozená čísla se v klasickém přístupu kódují shodně jako množiny, ale navíc s omezením na počet prvků množiny roven 1 (např. $a = 1 \wedge singleton(a)$). V Moně se tento přístup liší, a to tak, že tyto proměnné jsou neprázdné množiny, které jsou interpretovány jako jejich nejmenší hodnota. Booleovské proměné jsou zakodovány v řetězci tak, že je na jeho začátek přidána speciální pozice, značená -1. Jestliže je proměnná typu bool, pak je na tuto pozici vložena $0, 1$, pokud je proměnná ohodnocena jako postupně $true, false$. Vidíme, že když nepoužíváme negaci, je jazyk automatů bez prázdného řetězce, ikdyž by ve formuli nebyly globální proměnné. Další rozdíl mezi klasickým přístupem a Mona přístupem je použití výrazně většího počtu speciálních automatů pro příslušné logické operace, aby se nemuselo často zjednodušovat formule (např. převádět každou implikaci na negaci a disjunkci).



Obrázek 2.5: Transformace formule $\exists q : p < q \wedge q < r$ z kódového stromu na OAG.

2.3.2 Dagifikace formulí

Mona je rozdělena na front-end a back-end. Front-end se stará o načítání, syntaktickou a sémantickou analýzu vstupní formule a vytvoření AST (*abstraktního syntaktického stromu*), jehož vnitřní uzly jsou podporované automatové operace a listy jsou atomické formule. Back-end poté induktivně provádí tyto automatové operace a nakonec vytvoří automat korespondující k dané formuli. V dřívějších verzích nástroje se přímo z AST vytvořil kódový strom a z něj se postupně generovaly automaty. Lze ovšem vypozorovat, že v kódovém stromu se nachází mnoho stejných podstromů modulo přejmenování proměnných. Výše zmíněný fakt vedl k tomu, že byla vytvořena tato optimalizace a shodné podstromy modulo přejmenování se vytváří pouze jednou. To má za následek, že kód se již nevytvoří ze stromu, ale z OAG (*orientovaného acyklického grafu*) a již vytvořené automaty se použijí vícekrát. Kódové stromy mohou být například ve formě $mk\_basic\_less(i, j)$, $mk\_product(C, C', op)$, $mk\_project(i, C)$, kde i, j jsou indexy BDD proměnných, op je booleovská funkce s dvěma proměnnými a C, C' jsou kódové stromy. Pro ilustraci uvažme formuli $\exists q : p < q \wedge q < r$. Předpokládejme, že proměnné p, q, r mají postupně index 1,2,3, kde index udává pořadí řazení proměnných, pak je tato formule transformována na kódový strom $mk\_project(2, mk\_product(mk\_less(1, 2),\ mk\_less(2, 3), \wedge))$. Na tomto stromě můžeme vidět, že obsahuje dva izomorfní podstromy, ze kterých by se později dvakrát počítal stejný automat. Automat A pro $mk\_less(1, 2)$ je identický s automatem A' pro $mk\_less(2, 3)$ modulo přejmenování proměnných. Ukázka transformace z kódového stromu na OAG je na obrázku 2.5. Lze nahlédnout, že listové uzly jsou izomorfní modulo přejmenování, tudíž se sjednotí v jeden uzel a obě hrany jsou označeny použitým přejmenováním. Obecně bychom chtěli přejmenovat indexy v A kdykoliv kdy potřebujeme A'. Touto optimalizací můžeme podstatně zlepšit celkovou rychlost programu, protože přejmenování indexů je lineární operace, zatím co konstrukce A' je většinou časově složitější. Řekneme, že kódový strom C je ekvivalentní C', pokud existuje přejmenování proměnných, které zachovává pořadí, v C' takové, že z C' se stane C.

Z uvedené ekvivalence plyne, že uvažovaná formule splňuje tuto ekvivalenci a tím pádem automat může být znovupoužit. Výhoda BDD reprezentace je, že jestliže dva kódové stromy splňují výše uvedenou ekvivalenci, pak mají identické BDD až na listové uzly, kde každý má obecně jiné stavy do kterých může přejít. Pokud bychom ve výše zmíněné formuli zachovali indexy a změnili ji do podoby $\exists q : p < q \wedge r < q$, pak definice ekvivalence není splněna (tedy dvě BDD jsou různé pro obě atomické formule) a dagifikace se neprovede. Cílem je vytvořit OAG, který vznikne z kódového stromu pomocí odstranění ekvivalentních podstromů. Celkový čas potřebný pro tuto procedůru je ovšem kvadratický, protože jak již bylo zmíněno, je potřeba lineární čas na samotné přejmenování a další lineární čas

zabere výpočet relace ekvivalence. Z toho důvodu je dagifikace omezena na podstromy o maximálním počtu proměnných, který uživatel specifikuje pomocí parametru.

2.3.3 Restrikce formulí

Motivací pro vytvoření této optimalizace je problém zakódování prvořádových proměnných, tedy jak efektivněji vyjádřit prvořádové proměnné pomocí druhořádových. Hlavní myšlenka restrikcí je, že k hlavní formuli ϕ přidružíme restrikci ϕ_R, kde ϕ_R je také formule. Pokud je ϕ formule s restrikcí, pak ϕ_R je různá od *true*, jinak pokud neobsahuje restrikci, pak ϕ_R je *true*. ϕ můžeme dále uvažovat pouze pokud ϕ_R platí. Například nechť $\phi = \exists x : x \cup y = \emptyset$ a $\phi_R = x \subseteq y$, pak vytvořený automat je ekvivalentní s automatem pro formuli $\phi' = \exists x : x \cup y \wedge x \subseteq y$. Nyní budou popsány různé způsoby řešení tohoto problému, tak jak postupem času byly implementovány v nástroji Mona, včetně zde naznačených restrikcí.

Ad hoc strategie a konjunktivní sémantika. Nejjednodušší přístup pro vyjádření prvořádových proměnných je *ad hoc strategie*, což je klasický přístup, kdy s proměnnou p zacházíme jako s druhořádovou proměnnou, tedy množinou o jednom prvku a v místě vzniku p (hlavní formule, pokud je volná proměnná nebo místo kde vzniká kvantifikováním, pokud je vázaná) k ní přidáme konjunkcí singleton predikát. Tato strategie znamená, že sémantika formule obsahující p není robustní, protože její význam na interpretacích řetězců jazyka w nesplňujících $singleton(p)$ není dobře definován. Ikdyby byl tento predikát přidán ke každému výskytu p v atomických formulích tak, jak již bylo rozebíráno v předchozí kapitole, sémantika této reprezentace není uzavřená vůči komplementu. Řešení je přidat konjunkci s restrikcí ke každé podformuli formule p, čemuž říkáme *konjunktivní sématika*. V tomto případě má každý řetězec w korektní sémantiku, jelikož po případné negaci ve vnořené formuli obsahující prvořádové proměnné p_i následuje konjunkce se singleton automatem pro všechny proměnné p_i, tedy tyto proměnné zůstanou nadále interpretované jako prvořádové, a to po celou dobu výpočtu. Praktický problém s tímto přístupem je to, že se provede navíc velké množství operací minimalizace a průniku. Konkrétně, pro každý automat A reprezentující podformuli ϕ a každou volnou proměnnou P_i, automat reprezentující $singleton(P_i)$, musí být konjunkcí přidán k A. Tyto operace, provedené navíc oproti *ad hoc* přístupu, zpomalí rozhodovací procedůru alespoň dvojnásobně. Komplementace, která je rychlá operace, protože automaty jsou deterministické a úplné, sestává z prohození koncových a nekoncových stavů, ovšem nyní by obsahovala navíc minimalizaci a průnik. Stějně tak operace průniku by provedla navíc alespoň jednou minimalizaci a průnik. Z těchto důvodu se v Moně, před implementací restrikcí, používal *ad hoc* přístup s tím, že se singleton vlastnost proměnných přidala k atomickým formulím, kde se daná proměnná vyskytovala a k formuli, kde příslušná proměnná vznikla. Add hoc sématiku můžeme nadefinovat pomocí významové funkce $[\![\phi]\!]^{ah}$ takto:

$$[\![\neg\phi']\!]^{ah}w = \neg[\![\phi']\!]^{ah}w \tag{2.1}$$

$$[\![\phi' \wedge \phi'']\!]^{ah}w = [\![\phi']\!]^{ah}w \wedge [\![\phi'']\!]^{ah}w \tag{2.2}$$

$$[\![\exists P^i \ where \ \rho : \phi']\!]^{ah}w = \begin{cases} 1 & \text{pokud } \exists M : [\![\phi']\!]^{ah}w[P^i \mapsto M] = 1 \wedge [\![\rho^*(P^i)]\!]^{ah} = 1 \\ 0 & \text{pokud } \forall M : [\![\phi']\!]^{ah}w[P^i \mapsto M] = 0 \vee [\![\rho^*(P^i)]\!]^{ah} = 0 \end{cases} \tag{2.3}$$

$$[\![P^i \subseteq P^j]\!]^{ah}w = \begin{cases} 1 & \text{pokud } w \vDash P^i \subseteq P^j \wedge [\![\rho^*(P^i \subseteq P^j)]\!]^{ah}w = 1 \\ 0 & \text{pokud } w \nvDash P^i \subseteq P^j \vee [\![\rho^*(P^i \subseteq P^j)]\!]^{ah}w = 0 \end{cases} \tag{2.4}$$

kde ρ^* jsou všechny restrikce relevantních proměnných (bude detailněji vysvětleno v následujícím odstavci). Například intuitivní význam rovnice 2.4 je takový, že *true* vrátí, pokud w je modelem a všechny relevantní restrikce jsou splněny, naopak $false$ vrátí, pokud w není modelem nebo nějaká relevantní restrikce není splněna. Sémantika na ostatních základních atomických formulích se definuje analogicky. Konjunktivní sémantika se definuje stejně, akorát jsou restrikce aplikovány také na operátory $\wedge$ a $\neg$.

Restrikce a tří-hodnotová sémantika. Mona si mírně upravila syntax WS1S, a to tak, že umožňuje vytvářet restrikce explicitně. Existenční kvantifikace poté vypadá jako $\exists P^i \ where\ \rho : \phi'$, kde ρ je restrikce proměnné P^i, značeno jako $\rho(P^i)$. Předpokládejme, že každá P^i má nějakou restrikci (pokud by neměla, tak $\rho(P^i) = true$). Dále řekněme, že v $\rho(P^i)$ je proměnná P^i *ρ-vázaná*. Proměnná P^i je *existenčně vázaná* v $\rho(P^i)$ i v ϕ'. Výskyt proměnné P^i je *volný* v konvenční části ϕ, pokud P^i je volná v ϕ v běžném významu, kde ϕ je považována za nezávislou formuli a výskyt není v rámci restrikce existenční kvantifikace v rámci ϕ. *Relevantní proměnné* formule ϕ, $RV(\phi)$, je nejmenší množina proměnných P taková, že existuje výskyt P, který není ρ-vázaný a zároveň je volný v konvenční části ϕ nebo volný v konvenční části $\rho(P')$, kde $P' \in RV(\phi)$. Dále definujme *indukovanou restrikci* $\rho^*(\phi)$ jako $\rho^*(\phi) = \wedge_{P^i \in RV(\phi)}$, což, neformálně řečeno, je konjunkce restrikcí relevantních proměnných. Nechť $\mathbb{B}^- = \mathbb{B} \cup \{\bot\}$ je *rozšířená booleovská doména*. Symbol $\bot$ bude označovat situaci, kdy existuje restrikce, která neplatí. Booleovské operátory ($\wedge^3$ a $\neg^3$ jsou definovány na $\mathbb{B}^- \times \mathbb{B}^-$ obvyklým způsobem, navíc je situace s $\bot$, kdy je vždy výsledek $\bot$) a tří-hodnotová sémantika jsou definovány následovně:

$\neg^3$	
$\bot$	$\bot$
$-$	$+$
$+$	$-$

$\wedge^3$	$\bot$	$-$	$+$
$\bot$	$\bot$	$\bot$	$\bot$
$-$	$\bot$	$-$	$-$
$+$	$\bot$	$-$	$+$

$$[\![\neg\phi']\!]^3 w = \neg^3 [\![\phi']\!]^3 w \tag{2.5}$$

$$[\![\phi' \wedge \phi'']\!]^3 w = [\![\phi']\!]^3 w \wedge^3 [\![\phi'']\!]^3 w \tag{2.6}$$

$$[\![\exists P^i \ where\ \rho : \phi']\!]^3 w = \begin{cases} 1 & \text{pokud } \exists M : [\![\phi']\!]^3 w[P^i \mapsto M] = 1 \\ 0 & \text{pokud } \forall M : [\![\phi']\!]^3 w[P^i \mapsto M] \neq 1 \wedge \exists M : [\![\phi']\!]^3 w[P^i \mapsto M] = 0 \\ \bot & \text{pokud } \forall M : [\![\phi']\!]^3 w[P^i \mapsto M] = \bot \end{cases} \tag{2.7}$$

$$[\![P^i \subseteq P^j]\!]^3 w = \begin{cases} 1 & \text{pokud } w \vDash P^i \subseteq P^j \wedge [\![\rho^*(P^i \subseteq P^j)]\!]^3 w = 1 \\ 0 & \text{pokud } w \nvDash P^i \subseteq P^j \wedge [\![\rho^*(P^i \subseteq P^j)]\!]^3 = 1 \\ \bot & \text{pokud } [\![\rho^*(P^i \subseteq P^j)]\!]^3 \neq 1 \end{cases} \tag{2.8}$$

Zde například rovnice 2.8 neformálně říká, že pokud je w model a všechny relevantní restrikce jsou splněny, pak vrací *true*, jinak pokud w není model a všechny relevantní restrikce jsou splněny, vrací $false$, jinak pokud nějaká relevantní restrikce není splněna, vrací $\bot$. Následně můžeme sestrojit tři ekvivalence:

$$w \nvDash \rho(P^i) \text{ pro nějaké } P^i \in RV(\phi) \Leftrightarrow w \nvDash \rho^*(\phi) \Leftrightarrow [\![\phi]\!]^3 w = \bot \tag{2.9}$$

$$w \vDash \phi \wedge \rho^*(\phi) \Leftrightarrow [\![\phi]\!]^3 w = 1 \tag{2.10}$$

$$w \vDash \neg\phi \wedge \rho^*(\phi) \Leftrightarrow [\![\phi]\!]^3 w = 0 \tag{2.11}$$

Ekvivalence 2.9 hovoří o situaci, kdy je porušena nějaká restrikce, ekvivalence 2.10 o případu, kdy je splněna formule i restrikce a konečně ekvivalence 2.11 o situaci, kdy není splněna formule, ale je splněna restrikce. Pokud upravíme původní algoritmus tak, aby reflektoval tří-hodnotovou sémantiku, získáme výsledný algoritmus tří-hodnotové rozhodovací procedůry pro WS1S.

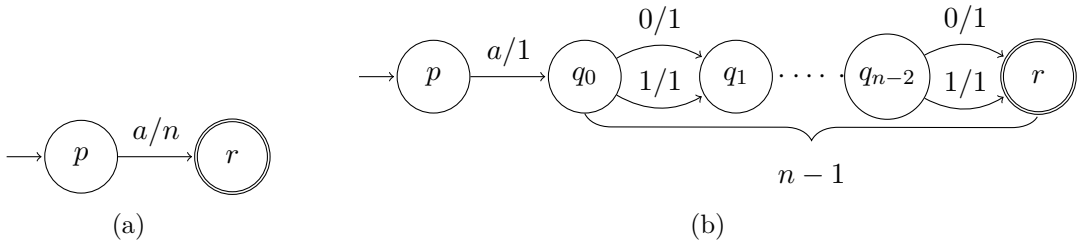
Kapitola 3

Nová rozhodovací procedůra

Tato kapitola představí novou reprezentaci automatů a přizpůsobené automatové algoritmy pro efektivní práci s touto reprezentací, což představuje hlavní konceptuální přínos této práce.

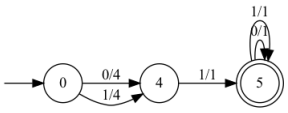
3.1 Hlavní myšlenka

Hlavní změna v nové rozhodovací procedůře je, že BDD uzly použité k reprezentaci přechodů integrujeme přímo do automatu jako stavy. Toto lze snadno, protože protože BDD procedůry jsou podobné automatovým. Abychom docílili funcionality BDD, kde jsou k uzlům přidruženy proměnné, musíme v automatu použít skip hrany místo běžných hran. Běžný konečný automat tedy upravíme tak, že změníme definici konečné množiny pravidel, což je relace $R \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$ do podoby $R \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times \mathbb{N} \times Q$, kde nová relace přidává ke každému pravidlu jedno kladné číslo, které značí délku hrany, což odpovídá počtu přeskočených hran $+1$. Poznamenejme, že délka hrany 1 je speciální případ odpovídající běžným hranám. Automat se skip hranou ze stavu p do stavu r s přechodovým symbolem a a délkou hrany n, kde $n > 0$, přijme a, poté $n-1$-krát všechny symboly abecedy a následně skončí ve stavu r (na obrázku 3.1 vidíme ukázku obecné skip hrany a korespondující běžné hrany).

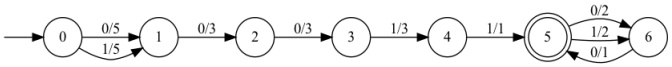


Obrázek 3.1: Porovnání obecné skip hrany s přechodovým symbolem a o délce n (obrázek 3.1a) a jí ekvivalentním běžným hranám (obrázek 3.1b). Notaci na hranách a/n interpretujeme jako přechodový symbol a a délku hrany n.

Se skip hranami lze zajít dál než s BDD, protože umožňují neomezeně dlouhý skok, tedy oproti BDD kódování přechodů je výhoda, že skip hrany mohou přeskočit i několik stavů automatu. BDD dokáží přiřadit booleovské ohodnocení každé proměnné ve formuli a pokud na její hodnotě nezáleží, tak nevytvoří vnitřní uzel, ale pokud bychom měli formuli, kde



(a) Množina proměnných $\{X\}$ a formule $3 \in X$.



(b) Množina proměnných $\{X, Y, Z\}$ a formule $Z = \{2, 3\}$.

Obrázek 3.2: Nastínění možných výhod přechodů pomocí skip hran oproti pomocí BDD. BDD přechody by museli přidat stav vždy po jednom přiřazení do všech proměnných, což v terminologii skip hran znamená maximálnní délku hrany rovnu počtu proměnných a před přiřazením do první proměnné musí existovat stav.

například mohou mít n-krát všechny proměnné libovolnou hodnotu, pak sice BDD nebude mít žádný vnitřní uzel, ale musí se vytvořit n automatových stavů, protože pomocí BDD nelze říct, že proměnné přeskočím vícekrát. Naopak u skip hran může být délka hrany větší něž počet proměnných a tedy není nutné vytvářet zbytečné stavy. Ukázka takové formule je na obrázku 3.2. Díky této vlastnosti se také ušetří stavy, pokud přiřazujeme do jedné proměnné a na ostatních nám nezáleží, protože vždy po každém jednom přiřazení do všech proměnných se nevytvoří stav, pokud to není nutné (viz příklad z obrázku 3.2). Zmíníme také, že čistě automatová reprezentace, namísto kombinace automatů a BDD, je flexibilnější, lze do ní lépe vidět a umožňuje navíc třeba nedeterminismus nebo reverzaci.

3.2 Automatové algoritmy

V následující části budou popsány vybrané automatové algoritmy potřebné pro rozhodování WS1S upravené tak, aby efektivně pracovaly s představenou reprezentací. Zde je výčet často používaných symbolů a vysvětlení jejich intuitivního významu:

- s_a / S_a znamená aktuální stav / množinu stavů
- s_f / S_f znamená následující stav / množinu stavů
- s_i / S_i znamená mezistav / množinu mezistavů
- $symb$ znamená aktuální přechodový symbol

3.2.1 Průnik

Průnik je běžná automatová operace, proto jsme se rozhodli, že tento algoritmus popíšeme zjednodušeně a dáme důraz na odlišnosti při práci se skip hranami, což vizualizuje algoritmus 1.

Společný rámec průniku. Nejprve vytvoříme všechny dvojice iniciálních stavů obou vstupních automatů a umístíme je do vektoru. Následně dokud je vektor neprázdný, vytahujeme z něj dvojice stavů. Podmínka na řádku 11 říká, že ze stavu s_{a_1} existuje přechod v aut_1 do nějakého stavu s_{f_1} přes symbol $symb$ a ze stavu s_{a_2} je možné v aut_2 přejít přes symbol $symb$ do stavu s_{f_2}. Funkce $getNextMove$ je obdoba výše uvedené podmínky, ovšem nevrací binární hodnotu, ale množinu všech s_{f_1} (značeno jako S_{f_1}) a všechny s_{f_2} (označíme jako S_{f_2}). Procedůra $createResultStateAndTrans$ vytvoří přechod přes aktuální symbol

Algorithm 1 Průnik automatů se skip hranami

```
1:  procedure INTERSECTION(aut_1, aut_2)
2:      result ← emptyAutomaton()
3:      q ← emptyVector()
4:      for init_1 in aut_1.initial_states do
5:          for init_2 in aut_2.initial_states do
6:              q.push((init_1, init_2))
7:          end for
8:      end for
9:      while q is not empty do
10:         s_{a_1}, s_{a_2} ← q.pop()
11:         while exists next move from s_{a_1}, s_{a_2} do
12:             S_{f_1}, S_{f_2}, symb ← getNextMove()
13:             for s_{f_1} in S_{f_1} do
14:                 for s_{f_2} in S_{f_2} do
15:                     addInnersInter(aut_1, aut_2, s_{a_1}, s_{a_2}, s_{f_1}, s_{f_2}, symb)
16:                     createResultStateAndTrans()
17:                 end for
18:             end for
19:             addResultTrans()
20:         end while
21:     end while
22:     return result
23: end procedure
```

a nový stav výsledného automatu, pokud je aktuální dvojice stavů viděna poprvé a také pokud jsou oba stavy koncové, nastaví nový stav jako koncový a *addResultTrans* přidá nově vzniklé přechody a stavy do výsledného automatu. Ve funkci *addInnersInter* se odehrává to nejpodstatnější, což je vypořádání se se skip hranami. Algoritmy 2 a 3 představují dvě nejefektivnější realizace, které jsme implementovali.

Princip vytváření mezistavů pro průnik. Princip vytváření mezistavů je následující. Nechť máme aktuální dvojici stavů s_{a_1} a s_{a_2}, která přes symbol *symb* přejde do stavů s_{f_1} a s_{f_2} (index 1 značí stavy v automatu aut_1, index 2 analogicky, ale v aut_2). Pokud je délka hrany z s_{a_1} do s_{f_1} rovna délce z s_{a_2} do s_{f_2}, pak je vše v pořádku a nic nemusíme řešit. V případě, že například délka hrany v prvním automatu je kratší, je nutné přidat mezistav s_{i_2} do druhého automatu, a to tak, že hranu z s_{a_2} do s_{f_2} odstraníme, přidáme hranu začínající v s_{a_2} a jdoucí do s_{i_2} přes symbol *symb* o stejné délce jakou má hrana s_{a_1} až s_{f_1}. Abychom odstraněnou hranu nahradili kompletně, musíme ještě přidat dvě hrany z s_{i_2} do s_{f_2} přes oba symboly $0, 1$, které budou mít délku původní hrany sníženou o vzdálenost z s_{a_1} do s_{f_1}. Dále budeme zpracovávat místo původní dvojice fs_1, fs_2, dvojici s_{f_1}, s_{i_2}.

Algoritmus vytváření mezistavů pro průnik (verze 1). Funkce *getEdgeLen* vrátí délku hrany mezi dvěma stavy předanými jako její parametry. Metody *remove_trans* a *add_trans* jsou celkem přímočaré, a tedy odstraní či přidají přechod z prvního parametru, přes symbol z druhého parametru do stavu v třetím parametru. Pokud má *add_trans* navíc čtvrtý parametr, znamená to, že nastavíme délku přidané hrany na jeho hodnotu, jinak

Algorithm 2 Vytváření mezistavů během průniku (verze 1)

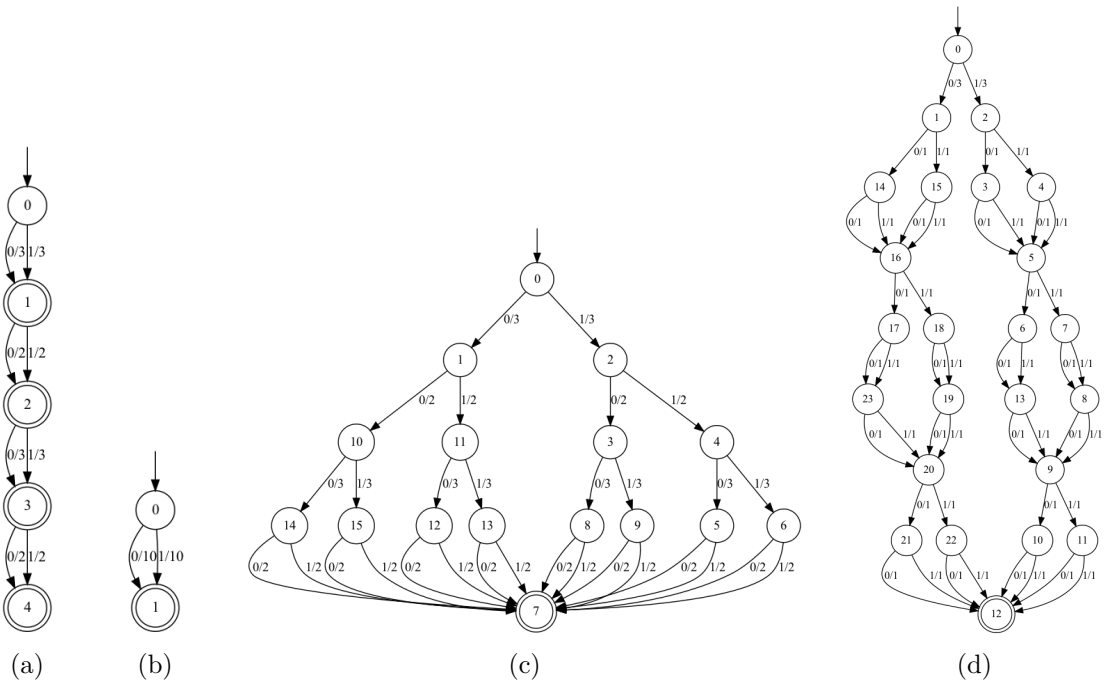
```
1:  procedure addInnersInter(aut_1, aut_2, s_{a_1}, s_{a_2}, s_{f_1}, s_{f_2}, symb)
2:      jump_1 ← getEdgeLen(s_{a_1}, s_{f_1})
3:      jump_2 ← getEdgeLen(s_{a_2}, s_{f_2})
4:      if jump_1 < jump_2 then
5:          aut_2.remove_trans(s_{a_2}, symb, s_{f_2})
6:          s_i ← aut_2.add_state()
7:          aut_2.add_trans(s_{a_2}, symb, s_i, jump_1)
8:          aut_2.add_one_jump_trans(s_i, 0, s_{f_2}, jump_2 − jump_1)
9:          aut_2.add_one_jump_trans(s_i, 1, s_{f_2}, jump_2 − jump_1)
10:         s_{f_2} ← s_i
11:     end if
12:     if jump_1 > jump_2 then
13:         aut_1.remove_trans(s_{a_1}, symb, s_{f_1})
14:         s_i ← aut_1.add_state()
15:         aut_1.add_trans(s_{a_1}, symb, s_i, jump_2)
16:         aut_1.add_one_jump_trans(s_i, 0, s_{f_1}, jump_1 − jump_2)
17:         aut_1.add_one_jump_trans(s_i, 1, s_{f_1}, jump_1 − jump_2)
18:         s_{f_1} ← s_i
19:     end if
20: end procedure
```

se použije implicitní délka hrany, která je 1. Metoda *add_one_jump_trans* má identické parametry jako *add_trans*, ale nevytvoří jednu hranu, nýbrž délka hrany -1 mezistavů a z počátečního stavu hrany do prvního mezistavu bude přechod se symbolem z parametrů metody a ostatní mezistavy (včetně koncového stavu hrany) budou propojeny oběma symboly abecedy $0, 1$. Intuitivně, nevytvoří se jedna skip hrana, ale analogie této skip hrany s použitím pouze běžných hran. Na první pohled to působí rozpačitě, protože by se mohlo zdát, že se naplno nevyužije potenciál skip hran, ovšem pokud na začátku algoritmu 2 dospějeme k situaci, že máme různé délky hran, tak se musí přidat mezistav a z něj kdybychom přidali pouze dvě hrany se symboly $0, 1$ do cílových stavů, obdrželi bychom nežádocí větvění do šířky ve výsledném automatu. Ukázku tohoto efektu můžeme vidět na obrázku 3.3. Abychom mohli skip hrany maximálně využít a zbavili se větvícího efektu, potřebujeme tyto mezistavy sdílet, pokud je to možné. Toho docílíme při použití druhé verze algoritmu *addInnersInter* vyjádřeného algoritmem 3.

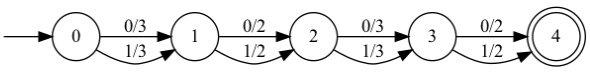
Algoritmus vytváření mezistavů pro průnik (verze 2). Zde se navíc nachází funkce *getInnerInter*(), která pokud již byl v automatu aut_2 (jestliže voláme z řádku 6) či aut_1 (jestliže voláme z řádku 12) vytvořen mezistav, ze kterého vede hrana délky $jump_2 - jump_1$ (invokace z řádku 6) nebo $jump_1 - jump_2$ (invokace z řádku 12) a směřuje do cílového stavu aktuálně zpracovávané hrany (přechodový symbol může být obecně různý), pak vrátí tento mezistav, jinak vytvoří nový mezistav, který oba symboly $0, 1$ propojí s cílovým stavem této hrany. Tento přístup nám zajistí sdílení mezistavů a tím pádem již nebude docházet k nežádoucímu větvení, což je demonstrováno na obrázku 3.4.

Algorithm 3 Vytváření mezistavů během průniku (verze 2)

1: **procedure** ADDINNERSINTER($aut_1, aut_2, s_{a_1}, s_{a_2}, s_{f_1}, s_{f_2}, symb$)
2: $jump_1 \leftarrow getEdgeLen(s_{a_1}, s_{f_1})$
3: $jump_2 \leftarrow getEdgeLen(s_{a_2}, s_{f_2})$
4: **if** $jump_1 < jump_2$ **then**
5: $aut_2.remove\_trans(s_{a_2}, symb, s_{f_2})$
6: $s_i \leftarrow getInnerInter()$
7: $aut_2.add\_trans(s_{a_2}, symb, s_i, jump_1)$
8: $s_{f_2} \leftarrow s_i$
9: **end if**
10: **if** $jump_1 > jump_2$ **then**
11: $aut_1.remove\_trans(s_{a_1}, symb, s_{f_1})$
12: $s_i \leftarrow getInnerInter()$
13: $aut_1.add\_trans(s_{a_1}, symb, s_i, jump_2)$
14: $s_{f_1} \leftarrow s_i$
15: **end if**
16: **end procedure**



Obrázek 3.3: Ukázka průniku automatů z obrázků 3.3a a 3.3b pomocí algortimu 2. Obrázek 3.3c vyobrazovává výsledek při použití maximálních možných délek skip hran (odpovídá použití metody *add_trans* místo *add_one_jump_trans* na řádcích $8, 9, 16, 17$) a obrázek 3.3d ukazuje výsledek za přidávání pouze hran délky 1 z mezistavů (tedy přesně dle algoritmu).



Obrázek 3.4: Vizualizace průniku automatů z obrázků 3.3a a 3.3b za použití algoritmu 3.

3.2.2 Determinizace

Stejně jako průnik, tak i determinizace je klasická automatová operace, tudíž jsme se rozhodli pro zjednodušenou formu zápisu částí algoritmu, které jsou shodné pro automaty s běžnými hranami i se skip hranami. Jedná se o jednu z nejčastěji využívaných operací, která se používá například během komplementace, projekce či minimalizace. Samotný pseudokód uvadíme v algoritmu 4.

Společný rámec determinizace. Algoritmus začne tím, že vloží do vektoru, sloužícího ke zpracovávání stavů, množinu iniciálních stavů automatu. Následně z vektoru vybere jednu položku, která odpovídá množině stavů původního automatu, zde značíme jako S_a. Podmínka na řádku 7 je splněna, pokud lze přejít z aktuální množiny stavů S_a (nebo-li existuje přechod z alespoň jednoho stavu obsaženého v množině stavů S_a do libovolného stavu, který poté bude obsažen v S_f, přes symbol *symb*) pomocí daného symbolu *symb* do nějaké množiny stavů, označované jako S_f. Analogicky jako u průniku, funkce $getNextMove()$ vrátí výše popsanou množinu S_f a přechodový symbol *symb*. Procedůra $addResultTrans()$ zjednodušeně řečeno zkontroluje, zda množina cílových stavů S_f již byla viděna ve výsledném automatu a pokud ano, přidá se pouze přechod z S_a přes *symb* do S_f, jinak je navíc vytvořen nový stav výsledného automatu, který se také vloží do vektoru q k dalšímu zpracování. V neposlední řadě, funkce *addInnersDet*, která slouží k přidávání mezistavů v momentu, kdy máme různé délky hran, bude popsána samostatně a ve dvou verzích algoritmy 5 a 6.

Algorithm 4 Determinizace automatu se skip hranami

```
1:  procedure DETERMINIZATION(aut)
2:      result ← emptyAutomaton()
3:      q ← emptyVector()
4:      q.push((aut.initial_states))
5:      while q is not empty do
6:          S_a ← q.pop()
7:          while exists next move from S_a do
8:              S_f, symb ← getNextMove()
9:              addInnersDet(aut, S_a, S_f, symb)
10:             addResultTrans()
11:         end while
12:     end while
13:     return result
14: end procedure
```

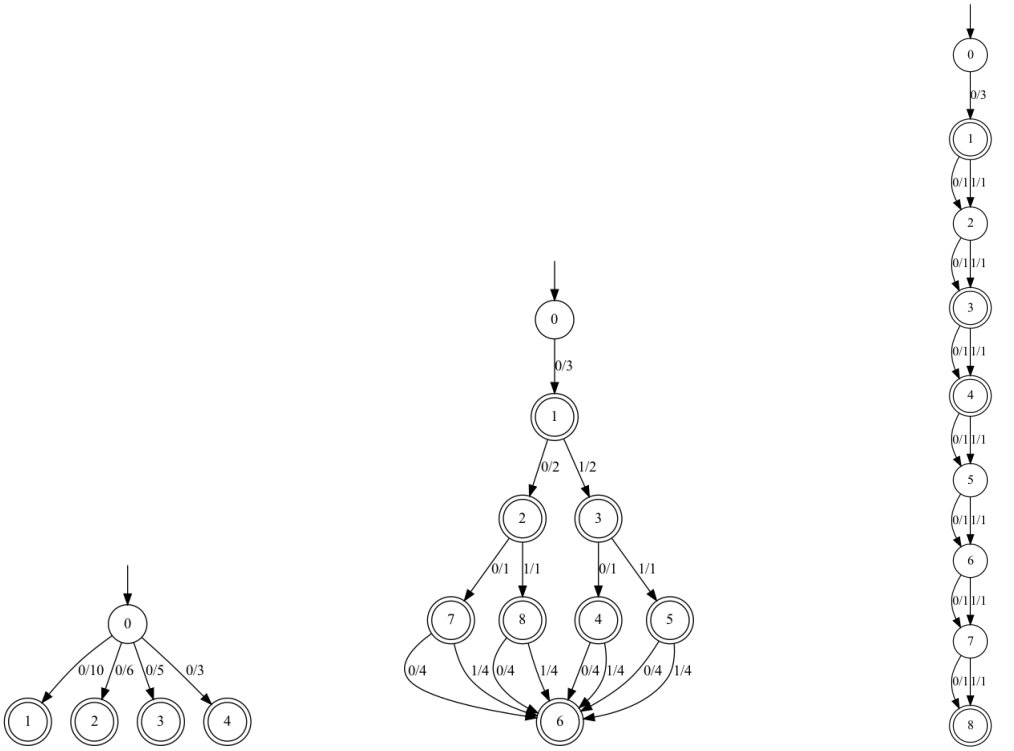
Algoritmus vytváření mezistavů pro determinizaci (verze 1). Intuice za algoritmem 5 je následující. Máme množinu stavů S_a (aktuální stavy), S_f (cílové stavy) a přechodový symbol *symb*. Jestliže by všechny hrany měly stejnou délku, pak není potřeba tvořit mezistavy. V opačném případě nejprve potřebujeme zjistit délku nejkratší hrany mezi množinami stavů, označenou jako *min_jump*. Hrany s touto délkou zůstávají beze změny, pro ostatní přidáme mezistavy s_{i_i} a nahradíme je hranami z s_{a_i} do s_{i_i} přes symbol *symb* o délce *min_jump* a také dvojicemi hran přes oba symboly $0, 1$ z s_{i_i} do s_{f_i} s délkami hran odstraněné hrany snížené o konstantu *min_jump*. Funkce *getShortestEdgeLen* bere

Algorithm 5 Vytváření mezistavů během determinizace (verze 1)

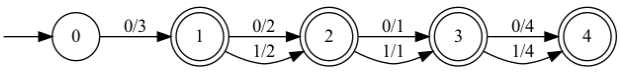
```
1:  procedure addInnersDet(aut, S_a, S_f, symb)
2:      S_fnew ← emptyVector()
3:      min_jump ← getShortestEdgeLen(S_a, S_f)
4:      for s_f in S_f do
5:          if s_f has not lowest edge length then
6:              for s_a in S_a do
7:                  if aut.has_trans(s_a, symb, s_f) then
8:                      aut.remove_trans(s_a, symb, s_f)
9:                      s_i ← aut.add_state()
10:                     aut.add_trans(s_a, symb, s_i, min_jump)
11:                     jump ← getEdgeLen(s_a, s_f)
12:                     aut.add_one_jump_trans(s_i, 0, s_f, jump − min_jump)
13:                     aut.add_one_jump_trans(s_i, 1, s_f, jump − min_jump)
14:                     S_fnew.push(s_i)
15:                 end if
16:             end for
17:         else
18:             S_fnew.push(s_f)
19:         end if
20:     end for
21:     S_f ← S_fnew
22: end procedure
```

za parametry dvě množiny stavů a vrátí délku nejkratší hrany z nějakého stavu z první množiny do nějakého stavu z druhé množiny. Podmínka při řádku 5 říká, že hrana jdoucí z nějakého stavu z množiny S_a do stavu s_f nemá délku rovnu hodnotě min_jump (tedy má ji větší a tím pádem to není žádná z nejkratších hran mezi S_a a S_f), což znamená, že pro ni nemáme tvořit mezistav. Ostatní uvedené funkce se již vyskytly v některém z předchozích algortimů, tudíž předpokládáme, že jejich význam je jasný. Uvedený algoritmus je principiálně podobný algoritmu 2 pro průnik dvou automatů, protože opět tvoříme mezistavy pro hrany s různou délkou a z těchto mezistavů již přidáváme pouze běžné hrany s délkou 1 (rozdílné oproti úvodnímu zjednodušenému intuitivnímu vysvětlení). Ovšem i v tomto případě mají hrany délky 1 svůj smysl, a to ten, abychom neobdrželi efekt nežádoucího větvení ve výsledném automatu. Ukázka efektu a práce algoritmu je na obrázku 3.5.

Algoritmus vytváření mezistavů pro determinizaci (verze 2). Ve vylepšeném algoritmu 6 se opět snažíme zbavit zbytečného větvení pomocí sdílení mezistavů. Myšlenka je taková, že pokud je do nějaké hrany z s_{a_i} do s_{f_i} přidán mezistav s_{i_i}, pak se do speciálního vektoru uloží informace o tom, že s_{i_i} je mezistav uvnitř s_{a_i} a s_{f_i}. Předveďme si tento přístup na konkrétním příkladu. Nechť existuje automat s dvěma stavy s_a a s_f, mezi kterými jsou dvě hrany délky 3 a 5 přes symbol 0 a dvě hrany délky 3 a 5 přes symbol 1. Determinizace započne stavem s_a a symbolem 0, kde se podívá do speciálního vektoru, jestli existuje mezistav s_{i_i}, kde s_f a s_{f_i} jsou tentýž stavy a zároveň hrany s_{i_i} až s_{f_i} a s_i až s_f mají stejnou délku. Takový mezistav zatím neexistuje, vytvoříme tedy nový a uložíme ho do vektoru. Jakmile přijde na řadu zpracování hrany z s_a do s_f přes symbol 1, tak již podmínka pro nalezení existujícího mezistavu bude splněna a tím pádem se nevy-



Obrázek 3.5: Ukázka determinizace automatu z obrázku 3.5a pomocí algoritmu 5. Obrázek 3.5b demonstruje výsledek determinizace za použití metody *add_trans* místo *add_one_jump_trans* na řádcích 12, 13 (což znamená použití maximálních možných délek skip hran) a obrázek 3.5c ukazuje výsledný automat přesně dle algoritmu (tedy s přidáváním pouze hran délky 1 z mezistavů).



Obrázek 3.6: Ukázka determinizace automatu z obrázku 3.5a pomocí algoritmu 6.

tvoří nový mezistav, ale použije se existující. Právě tuto funkcionalitu zajišťuje procedůra $getInnerDet()$. Demonstrace pozitivního efektu této optimalizace je na obrázku 3.6.

Algorithm 6 Vytváření mezistavů během determinizace (verze 2)

```
1:  procedure addInnersDet(aut, S_a, S_f, symb)
2:      S_fnew ← emptyVector()
3:      min_jump ← getShortestEdgeLen(S_a, S_f)
4:      for s_f in S_f do
5:          if s_f has not lowest edge length then
6:              for s_a in S_a do
7:                  if aut.has_trans(s_a, symb, s_f) then
8:                      aut.remove_trans(s_a, symb, s_f)
9:                      s_i ← getInnerDet()
10:                     aut.add_trans(s_a, symb, s_i, min_jump)
11:                     S_fnew.push(s_i)
12:                 end if
13:             end for
14:         else
15:             S_fnew.push(s_f)
16:         end if
17:     end for
18:     S_f ← S_fnew
19: end procedure
```

3.2.3 Komplement

Komplement pracuje tak, že daný automat zúplní a následně vymění akceptující stavy za neakceptující a obráceně. Toto ovšem pro naši reprezentaci není dostačující, protože skip hrana může přeskočit několik stavů, které nejsou akceptující a tedy tyto stavy se v automatu nevyskytují, což znamená, že v případě potřeby nemohou být změněny na akceptující. Z toho plyne, že budeme přidávat mezistavy, které budou koncové. Polaritu nemění všechny stavy, ale pouze ty do kterých suma délek hran jdoucích z libovolného počátečního stavu modulo počet proměnných ve formuli je rovna 0, nebo-li skončit můžeme, když je stejněkrát přiřazeno do všech proměnných. Komplementace je popsána v algoritmu 7.

Triviální případy. Nejprve řešíme triviální případy, které jsou dva. První je pokud máme automat s jedním iniciálním a zároveň akceptujícím stavem a žádné přechody. Toto je detekované podmínkou na řádku 2, kde *id_cnt* je aktuální počet proměnných ve formuli a metoda *lang_is_empty* vrací *true*, jestliže je jazyk automatu prázdný, jinak $false$. Tuto situaci musíme ošetřit separátně, protože algoritmus by v tomto případě začal tvořit hrany délky 0, což není povoleno. Druhý speciální případ je, když má vstupní automat prázdný jazyk, což odpovídá podmínce z řádku 5, pak stačí vrátit automat přijímající všechna přiřazení proměnných. Jak již vyplývá z názvu, funkce *codeTrue* vyrobí přesně takový automat, naopak procedůra $codeFalse$ vyrobí jednostavový automat bez koncových stavů.

Tělo komplementu. Nyní provedeme determinizaci vstupního automatu, abychom získali vstupní automat s jedním počátečním stavem. Metoda *add_sink_state* vytvoří jeden

Algorithm 7 Komplementace automatu se skip hranami

```
1: procedure COMPLEMENT(aut, id_cnt)
2:     if id_cnt = 0 and not aut.lang_is_empty() then
3:         return codeFalse()
4:     end if
5:     if aut.lang_is_empty() then
6:         return codeTrue(id_cnt)
7:     end if
8:     result ← emptyAutomaton(aut.num_states())
9:     result.add_sink_state()
10:    q ← emptyVector()
11:    level ← emptyArray(aut.num_states())
12:    aut ← determinization(aut)
13:    for init in aut.init_states() do
14:        level[init] ← 0
15:        result.make_initial(init)
16:        q.push(init)
17:    end for
18:    while q is not empty do
19:        s_a ← q.pop()
20:        for symb, s_f going from s_a in aut do
21:            addFinalStates(result, s_a, symb, s_f, level, id_cnt)
22:            if s_f was not seen then
23:                q.push(s_f)
24:            end if
25:        end for
26:        if does not exists 0 trans from s_a then
27:            result.add_trans(s_a, 0, s_sink, id_cnt − level[s_a])
28:        end if
29:        if does not exists 1 trans from s_a then
30:            result.add_trans(s_a, 1, s_sink, id_cnt − level[s_a])
31:        end if
32:    end while
33:    for s_f in aut.final_states do
34:        result.remove_final(s_f)
35:    end for
36:    return result
37: end procedure
```

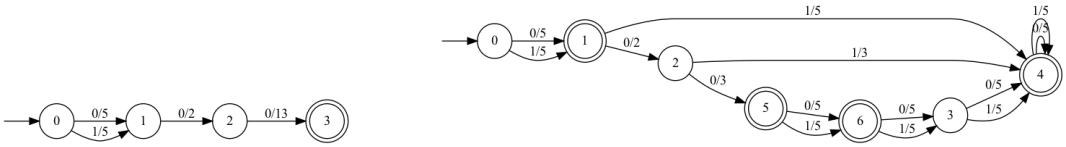
Algorithm 8 Vytvoří nové stavy a nastaví je na koncové

```
1: procedure addFinalStates(aut, s_a, symb, s_f, level, id_cnt)
2:     jump ← getEdgeLen(s_a, s_f)
3:     num_states ← aut.num_states()
4:     num_inner_states ← max(0, ⌈(jump+level[s_a]−id_cnt)/id_cnt⌉)
5:     aut.increase_size(num_states + num_inner_states)
6:     if num_inner_states = 0 then
7:         aut.add_trans(s_a, symb, s_f, jump)
8:     else
9:         s_{a_tmp} ← s_a
10:        s_{f_tmp} ← num_states
11:        num_states ← num_states + 1
12:        aut.add_trans(s_{a_tmp}, symb, s_{f_tmp}, id_cnt − level[s_a])
13:        for i = 0 to num_inner_states − 1 do
14:            s_{a_tmp} ← s_{f_tmp}
15:            s_{f_tmp} ← num_states
16:            num_states ← num_states + 1
17:            aut.add_trans(s_{a_tmp}, 0, s_{f_tmp}, id_cnt)
18:            aut.add_trans(s_{a_tmp}, 1, s_{f_tmp}, id_cnt)
19:            aut.make_final(s_{a_tmp})
20:        end for
21:        s_{a_tmp} ← s_{f_tmp}
22:        s_{f_tmp} ← s_f
23:        jump_tmp ← jump − ((num_inner_states − 1) ∗ id_cnt + (id_cnt − level[s_a]))
24:        aut.add_trans(s_{a_tmp}, 0, s_{f_tmp}, jump_tmp)
25:        aut.add_trans(s_{a_tmp}, 1, s_{f_tmp}, jump_tmp)
26:        aut.make_final(s_{a_tmp})
27:    end if
28:    level[s_f] ← (level[s_a] + jump) mod id_cnt
29:    if level[s_f] = 0 then
30:        aut.make_final(s_f)
31:    end if
32: end procedure
```

koncový stav se smyčkou dvou přechodů o délce počtu proměnných přes symboly $0, 1$, bežně nazývaný jako *sink stav*. Ještě před hlavním cyklem algoritmu provedeme potřebné inicializace, včetně nastavení pole *level*, jenž slouží k zapamatování si úrovně stavu, kterou definujeme jako součet úrovně libovolného předchůdce daného stavu a délky jejich propojující hrany, to celé modulo počet proměnných ve formuli, nebo-li vzorcem $level[s_f] = (level[s_a] + getEdgeLen(s_a, s_f)) \mod id\_cnt$ s tím, že iniciální stavy mají úroveň 0. Analogicky můžeme úroveň stavu definovat pomocí následníků a implicitní úrovně 0 koncových stavů. V hlavním cyklu komplementace poté postupně procházíme hrany (viz podmínka na řádku 20, která říká vrať všechny hrany reprezentované dvojicemi symbol *symb* a cílový stav s_f, které vedou ze stavu s_a v automatu *aut*), jenž předáme funkci *addFinalStates*.

Vytváření mezistavů pro komplement. Procedůra *addFinalStates* do hrany přidá akceptující mezistavy, pokud tato hrana přeskakuje nějaké stavy s úrovní 0. Přesněji kolik

stavů úrovně 0 hrana přeskočí, tolik koncových mezistavů přidáme. Zde již plně využíváme pole úrovní stavů, protože k vytváření mezistavů potřebujeme vědět úroveň aktuálního stavu a délku hrany z aktuálního do cílového stavu. V druhé části hlavního cyklu poté realizujeme zúplnění a na závěr ještě ve výsledném automatu změníme stavy odpovídající koncovým, ve vstupním automatu, na nekoncové. Vizualizace této procedůry je na obrázku 3.7.

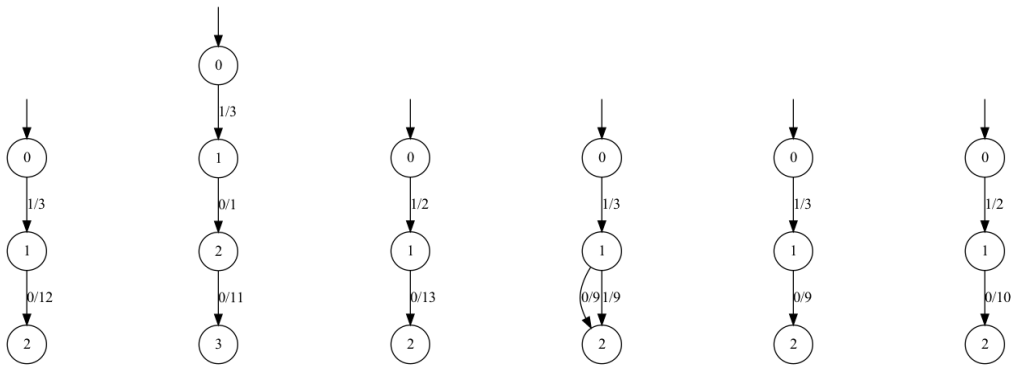


(a) (b)

Obrázek 3.7: Vizualizace činnosti funkce *complement* pro 5 proměnných. Na obrázku 3.7a vidíme hranu před a na obrázku 3.7b po zavolání procedůry. Počáteční stav 0 slouží pro booleovské proměnné a nenegujeme ho.

3.2.4 Projekce

Jako další automatovou operaci představíme projekci. Myšlenka je taková, že odstraníme určitou proměnnou x z automatu, reprezentovanou jejím indexem i, což v terminologii úrovní stavů znamená vymazat všechny přechody ze stavů úrovně i do stavů úrovně $i+1$. V rámci skip hran mohou během tohoto odstranění nastat tři situace, které rozebereme v rámci popisu algoritmu. Další obtíž je, že takto získaný automat je obecně nedeterministický, proto v této fázi musí být aplikována determinizace. Následně ještě nastavíme všechny stavy, ze kterých se lze dostat do libovolného koncového stavu přes symbol 0, jako koncové. Pseudokód projekce je uveden v algoritmu 10.



(a) (b) (c) (d) (e) (f)

Obrázek 3.8: Vizualizace projekce proměnné (algoritmus 10) s indexem 3 z celkového počtu 5ti proměnných pro tři různé vstupní automaty. Obrázek 3.8a představuje případ podmínky z řádku 24, jehož výsledný automat je na obrázku 3.8d. Analogicky pro dvojice obrázků 3.8b, 3.8e (podmínka z řádku 27) a 3.8c, 3.8f (případ na řádku 29).

Triviální případy. Začínáme opět se dvěma speciálními případy. Jelikož se nemůže stát, že by byla projekce volána s nulovým počtem proměnných ve formuli, budeme řešit případy s jednou proměnnou, a to konkrétně pokud je navíc jazyk vstupního automatu neprázdný, pak vracíme automat s jedním iniciálním a zároveň akceptujícím stavem, jinak vyrobíme automat s jedním iniciálním stavem.

Algorithm 9 Nastaví stavy, ze kterých se dá dostat po 0 do koncového stavu, na koncové

```
 1: procedure removeUselessZeros(aut, id_cnt)
 2:     preds ← emptyArray(aut.num_states())
 3:     for s_a in aut.states do
 4:         for symb, s_f going from s_a in aut do
 5:             preds[s_f].push((symb, s_a))
 6:         end for
 7:     end for
 8:     q ← emptyVector()
 9:     level ← emptyArray(aut.num_states())
10:     for final in aut.final_states do
11:         level[final] ← 0
12:         q.push(final)
13:     end for
14:     while q is not empty do
15:         s_f ← q.pop()
16:         for p in preds[s_f] do
17:             addFinalStatesRev(aut, p.second, p.first, s_f, level, id_cnt)
18:             if p.first = 0 and p.second was not seen then
19:                 q.push(p.second)
20:             end if
21:         end for
22:     end while
23:     return aut
24: end procedure
```

Odstranění proměnné. Na rozdíl od dříve zmíněných algoritmů je zde navíc vektor *edges_to_be_removed* sloužící k uchovávání dvojic stavů, kde všechny hrany jdoucí do prvního stavu, přesměrujeme do druhého stavu. Zmíněné přesměrování se používá, pokud odstraňujeme hranu délky 1. Funkce *computeEdgeLen* zkrátí délku hrany o konstantu, jenž odpovídá počtu uzlů úrovně i přeskočených danou hranou. Následně v hlavním cyklu algoritmu tradičně procházíme všechny hrany, přičemž mohou nastat tři situace (v pseudokódu reprezentováno podmínkami na řádcích $24, 27, 29$). Úvodní případ znamená, že index i projektované proměnné x je roven úrovni aktuálního stavu a délka hrany je větší než 1. Intuitivní význam je, že projektovaná proměnná je na začátku skip hrany. Prakticky to znamená, že musíme tuto hranu nahradit dvěma hranami přes symboly $0, 1$ s délkami sníženými o počet přeskočených uzlů úrovně i původní hranou. Druhá situace je podobná předchozí, akorát se jedná o běžnou hranu, tedy její délka je 1. Toto je typická situace vyskytující se i v reprezentaci bez skip hran, takže už budeme muset hranu odstranit a přesměrovat všechny hrany, směřující do jejího počátečního stavu, do jejího cílového stavu. Na to

Algorithm 10 Odstranění proměnné v automatu se skip hranami

```
1: procedure PROJECT(aut, x, id_cnt)
2:     if id_cnt = 1 and not aut.lang_is_empty() then
3:         return codeTrue(0)
4:     end if
5:     if id_cnt = 1 and aut.lang_is_empty() then
6:         return codeFalse()
7:     end if
8:     result ← emptyAutomaton(aut.num_states())
9:     result.initial_states ← aut.initial_states
10:    result.final_states ← aut.final_states
11:    q ← emptyVector()
12:    level ← emptyArray(aut.num_states())
13:    edges_to_be_removed ← emptyVector()
14:    for init in aut.initial_states do
15:        level[init] ← 0
16:        q.push(init)
17:    end for
18:    while q is not empty do
19:        s_a ← q.pop()
20:        for symb, s_f going from s_a in aut do
21:            jump ← getEdgeLen(s_a, s_f)
22:            edge_len ← computeEdgeLen(level[s_a], jump, x, id_cnt)
23:            level[s_f] ← (level[s_a] + jump) mod id_cnt
24:            if level[s_a] = x and jump > 1 then
25:                result.add_trans(s_a, 0, s_f, edge_len)
26:                result.add_trans(s_a, 1, s_f, edge_len)
27:            else if level[s_a] = x and jump = 1 then
28:                edges_to_be_removed.push((s_a, s_f))
29:            else
30:                result.add_trans(s_a, symb, s_f, edge_len)
31:            end if
32:            if s_f was not seen then
33:                q.push(s_f)
34:            end if
35:        end for
36:    end while
37:    result ← redirectEdges(result, edges_to_be_removed)
38:    result ← determinize(result)
39:    result ← removeUselessZeros(result, id_cnt − 1)
40:    return result
41: end procedure
```

Algorithm 11 Přidání koncových stavů mezi stavy s_a a s_f

```
1: procedure addFinalStatesRev(aut, s_a, symb, s_f, level, id_cnt)
2:     jump ← getEdgeLen(s_a, s_f)
3:     num_states ← aut.num_states()
4:     num_inner_states ← max(0, ⌈(jump+level[s_f]−id_cnt)/id_cnt⌉)
5:     aut.increase_size(num_states + num_inner_states)
6:     if num_inner_states > 0 then
7:         s_atmp ← num_states
8:         s_ftmp ← s_f
9:         num_states ← num_states + 1
10:        aut.add_trans(s_atmp, 0, s_ftmp, id_cnt − level[s_f])
11:        aut.add_trans(s_atmp, 1, s_ftmp, id_cnt − level[s_f])
12:        aut.make_final(s_atmp)
13:        for i = 0 to num_inner_states − 1 do
14:            s_ftmp ← s_atmp
15:            s_atmp ← num_states
16:            num_states ← num_states + 1
17:            aut.add_trans(s_atmp, 0, s_ftmp, id_cnt)
18:            aut.add_trans(s_atmp, 1, s_ftmp, id_cnt)
19:            aut.make_final(s_atmp)
20:        end for
21:        s_ftmp ← s_atmp
22:        s_atmp ← s_a
23:        jump_tmp ← jump − ((num_inner_states − 1) ∗ id_cnt + (id_cnt − level[s_f]))
24:        aut.add_trans(s_atmp, symb, s_ftmp, jump_tmp)
25:        aut.remove_trans(s_a, symb, s_f)
26:    end if
27:    level[s_a] ← (level[s_f] + jump) mod id_cnt
28:    if level[s_a] = 0 and symb = 0 and s_a is not initial then
29:        aut.make_final(s_a)
30:    end if
31: end procedure
```

použijeme již zmíněný vektor *edges_to_be_removed*. Důvod proč neodstraňujeme hrany teď je, protože iterujeme přes vstupní automat a hrany přidáváme do nového automatu, tím pádem nemáme zaručeno, že jsme již viděli všechny hrany jdoucí do aktuálního stavu a také některé hrany směřující do aktuálního stavu budou později odstraněny a nahrazeny novými, což by se opět neprojevilo ve výsledném automatu. Poslední případ představuje situaci, kdy i a úroveň aktuálního stavu jsou různé. Neformálně řečeno, projektovaná proměnná není na začátku skip hrany. Zde stačí snížit délku hrany o počet přeskočených uzlů úrovně i. Tyto tři případy jsou přehledně vizualizovány na obrázku 3.8. Poté funkce *redirectEdges* realizuje podstatu vektoru *edges_to_be_removed*, tedy provede zmíněné přesměrování hran.

Odstranění zbytečných nul. V neposlední řadě ještě procedůrou *removeUselessZeros* nastavíme všechny stavy ze kterých se lze dostat do koncových stavů pomocí libovolného počtu přechodových symbolů 0, jako koncové. Ta je implementována tak, že nejprve do pole *preds* vypočte všechny předchůdce všech stavů spolu s příslušnými přechodovými

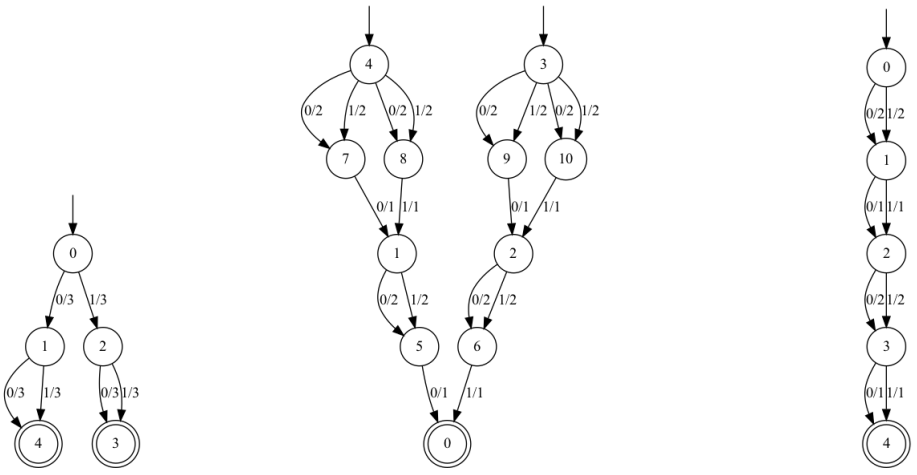
symboly a následně zpětně iteruje automatem, tedy směrem z koncových do počátečních stavů a všechny hrany jdoucí z předchůdců do aktuálního stavu rozmělní mezistavy, podobně jako při komplementaci. Tato optimalizace slouží k tomu, abychom nemuseli několikrát prohledávat celý automat při hledání předchozích stavů, ale pouze jedenkrát. Procedůra $addFinalStatesRev$ realizuje zmiňované přidání mezistavů obdobným způsobem jako $addFinalStates$ v algoritmu 7, akorát v opačném směru.

3.2.5 Minimalizace

Jako minimalizační algoritmus jsme zvolili Brzozowski algoritmus doplněný o redukci běžných hran na skip hrany, a to z důvodu, že se již nachází v námi používané knihovně Mata a patří mezi používané, ovšem může u něj dojít ke stavové explozi. Jedná se o dvojitou reverzaci následovanou determinizací, což je formálněji vyjádřeno algoritmem 12.

Algorithm 12 Minimalizace vstupního automatu

```
1: procedure MINIMIZATION(aut)
2:     aut = revert(aut)
3:     aut = determinization(aut)
4:     aut = revert(aut)
5:     aut = determinization(aut)
6:     aut = reduceToSkipEdges(aut)
7:     return aut
8: end procedure
```



(a) (b) (c)

Obrázek 3.9: Vizualizace průběhu reverzace automatu 3.9a, kde na obrázku 3.9b je reverzovaný s přidanými nesdílenými mezistavy $5, 6, 7, 8, 9, 10$ a následně determinizovaný na obrázku 3.9c. Množiny stavů během determinizace byly $\{3, 4\}$, $\{7, 8, 9, 10\}$, $\{1, 2\}$, $\{5, 6\}$, $\{0\}$, kde počet uvedených množin by zůstal stejný i při použití sdílených mezistavů, tedy determinizace by trvala stejný počet kroků.

Reverzace. Funkce *revert* (algoritmus 13), jak již název napovídá, provede reverzaci automatu. Pro hrany délky 1 je algoritmus totožný s klasickou reverzací, jinak se od běžné reverzace lišíme tím, že pro hranu délky n jdoucí ze stavu s_a do s_f přes symbol *symb* vytvoříme mezistav s_i a nahradíme ji hranami přes oba symboly $0, 1$ z s_f do s_i délky $n-1$ a hranou z s_i do s_a přes symbol *symb* o délce 1. Díky tomu, že po reverzaci přijde vždy determinizace, tak není nutné sdílet mezistavy (např. jako v algoritmech 3 a 6 pro průnik či determinizaci), protože determinizace sjednotí nesdílené mezistavy do jednoho stejným způsobem, jako kdyby byl sdílený, což můžeme vidět na obrázku 3.9.

Algorithm 13 Reverzace vstupního automatu

```
1:  procedure REVERT(aut)
2:      result ← emptyAutomaton(aut.num_states())
3:      result.initial_states ← aut.final_states
4:      result.final_states ← aut.initial_states
5:      for s_a in aut.states do
6:          for symb, s_f going from s_a in aut do
7:              jump ← getEdgeLen(s_a, s_f)
8:              if jump = 1 then
9:                  result.add_trans(s_f, symb, s_a)
10:             else
11:                 s_i ← result.add_state()
12:                 result.add_trans(s_f, 0, s_i, jump − 1)
13:                 result.add_trans(s_f, 1, s_i, jump − 1)
14:                 result.add_trans(s_i, symb, s_a)
15:             end if
16:         end for
17:     end for
18:     return result
19: end procedure
```

Redukce na skip hrany. Hlavní odlišnost oproti běžné minimalizaci je použití funkce *reduceToSkipEdges*, popsané algoritmem 14. Hlavní myšlenka této procedůry je nahradit hrany v automatu za skip hrany s co nejvetší možnou délkou a díky tomu snížit celkový počet stavů. Na začátku si pomocí funkce *getPredCount* spočítáme počet předcházejících stavů pro každý stav. Poté iterujeme skrz hrany automatu a voláme na ně funkci *addSkipEdge*, která danou hranu prodlouží na maximální možnou délku. Tato procedůra je vytvořená pouze pro deterministické konečné automaty, což nevadí protože redukce na skip hrany je aplikována právě po determinizaci. Nicméně vytvořili jsme také nedeterministickou verzi ekvivalentního algoritmu s vidinou toho, že bychom ji aplikovali mezi reverzací a determinizací, ale ta se prokázala jako neefektivní. Funkce *addSkipEdge* (algoritmus 15) funguje tak, že v nekonečném cyklu máme první ukončující podmínku říkající, že nově vytvořená skip hrana se už nemůže dále prodloužit, pokud z aktuálního stavu již žádná hrana nevede nebo je koncový (koncový stav nelze přeskočit, protože bychom o něj přišli) nebo už je v aktuálně zpracovávané hraně obsažen. Dále funkce *getNextState* vrací libovolný stav do kterého vede hrana ze stavu uvedeného jako parametr. Druhá podmínka zní, že z aktuálního stavu do takto získaného musí vést přechody přes oba symboly $0, 1$ a aktuální stav má méně než 2 předchůdce. Pokud není splněna, pak výroba hrany končí. V tomto okamžiku

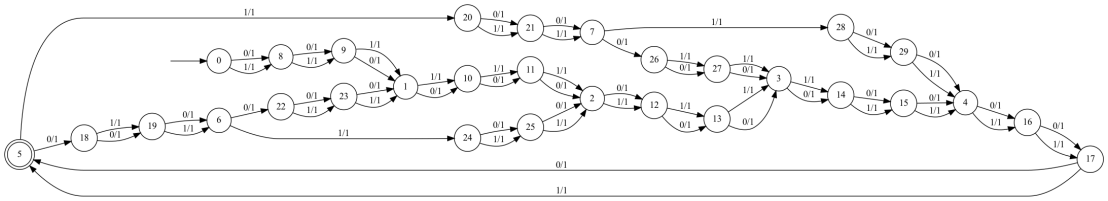
Algorithm 14 Nahradí všechny hrany za co nejdelší skip hrany

```
1: procedure reduceToSkipEdges(aut)
2:     result ← emptyAutomaton(aut.num_states())
3:     result.initial_states ← aut.initial_states
4:     result.final_states ← aut.final_states
5:     q ← emptyVector()
6:     pred ← getPredCount(aut)
7:     for init in aut.initial_states do
8:         q.push(init)
9:     end for
10:    while q is not empty do
11:        s_a ← q.pop()
12:        for symb, s_ftmp going from s_a in aut do
13:            jump ← getEdgeLen(s_a, s_ftmp)
14:            s_f, jump ← addSkipEdge(aut, s_ftmp, jump, pred)
15:            result.add_trans(s_a, symb, s_f, jump)
16:            if s_f was not seen then
17:                q.push(s_f)
18:            end if
19:        end for
20:    end while
21:    return result
22: end procedure
```

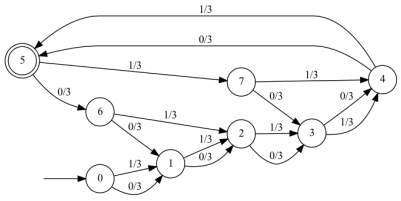
Algorithm 15 Vypočítá délku skip hrany a cílový stav

```
1: procedure addSkipEdge(aut, init_state, init_jump, pred)
2:     edge_len ← init_jump
3:     s_a ← init_state
4:     while true do
5:         if does not exists trans from s_a or s_a is final or s_a was seen then
6:             return s_a, edge_len
7:         end if
8:         s_f ← getNextState(s_a)
9:         if aut.has_trans(s_a, 0, s_f) and aut.has_trans(s_a, 1, s_f) and pred[s_a] < 2 then
10:            s_a ← s_f
11:            jump ← getEdgeLen(s_a, s_f)
12:            edge_len ← edge_len + jump
13:        else
14:            return s_a, edge_len
15:        end if
16:    end while
17: end procedure
```

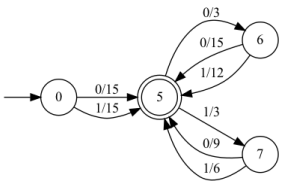
máme hotový algoritmus, jenž vyrobí maximálně dlouhé skip hrany mezi všemi dvojicemi stavů s méně než dvěma předchůdci, což na první pohled zní uspokojivě, ale po bližším zkoumání jsme zjistili, že při rozvolnění podmínky o počtu předchůdců (část $pred[s_a] < 2$ na řádku 9 v algoritmu 15) budou sice některé skip hrany delší o příslušnou konstantu odpovídající délce společné cesty v původním automatu, ale získáme menší celkový počet stavů. Z těchto důvodů jsme se rozhodli dále používat druhý přístup (bez podmínky na předchůdce stavů). Porovnání obou přístupů k redukci na skip hrany můžeme vidět na obrázku 3.10.



(a)



(b)



(c)

Obrázek 3.10: Demonstrace redukce na skip hrany automatu z obrázku 3.10a. Na obrázku 3.10b je verze algoritmu s podmínkou na počet předchůdců a na obrázku 3.10c je verze bez této podmínky.

Kapitola 4

Implementace

V naší implementaci používáme překladač formulí a abstraktní syntaktický strom z nástroje Mona (zdrojové soubory z `/src/ast` a `/include`), kam jsme přidali funkce pro výrobu automatů. Pro práci s konečnými automaty jsme využili knihovnu Mata[1] (zdrojové soubory z `/src/mata` a `/include`), do které jsme přidali implementaci skip hran a upravili tradiční automatové algoritmy pro efektivní práci s naší reprezentací. V adresáři `/src/core` jsou naše vlastní zdrojové kódy pro řízení transformace formule na automat, zbylé automatové operace a statistiky.

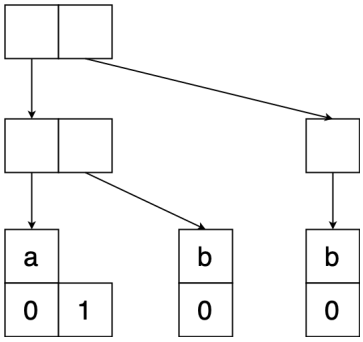
4.1 Mata reprezentace automatů

Předtím než popíšeme implementaci složitějších součástí, rozebereme jak vypadá reprezentace konečného automatu v knihovně Mata. Přechodová relace je vektor obsahující *Moves* o délce počtu stavů automatu, kde každý index i obsahuje přechody automatu ze stavu i, což je označno jako Moves. Moves je poté vektor složený z *Move*, který je seřazený podle přechodového symbolu a každá jeho položka představuje přechod přes právě jeden symbol. Konečně, Move je struktura obsahující přechodový symbol (nezáporné číslo) a vektor cílových stavů, opět seřazený. Vidíme, že zmíněná datová struktura je vytvořena obecně pro nedeterministické konečné automaty. Počátěční a koncové stavy jsou navíc ještě uloženy v samostatných dvou vektorech. Grafická ukázka je na obrázku 4.1.

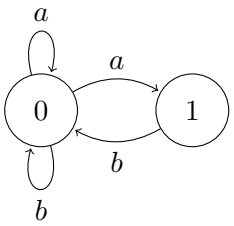
4.2 Skip hrany

Datovou strukturu konečného automatu jsme upravili tak, že vektor cílových stavů již neobsahuje cílové stavy, ale obsahuje dvojice cílový stav a délka hrany. Tento přístup odpovídá přidání informace o přeskakování ke hranám. Další možnost byla přiřadit tuto informaci, modulovanou, ke stavům a dát jim úroveň, pak bychom ale dostali chování totožné s BDD, tedy maximální délka hrany by odpovídala počtu proměnných, protože pokud bychom měli přechod z úrovně 0 do úrovně 2, tak nevíme jestli je délka této hrany 2 nebo počet proměnných +2. Jestliže bychom chtěli úroveň udělat nemodulovanou, narazíme na to, že pokud bude na nějakém stavu smyčka, tak by jeho úroveň byla nekonstantní. Ukázka datové struktury pro jednoduchý automat je na obrázku 4.2.

[1] https://github.com/VeriFIT/mata

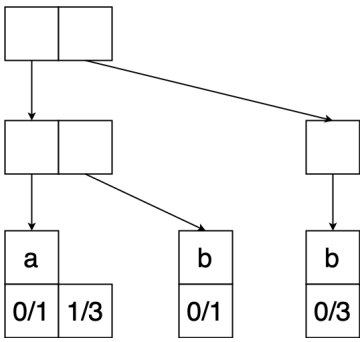


(a) Mata reprezentace automatu.

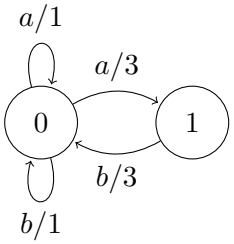


(b) Konečný automat.

Obrázek 4.1: Vizualizace datové struktury konečného automatu v knihovně Mata. Jako symboly jsou z důvodu přehlednosti zvolena písmena a, b, ovšem reálně mohou symboly být pouze nezáporná čísla.



(a) Mata reprezentace automatu obohacená o skip hrany.



(b) Konečný automat se skip hranami.

Obrázek 4.2: Ilustrace konečného automatu se skip hranami a jeho implementace s využitím automatové knihovny Mata. V této knihovně mohou být symboly pouze nezáporná čísla, ale pro větší přehlednost jsou na obrázku zvolena písmena a, b.

4.3 Kostra převodu formule na automat

Z nástroje Mona jsme převzali překladač formulí, tabulku predikátů, tabulku symbolů a abstraktní syntaktický strom. Tím pádem WS1S formule na vstupu našeho programu má identickou syntax jako v nástroji Mona. Výpočet formule probíhá tedy klasicky, a to od atomických formulí směrem k hlavní formuli. Pro zvýšení efektivity jsme se rozhodli v každém kontextu uvažovat pouze proměnné, které jsou relevantní. Například formule $(\exists y : y = x) \wedge (\exists z : z = x)$ obsahuje celkově tři proměnné x, y, z, ale v části vlevo od konjunkce lze uvažovat pouze x, y, podobně ve zbylé části x, z. V tabulce symbolů se proměnným přiřazuje nezáporné číslo, označované jako *id*. Nejprve se přiřadí globálním proměnným (proměnné zadefinované na začátku programu) a poté proměnným vzniklým kvantifikací $(\exists, \forall)$, oběma vzestupně dle výskytu ve formuli. Ve výše uvedené formuli by *id* vypadala následovně: $(\exists 1 : 1 = 0) \wedge (\exists 2 : 2 = 0)$. Toto může vyústit v problémy, protože v pravé části formule máme dvě proměnné x, z, ale *id* jedné z nich je vyšší než celkový počet proměnných, proto jsme se rozhodli proměnné přemapovávat. Vždy při kvantifikaci přiřadíme nově vzniklým proměnným *id*, které bude rovno aktuálnímu počtu proměnných v

daném kontextu, nebo-li bude o 1 vyšší než aktuální nejvyšší *id* proměnné. Toto mapování, implementované třídou *remapVars*, bude procházet napříč generováním formule a sbírat potřebná přemapování. Seznam přemapování se využije když narazíme na použití proměnných v nějaké podformuli, a to tak, že v podformuli nahradíme reálné *id* proměnné za přemapované.

4.4 Popis průchodu formulí

Samotná transformace formule na automat se skládá z jednoho průchodu abstraktním syntaktickým stromem, reprezentovaným třídou AST a třídami dědícími z AST, které představují už jednotlivé termy z WS1S. Každá takováto třída pak obsahuje metodu *genCode*, která bere jako parametry aktuální počet proměnných, seznam přemapování, proměnnou určenou ke sdílení informace a následně z daného termu vytvoří automat. Průchod abstraktním syntaktickým stromem je rozdělen na dvě fáze, zanoření (směr z hlavní formule do atomických formulí) a vynoření (směr z atomických formulí do hlavní formule). Pro další vysvětlení potřebujeme termy rozřadit do šesti kategorií:

- Termy nultého (formule) řádu s argumenty nultého řádu bez kvantifikování (např. $\wedge, \neg$)
- Kvantifikování s argumenty nultého (formule) řádu (např. $\exists, \forall$)
- Proměnné (např. x)
- Termy nultého (formule) řádu s argumenty prvního (čísla) a druhého (množiny) řádu (např. $<, \subseteq, \in$)
- Termy prvního (čísla) a druhého (množiny) řádu (např. $+, \cup$)
- Predikáty (např. $xor(x, y) = (x \wedge \neg y) \vee (\neg x \wedge y)$)

Termy nultého řádu s argumenty nultého řádu bez kvantifikování. Zde je situace velmi jednoduchá, stačí pouze zavolat tvorbu automatů pro všechny argumenty s nezměněnými parametry a poté provést příslušnou automatovou operaci.

Kvantifikování s argumenty nultého řádu. Jak již bylo popsáno, v naší implementaci je potřeba měnit *id* proměnných podle aktuálního kontextu. Pro tento učel byla vytvořena třída *remapVars*, což je vektor tvořený z *id* proměnných, kde na i-té pozici je původní *id* proměnné a na $i + 1$-té pozici je nově vzniklé *id*. V tomto případě nám vznikají proměnné, které dosud nebyly uvažovány, tím pádem při zanoření vytvoříme všem novým proměnným přemapování takové, že první kvantifikovaná proměnná bude namapována na hodnotu aktuálního počtu proměnných, druhá na aktuální počet proměnných +1 a analogicky zbytek. Dále zvýšíme aktuální počet proměnných o počet kvantifikovaných proměnných a spolu s aktualizovaným přemapováním zavoláme funkci na generování automatu z formule, která následuje. V rámci vynoření pak kvantifikované proměnné odprojektujeme, ovšem pokud se jedná o univerzální kvantifikaci, pak zkomplementujeme automat vzniklý z dané formule, odprojektujeme proměnné a nakonec znovu aplikujeme komplement. Další omezení nastává u prvořádových proměnných, kdy před každou projekcí je konjunkcí přidán singleton automat.

Proměnné. Tento případ je triviální, ale dá se na něm demonstrovat využití přemapování proměnných. Pokud při průchodu abstraktního syntaktického stromu dojdeme na nejnižší úroveň, tedy k samotné proměnné, přistoupíme k jejímu *id*, ovšem nyní musíme projít seznam přemapování, abychom následně dostali správné *id* vzhledem k aktuálnímu kontextu. U proměnných nultého řádu pouze zavoláme funkci na tvorbu atomického automatu pro přemapovanou proměnnou, u prvního a druhého řádu zavoláme procedůru na rovnost přemapované proměnné a proměnné pro sdílení informace. To je z důvodu, že je nelze chápat jako samostatné formule, ale jejich výskyt je vždy ve formulích s argumenty těchto řádů, pro které proměnnou pro sdílení informace využíváme.

Termy nultého řádu s argumenty prvního a druhého řádu. Při zanoření se na oba dva argumenty zavolá metoda *getDepth*, která spočítá hloubku zanoření pro obě strany formule, nebo-li kolik pomocných proměnných je potřeba vytvořit. Problematiku uvedeme na příkladu. Nechť máme formuli s argumenty prvního řádu $x + 1 + 2 = y + 2$. Levá strana formule se převede do tvaru $x_0 = x_1 + 1 \wedge x_1 = x_2 + 2 \wedge x_2 = x$, pravá obdobně na $y_0 = y_1 + 2 \wedge y_1 = y$. Metoda *getDepth* tedy v tomto konkrétním případu spočítá, že je potřeba vytvořit 3 pomocné proměnné pro levou stranu a 2 pro stranu pravou. Je nutné tento krok udělat nyní, protože zmíněné hodnoty potřebujeme vědět pro výpočet obou stran formule. Následně se zavolá tvorba automatu pro levou stranu formule s tím, že se nastaví proměnná pro sdílení informace na aktuální počet proměnných, tedy 2, což představuje proměnnou x_0. Analogicky pro pravou část, ovšem proměnná pro sdílení informace bude rovna aktuálnímu počtu proměnných zvětšenému o počet pomocných proměnných levé strany, tedy $2 + 3 = 5$, a to značí proměnnou y_0. K těmto voláním ještě přiřadíme nový aktuální počet proměnných, který se vypočítá přičtením počtu pomocných proměnných k současnému, který činí 2, tedy výsledek bude $2 + 2 + 3 = 7$. Až se vynoříme, vytvoříme atomický automat pro rovnost proměnných x_0 a y_0, zprůnikujeme ho s výslednými automaty pro obě strany a odprojektujeme všechny pomocné proměnné x_0, x_1, x_2, y_0, y_1.

Termy prvního a druhého řádu. Ve fázi zanoření se podíváme na proměnnou pro sdílení informace, označme ji jako x a vytvoříme další proměnnou y s *id* o 1 větším než x. Nyní zavoláme tvorbu automatu pro argument termu (např. další vnořené $+, -$) s tím, že proměnnou pro sdílení informace nastavíme na y. Při vynoření uděláme průnik získaného automatu s daným atomickým automatem (např. $+, -$) pro x a y.

Predikáty. Predikáty jsou v nástroji Mona způsoby jak parametrizovat a znovupoužívat formule. Jako parametr, kromě proměnné, může figurovat libovolný term řádu odpovídajícímu definici predikátu. Například, pokud predikát foo má jeden prvořádový parametr, lze ho volat třeba jako $foo(x)$ i $foo(x + 2)$. Tvorba predikátů je často využívána a z toho důvodu je podporována i v našem nástroji. Způsob výpočtu budeme ilustrovat na příkladu s predikátem $multi\_eq(x, y, z) = x = y \wedge y = z$ a formulí $multi\_eq(a, b, b + 1)$. Nejprve při zanoření potřebujeme určit, zda je potřeba vytvářet pomocné proměnné. Iterujeme tedy přes parametry volání a pokud je parametr jen proměnná, není třeba vytvářet pomocnou a stačí ji jen přemapovat, stejně jako když se v běžné formuli narazí na proměnnou, jinak musíme vytvořit pomocnou proměnnou. Nyní budeme požadovat, aby kdykoliv, kdy narazíme v predikátu na proměnnou z jeho definice, tak aby se přemapovala na proměnnou z volání či pomocnou proměnnou, tedy ještě takové přemapování přidáme. Uvedená formule, okleštěná o volání predikátu, bude vypadat takto: $a = b \wedge b = x_0$, kde x_0 je vytvořená pomocná proměnná. Dále zavoláme funkci na tvorbu automatu pro výše uvedenou

formuli. Během fáze vynoření musíme všechny pomocné proměnné položit rovno jejím volaným parametrům, tedy v tomto příkladu by se nám formule rozrostla o jednu konjunkci: $a = b \wedge b = x_0 \wedge x_0 = b+1$. Konečně, jestě musíme všechny pomocné proměnné prvního řádu konjunktovat se singleton automatem a úplně všechny pomocné proměnné odprojektovat.

Kapitola 5

Experimenty

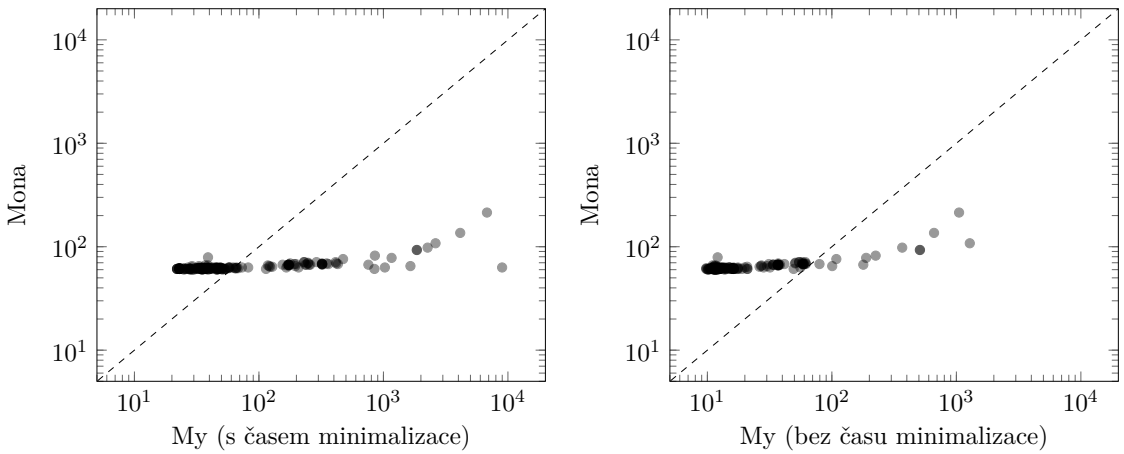
Pro otestování naší rozhodovací procedůry a změření její efektivity jsme použili testovací množinu čítající 196 WS1S formulí, rozdělených do následujících 11 kategorií: basic (104), horn-sub (6), horn-trans (6), presburger (10), set-closed (3), set-sing (3), strand (26), strand-adv (13), toss (3), unground (8) a veanes (14). V rámci vyhodnocování jsme se rozhodli použít metriky času a počtu stavů, což je detailně popsáno v tabulkách 5.1 a 5.2, kde tabulka 5.1 ukazuje výsledky evaluace pro náš algoritmus a tabulka 5.2 prezentuje ekvivalentní vyhodnocení z nástroje Mona. Časové výsledky vizualizované bodovým diagramem jsou na obrázku 5.1. Abychom získali časové a stavové porovnání na stejném počtu formulí, tak tato sada obsahuje pouze formule vypočetlé oběma algoritmy bez dovršení časového stropu. Formule, které byly terminovány jsou z 5-ti kategorií, jmenovitě horn-sub,

	$i_\wedge$	$\Sigma_\wedge$	$\Sigma_\wedge/i_\wedge$	$\max_\wedge$	$i_\exists$	$\Sigma_\exists$	$\Sigma_\exists/i_\exists$	$\max_\exists$	$t[ms]$
basic	837	7557	9,029	1571	629	4646	7,386	1065	2874
horn-sub	105	1932	18,4	773	111	867	7,811	450	10280
horn-trans	6147	43001	6,995	60	4635	24177	5,216	42	14750
presburger	490	5765	11,765	282	310	2626	8,471	199	2230
set-closed	93	1952	20,989	245	72	854	11,861	176	958
set-sing	54	1043	19,315	170	45	480	10,667	143	1753
strand	1340	19454	14,518	1006	976	9118	9,342	546	4395
strand-adv	1579	87421	55,365	15029	1140	21739	19,069	6833	9257
toss	39	1473	37,769	506	33	348	10,545	153	990
unground	80	921	11,512	178	40	414	10,35	118	236
veanes	178	2495	14,017	365	123	1159	9,423	245	593
Σ	10942	173014	219,674	20185	8114	66428	110,141	9970	48316

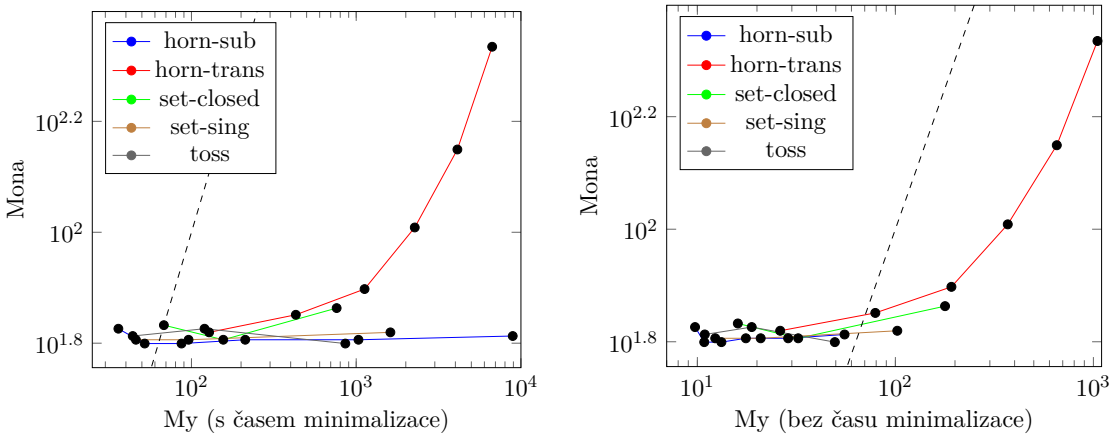
Tabulka 5.1: Výsledky experimentů z našeho algoritmu. Řádky odpovídají sumě meřené vlastnosti, definované sloupcem (např. $i_\wedge$), jednotlivých formulí z kategorie označené daným řádkem (např. basic). Poslední řádek (značen Σ) představuje součet daného sloupce. Symboly ve sloupcích znamenají následující, i_α označuje počet volání operace α, Σ_α značí sumu počtu stavů automatu po provedení operace α, Σ_α/i_α označuje podíl přechozích dvou hodnot, tedy průměrný počet stavů automatu po dokončení operace α a $\max_\alpha$ značí sumu maximálních velikostí automatů v jednotlivých formulích z příslušné kategorie po provedení operace α, kde α je buď operace průniku ($\wedge$) nebo projekce ($\exists$). Poslední sloupec ($t[ms]$) představuje dobu trvání výpočtu všech formulí z dané kategorie v milisekundách.

	$i_\wedge$	$\Sigma_\wedge$	$\Sigma_\wedge/i_\wedge$	$\max_\wedge$	$i_\exists$	$\Sigma_\exists$	$\Sigma_\exists/i_\exists$	$\max_\exists$	$t[ms]$
basic	466	6895	14,796	2012	263	1369	5,205	839	5653
horn-sub	21	2043	97,286	1143	33	753	22,818	699	336
horn-trans	855	2996	3,504	70	39	78	2	12	616
presburger	210	3224	15,352	232	102	1500	14,706	230	625
set-closed	27	874	32,37	270	15	393	26,2	203	206
set-sing	21	531	25,286	200	12	302	25,167	188	174
strand	608	25736	42,329	3476	194	4424	22,804	1536	1514
strand-adv	652	338958	519,874	19197	149	35112	235,651	17677	919
toss	12	592	49,333	395	12	188	15,667	109	170
unground	48	862	17,958	212	6	76	12,667	64	441
veanes	69	2507	36,333	1304	55	997	18,127	609	771
Σ	2989	385218	854,421	28511	880	45192	401,012	22166	11425

Tabulka 5.2: Výsledky experimentů z nástroje Mona. Značení ve sloupcích znamená následující, i_α označuje počet volání operace α, Σ_α značí sumu počtu stavů automatu po provedení operace α, Σ_α/i_α označuje podíl přechozích dvou hodnot, tedy průměrný počet stavů automatu po dokončení operace α a $\max_\alpha$ značí sumu maximálních velikostí automatů v jednotlivých formulích z příslušné kategorie po provedení operace α, kde α je buď operace průniku ($\wedge$) nebo projekce ($\exists$). Poslední sloupec ($t[ms]$) představuje dobu trvání výpočtu všech formulí z dané kategorie v milisekundách. Řádky odpovídají sumě meřené vlastnosti, definované sloupcem (např. $i_\wedge$), jednotlivých formulí z kategorie označené daným řádkem (např. basic). Poslední řádek (značen Σ) představuje součet daného sloupce.



Obrázek 5.1: Porovnání naší rozhodovací procedůry a nástroje Mona v bodovém grafu s logaritmickým měřítkem, kde na osách je čas v milisekundách a každý bod grafu odpovídá jedné formuli z testovací sady. Sytost barvy bodu značí koncentraci bodů v daném místě.



Obrázek 5.2: Srovnání naší rozhodovací procedůry a nástroje Mona v bodovém grafu s logaritmickým měřítkem pro parametrické formule, kde na osách je čas v milisekundách. Nepokračující čára značí, že naše rozhodovací procedůra byla terminována z důvodu dosažení časového stropu.

horn-trans, set-closed, set-sing a toss, jsou parametrické a pro účely následující evaluace jsme je přidali do těchto kategorií a jejich výpočet graficky vizualizujeme na obrázku 5.2. Vidíme, že náš algoritmus byl poměrně brzy vynuceně ukončen, nicméně před terminací, v grafu bez našeho času minimalizace, jsme dokázali mít lepší čas ve 2/5 kategorií.

Časové porovnání. Dle časových kriterií jednoznačně vítězí nástroj Mona, a to jak v základní sadě 196 formulí, tak i u doplněných parametrizovaných. Vidíme, že operace průniku i projekce jsou v naší rozhodovací procedůře volány výrazně vícekrát, což je způsobeno odlišnou tvorbou atomických formulí (např. $x = y$), kde naše rozhodovací procedůra volá obě tyto operace vícekrát. V našem algoritmu je největším problémem Brzozowski algortimus pro minimalizaci automatů, který při těžkých formulích dosahuje až 99% podílu na celém výpočtu. Podíl minimalizace na výpočetním čase pro všechny kategorie formulí vizualizuje obrázek 5.1 a tabulka 5.3, která potvrzuje neefektivitu minimalizačního algoritmu a naznačuje, že výměna algoritmu pro minimalizaci automatů by nás mohla přiblížit k výsledkům nástroje Mona nebo je také možné, že by se ukázala BDD minimalizace a následná jednodušší automatová minimalizace jako fundamentálně rychlejší.

$$\begin{aligned}\exists X : \forall X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9 : &(X_1 \subseteq X \Rightarrow X_2 \subseteq X) \wedge (X_2 \subseteq X \Rightarrow X_3 \subseteq X) \wedge \\ &(X_3 \subseteq X \Rightarrow X_4 \subseteq X) \wedge (X_4 \subseteq X \Rightarrow X_5 \subseteq X) \wedge \\ &(X_5 \subseteq X \Rightarrow X_6 \subseteq X) \wedge (X_6 \subseteq X \Rightarrow X_7 \subseteq X) \wedge \\ &(X_7 \subseteq X \Rightarrow X_8 \subseteq X) \wedge (X_8 \subseteq X \Rightarrow X_9 \subseteq X)\end{aligned} \tag{5.1}$$

Stavové porovnání. Naopak co se týče počtu vygenerovaných stavů, tak je na tom náš algoritmus lépe. Slovem stavy u našeho algoritmu myslíme klasické stavy automatu, ale u nástroje Mona tuto hodnotu chápeme jako součet stavů automatu a BDD uzlů. Mona vytváří vždy úplné (kde ona úplnost nějaké stavy stojí) minimalizované automaty, což na první pohled implikuje, že nelze vytvořit automaty s méně stavy. Důvod proč to lze leží v reprezentaci automatů, což se pokusíme vysvětlit v odstavci níže.

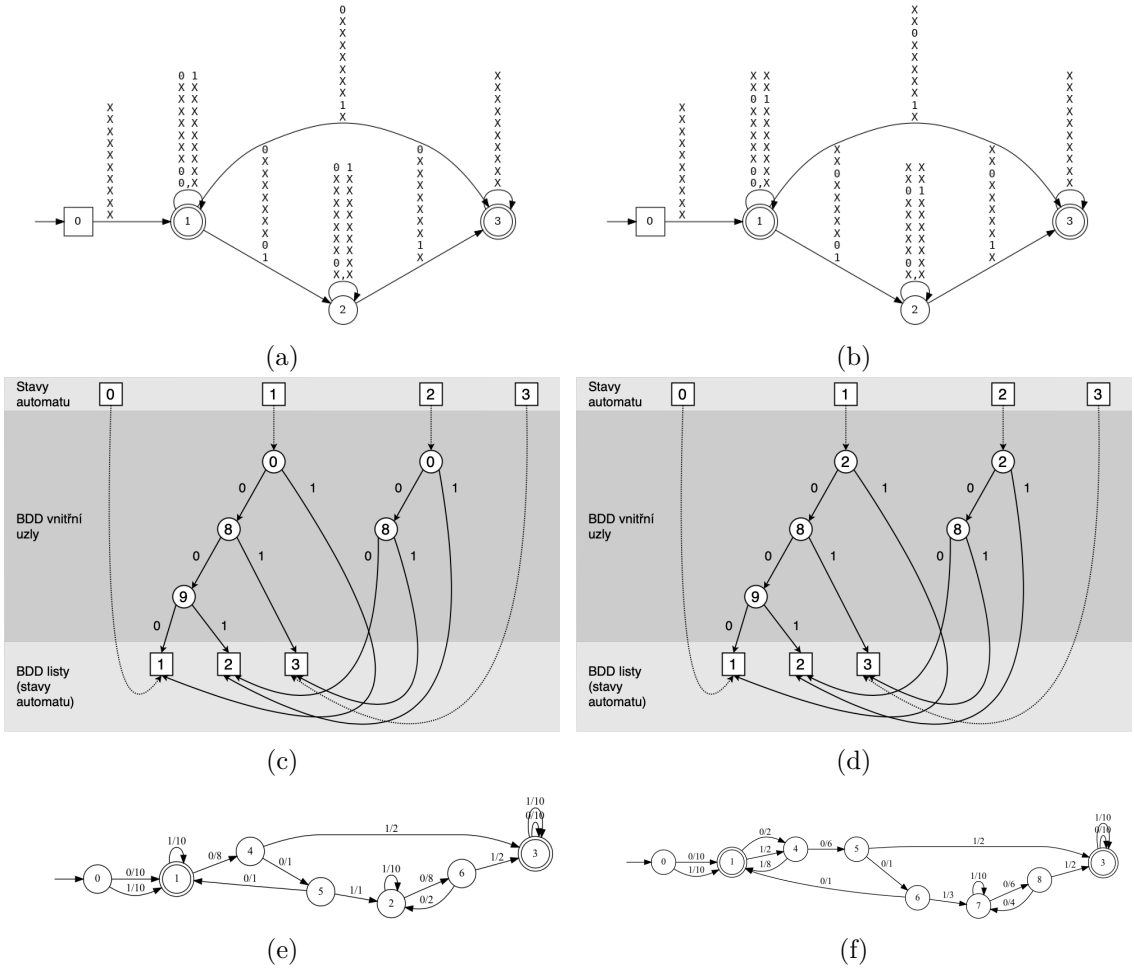
	$t[ms]$	$t - t_{min}[ms]$
basic	2874	1343
horn-sub	10280	136
horn-trans	14750	2353
presburger	2230	348
set-closed	958	227
set-sing	1753	133
strand	4395	928
strand-adv	9257	3022
toss	990	78
unground	236	95
veanes	593	191
Σ	48316	8854

Tabulka 5.3: Vizualizace neefektivity minimalizačního algoritmu. Druhý sloupec značí čas výpočtu příslušné třídy formulí (stejná hodnota jako v tabulce 5.1) a v třetím sloupci je čas z předchozího sloupce snížený o dobu běhu minimalizačního algoritmu. Nabízí se ještě porovnat dobu minimalizace pro nástroj Mona, ale jeho vypsané statistiky hovoří o tom, že všechny operace (minimalizace, průnik, projekce, komplement) u všech formulí trvají 0 s.

	My	Mona	Mona + NP	My + NP	My + NP + BP
průnik	16	27	27	18	21
průnik	37	60	60	40	48
průnik	85	132	132	89	108
průnik	193	288	288	198	240
průnik	433	624	624	439	528
průnik	961	1344	1344	968	1152
průnik	2113	2880	2880	2121	2496
projekce	1280	1792	1792	1535	1916
projekce	2	2	2	2	4
projekce	2	2	2	2	4
projekce	2	2	2	2	4
projekce	2	2	2	2	4
projekce	2	2	2	2	4
projekce	2	2	2	2	4
projekce	2	2	2	2	4
projekce	2	2	2	2	3
projekce	0	2	2	0	0
Σ	5134	7165	7165	5424	6540

Tabulka 5.4: Detailní rozbor formule 5.1 po každé operaci průniku a projekce (pořadí operací je směrem odzadu dopředu vzhledem k jejich zápisu). Sloupec My značí počet stavů při naší rozhodovací procedůře pro formuli 5.1 a sloupec Mona to stejné pro nástroj Mona. NP představuje formuli 5.1 s přidáním nevyužité proměnné s indexem 0 a BP značí zákaz přeskakování stavů úrovně 0.

Ukázka počtu generovaných stavů na konkrétní formuli. Situaci budeme ilustrovat na příkladu formule 5.1. Nechť pořadí proměnných je následující: $X, X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9$ a každé proměnné přisoudíme jeden index 0 až 9 dle uvedeného pořadí. Nejprve si rozeberme první podformuli $X_8 \subseteq X \Rightarrow X_9 \subseteq X$, jejíž automatová reprezentace se skip hranami je na obrázku 5.3e. Ekvivaletní automat v Mona reprezentaci můžeme vidět na obrázku 5.3a v podobě běžného automatu odstíněného od interní reprezentace a na obrázku 5.3c, který odpovídá tomu, jak přesně Mona automaty reprezentuje. Lze spatřit, že naše reprezentace obsahuje v tomto případě 8 stavů, zatímco Mona stavů 12, což je kvůli tomu, že Mona neumožňuje, aby samotný automatový stav mohl být použit k přiřazení do proměnné, ale používá automatové stavy jen k odkazům na BDD uzly. To může být výhodné, pokud se při přechodu ze stavu s_a do s_f jako první přiřazuje například do proměnné



Obrázek 5.3: Porovnání naší a Mona reprezentace automatu. Obrázky 5.3a, 5.3c, 5.3e vizualizují automat pro formuli $X_8 \subseteq X \Rightarrow X_9 \subseteq X$ s pořadím proměnných $X, X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9$. Obrázky 5.3b, 5.3d, 5.3f poté ukazují automat pro stejnou formuli, ale s pořadím proměnných $X_1, X_2, X, X_3, X_4, X_5, X_6, X_7, X_8, X_9$. Dále obrázky 5.3a, 5.3b vizualizují Mona reprezentaci převedenou do běžného automatu, obrázky 5.3c, 5.3d ukazují reálnou podobu Mona reprezentace a obrázky 5.3e, 5.3f vizualizují naši reprezentaci.

s indexem 3, pak stačí z s_a vytvořit odkaz na BDD uzel odpovídající proměnné 3 a velikost reprezentace se nezmění. Abychom nastínili tuto situaci, změníme pořadí proměných na $X_1, X_2, X, X_3, X_4, X_5, X_6, X_7, X_8, X_9$ a tím pádem i jejich indexy. U Mona reprezentace se změnily pouze indexy BDD uzlů, viz obrázky 5.3b a 5.3d. V naší reprezentaci, vizualizované obrázkem 5.3f, vznikl navíc jeden stav, ovšem pokud by stav 1 nebyl koncový, pak by zanikl a počet stavů automatu by zůstal také totožný. Všimněme si, že stav 2 zanikl, protože nebyl ani koncový, ani z něj nevycházely hrany se symboly $0, 1$ do různých stavů. Pokud budeme procházet formuli 5.1 dále a přitom si poznamenávat počty stavů po každé operaci průniku a projekce, dostaneme tabulku 5.4, kde vidíme srovnání počtu stavů naší rozhodovací procedůry a nástroje Mona s i bez přidání nevyužité proměnné s indexem 0, což znamená, že jsme zvětšili o 1 index všech proměnných a přidali novou na index 0, kde nová proměnná se nevyskytuje nikde ve formuli, tedy v každém okamžiku má přiřazení $0, 1$. Tato proměnná byla přidána, protože v původní formuli 5.1 se v každém kroce přiřazuje do proměnné s indexem 0, tedy nikdy se nepřeskočí stav úrovně 0 a nemohli bychom demonstrovat přínos čistě tohoto vylepšení. Ze zmíněné tabulky 5.4 plyne, že integrace BDD uzlů mezi běžné stavy automatu a také možnost přeskočit stavy úrovně 0, mají své opodstatnění a představují přínos.

Zhodnocení experimentů. Experimenty prokázaly, že po stránce počtu generovaných stavů jsme schopni dosáhnout zajímavých výsledků, ovšem doba běhu našeho algoritmu, zejména na formulích vyžadujících hodně minimalizace, je nevyhovující. Jak již bylo zmíněno, řešením by mohlo být vyměnit Brzozowski algoritmus za například nějaký simulační algoritmus, který by byl výrazně složitější na implementaci, ale mohl by přinést dramatické zrychlení.

Kapitola 6

Závěr

V této práci jsme popsali teoretické základy rozhodování WS1S, představili vlastní reprezentaci pro rozhodování WS1S, přizpůsobili a optimalizovali pro ni vybrané automatové algoritmy a implementovali ji. Následně jsme s naší implementací provedli experimenty, na základě kterých jsme se naše algoritmy snažili vylepšit a na závěr jsme se porovnali s nástrojem Mona.

Experimenty jsme provedli na sadě obsahující 196 formulí rozřazených do 11 kategorií. Ukázalo se, že námi použitý Brzozowski minimalizační algoritmus často stavově exploduje, což se podepisuje na časových výsledcích evaluace. Nicméně to otevírá další možnosti zrychlení rozhodovací procedůry čistě tím, že vyměníme minimalizaci za nějakou s nižší časovou složitostí. Změřili jsme i čas běhu představeného algoritmu na testovací sadě formulí, které zvládáme rozhodnout a při odečtení doby běhu naší minimalizace dostáváme, na naší testovací sadě, podobné výsledky jako Mona, tudíž po výměně minimalizačního algoritmu bychom se mohli ještě více přiblížit. Dále z provedených experimentů plyne, že navrhovaná reprezentace je schopna tvořit automaty s méně stavy než Mona.

V budoucích plánech je implementovat nový minimalizační algoritmus a dále pracovat na dílčích optimalizacích stávajících algoritmů, jako je přidávání globálních proměnných do automatů podle jejich výskytu či lepší práce s prvořádovými proměnnými. Dále jsme se zamýšleli i nad jinými možnostmi reprezentace, a to bez využití BDD kódování přechodů. Například, že bychom měli na hranách přechodové symboly $0, 1$ a navíc přechodové značky, které by obsahovaly proměnné do kterých se má přiřadit daný přechodový symbol. Ušetřili bychom stavy, jenž se musí vygenerovat namísto BDD uzlů, ale zároveň získáme okamžitě ekvivalentní informaci o tom do které proměnné je přiřazena 0 a 1. Takové automaty by byly v podstatě nedeterministické, čímž můžeme získat další úsporu. Naznačíme fungování průniku, kde bychom navíc průnikovali značky dvou hran jdoucích z dvojice stavů přes stejný symbol a determinizace, kde naopak hranám, se stejnými symboly jdoucích z množiny stavů, sjednotíme značky.

Literatura

[1] Basin, D. a Klarlund, N. Automata based symbolic reasoning in hardware verification. *Formal Methods In System Design*. Boston, USA: Kluwer Academic Publishers, 1992-. 1998, sv. 13, č. 3, s. 253–286.

[2] Baukus, K., Bensalem, S., Lakhnech, Y. a Stahl, K. Abstracting WS1S Systems to Verify Parameterized Networks. In: Graf, S. a Schwartzbach, M., ed. *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, s. 188–203. ISBN 978-3-540-46419-8.

[3] Büchi, J. R. Weak Second-Order Arithmetic and Finite Automata. *Mathematical Logic Quarterly*. 1960, sv. 6, 1-6, s. 66–92. DOI: https://doi.org/10.1002/malq.19600060105. Dostupné z: https://onlinelibrary.wiley.com/doi/abs/10.1002/malq.19600060105.

[4] D'Antoni, L. a Veanes, M. Minimization of Symbolic Automata. *SIGPLAN Not.* New York, NY, USA: Association for Computing Machinery. jan 2014, sv. 49, č. 1, s. 541–553. DOI: 10.1145/2578855.2535849. ISSN 0362-1340. Dostupné z: https://doi.org/10.1145/2578855.2535849.

[5] Elgaard, J., Klarlund, N. a Møller, A. MONA 1.x: new techniques for WS1S and WS2S. In: *Proc. 10th International Conference on Computer-Aided Verification, CAV '98*. Springer-Verlag, June/July 1998, sv. 1427, s. 516–520. LNCS.

[6] Fiedor, T., Holík, L., Janků, P., Lengál, O. a Vojnar, T. Lazy Automata Techniques for WS1S. In: Legay, A. a Margaria, T., ed. *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, s. 407–425.

[7] Havlena, V., Holík, L., Lengál, O., Valeš, O. a Vojnar, T. Antiprenexing for WSkS: A Little Goes a Long Way. In: Albert, E. a Kovacs, L., ed. *LPAR23. LPAR-23: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning*. EasyChair, 2020, sv. 73, s. 298–316. EPiC Series in Computing. DOI: 10.29007/6bfc. ISSN 2398-7340. Dostupné z: https://easychair.org/publications/paper/4JDl.

[8] Klarlund, N. A theory of restrictions for logics and automata. In: *Computer Aided Verification, CAV '99*. Sv. 1633. LNCS.

[9] Klarlund, N. Mona & Fido: The Logic-Automaton Connection in Practice. In: *Computer Science Logic, CSL '97*. 1998. LNCS. LNCS 1414.

[10] Klarlund, N. a Møller, A. *MONA Version 1.4 User Manual*. BRICS, Department of Computer Science, University of Aarhus, January 2001. Notes Series NS-01-1. Available from `http://www.brics.dk/mona/`.

[11] Klarlund, N., Møller, A. a Schwartzbach, M. I. MONA Implementation Secrets. *International Journal of Foundations of Computer Science*. 2002, sv. 13, č. 4, s. 571–586. World Scientific Publishing Company. Earlier version in Proc. 5th International Conference on Implementation and Application of Automata, CIAA '00, Springer-Verlag LNCS vol. 2088.

[12] Klarlund, N., Nielsen, M. a Sunesen, K. A case study in verification based on trace abstractions. In: Broy, M., Merz, S. a Spies, K., ed. *Formal Systems Specification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, s. 341–373. ISBN 978-3-540-49573-4.

[13] Madhusudan, P., Parlato, G. a Qiu, X. Decidable Logics Combining Heap Structures and Data. *SIGPLAN Not.* New York, NY, USA: Association for Computing Machinery. jan 2011, sv. 46, č. 1, s. 611–622. DOI: 10.1145/1925844.1926455. ISSN 0362-1340. Dostupné z: https://doi.org/10.1145/1925844.1926455.

[14] Meduna, A. *Automata and languages: theory and applications*. Springer, 2012. ISBN 9781852330743.

[15] Møller, A. a Schwartzbach, M. I. The Pointer Assertion Logic Engine. In: *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 2001, s. 221–231. PLDI '01. DOI: 10.1145/378795.378851. ISBN 1581134142. Dostupné z: https://doi.org/10.1145/378795.378851.

[16] Sandholm, A. a Schwartzbach, M. I. Distributed safety controllers for web services. In: Astesiano, E., ed. *Fundamental Approaches to Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, s. 270–284. ISBN 978-3-540-69723-7.

[17] Tateishi, T., Pistoia, M. a Tripp, O. Path- and Index-Sensitive String Analysis Based on Monadic Second-Order Logic. *ACM Trans. Softw. Eng. Methodol.* New York, NY, USA: Association for Computing Machinery. oct 2013, sv. 22, č. 4. DOI: 10.1145/2522920.2522926. ISSN 1049-331X. Dostupné z: https://doi.org/10.1145/2522920.2522926.

[18] Topnik, C., Wilhelm, E., Margaria, T. a Steffen, B. JMosel: A Stand-Alone Tool and jABC Plugin for M2L(Str). In: Valmari, A., ed. *Model Checking Software*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, s. 293–298. ISBN 978-3-540-33103-2.

[19] Traytel, D. A Coalgebraic Decision Procedure for WS1S. In: Kreutzer, S., ed. *24th EACSL Annual Conference on Computer Science Logic (CSL 2015)*. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, sv. 41, s. 487–503. Leibniz International Proceedings in Informatics (LIPIcs). DOI: 10.4230/LIPIcs.CSL.2015.487. ISBN 978-3-939897-90-3. Dostupné z: http://drops.dagstuhl.de/opus/volltexte/2015/5433.