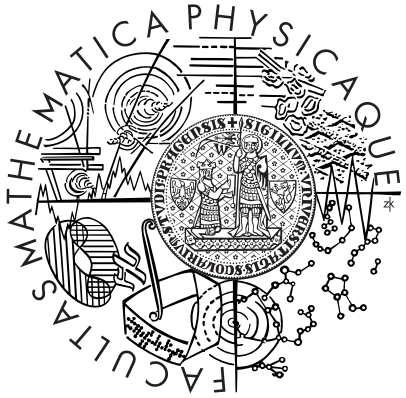
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Radek Miček

Automatická konstrukce modelů

Katedra algebry

Vedoucí diplomové práce: doc. RNDr. David Stanovský, Ph.D.

Studijní program: Informatika

Studijní obor: Teoretická informatika

Praha, 2015

Děkuji panu doc. RNDr. Davidu Stanovskému, Ph.D., vedoucímu mé diplomové práce, za téma i nápady, jenž mi věnoval.

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ..... Podpis autora

Název práce: Automatická konstrukce modelů
Autor: Radek Miček
Katedra: Katedra algebry
Vedoucí diplomové práce: doc. RNDr. David Stanovský, Ph.D., Katedra algebry

Abstrakt: V této práci implementujeme metodu MACE pro hledání konečných modelů v klasické logice prvního řádu bez sort. Kromě známých modifikací metody MACE jsou popsány a implementovány i úplně nové modifikace, například: odzplošťování, generování redundantních klauzulí a kódování do podmínek řešiče Gecode. Naše implementace je porovnána s programy Mace4, Paradox a iProver. V závěru práce jsou dány náměty na vylepšení a další výzkum.

Klíčová slova: konečné modely, propagace omezujících podmínek, SAT, redukce symetrií

Title: Automated model building
Author: Radek Miček
Department: Department of Algebra
Supervisor: doc. RNDr. David Stanovský, Ph.D., Department of Algebra

Abstract: We implement a MACE-style method for finding finite models in unsorted classical first-order logic. Additionally to well-known modifications of the method we also describe and implement several new modifications such as: unflattening, generation of redundant clauses and encoding into Gecode constraints. Our implementation is benchmarked together with Mace4, Paradox and iProver. Finally, some suggestions for improvements and further research are given.

Keywords: finite models, constraint propagation, SAT, symmetry reduction

Obsah

Značení

A	atom
C	klauzule
c, f, g	funkční symboly (c může značit i buňku)
$\mathcal{F}$	množina funkčních symbolů
$\mathcal{I}$	interpretace
L	literál
N	množina klauzulí
$\mathbb{N}$	množina přirozených čísel, tj. $\{1, 2, 3, \ldots\}$
$\mathbb{N}_0$	množina nezáporných celých čísel, tj. $\{0, 1, 2, \ldots\}$
P	predikátový symbol
$\mathcal{P}$	množina predikátových symbolů
S	sorta
$\mathcal{S}$	množina sort
s, t	termy
x, y, z	proměnné
$\mathcal{X}$	množina proměnných

KAPITOLA 1

Úvod

Cílem této práce je implementovat program pro hledání konečných modelů v klasické logice prvního řádu s rovností. Bohatost logiky prvního řádu hraje důležitou roli při modelování praktických problémů. Díky ní se hledače modelů mohou uplatnit nejen při výzkumu matematických struktur, ale i při verifikaci softwaru a hardwaru, jako součást běhových prostředí deklarativních programovacích jazyků anebo v jiných oblastech, kde se dnes používají SAT řešiče a SMT řešiče.

Existuje celá řada metod pro hledání modelů, tato práce se však zabývá pouze metodou MACE a jejími modifikacemi. Metoda MACE převádí problém hledání konečného modelu s doménou velikosti n v klasické logice prvního řádu na problém SAT. Většina modifikací metody MACE se snaží o to, aby byl výsledný SAT vyřešitelný rychle a s malým množstvím paměti.

Náš program implementuje metodu MACE, její modifikace známé z programu Paradox a některé nové modifikace, které, pokud je nám známo, zatím ještě nikdo nezkoušel. Například kromě převodu na SAT je implementován převod do jazyka řešiče omezujících podmínek Gecode nebo přidávání redundantních formulí ke vstupu.

Práce je rozdělena do šesti kapitol. Tato kapitola stručně představuje téma práce. Druhá kapitola obsahuje definice základních pojmů, jenž budeme používat v dalších kapitolách, a přesnější popis řešeného problému.

Třetí kapitola popisuje existující metody pro hledání modelů. Důraz je kladen na metody MACE a SEM pro hledání konečných modelů.

Čtvrtá kapitola začíná popisem našich vlastních vylepšení metody MACE, která jsou v programu implementována. Následuje velmi stručný popis samotné implementace. Pátá kapitola obsahuje experimentální srovnání našeho hledače konečných modelů s jinými programy. Šestá kapitola shrnuje dosažené výsledky a popisuje další vylepšení, která však nejsou implementována v našem programu.

KAPITOLA 2

Základní definice

Hlavním účelem této kapitoly je sjednotit základní definice a značení. Kromě toho si v této kapitole popíšeme, co přesně má náš program dělat – co přesně je vstupem a výstupem programu. Předpokládáme, že čtenář již zná základy klasické logiky prvního řádu a základní techniky pro řešení omezujících podmínek. Chybějící znalosti logiky lze doplnit z [12] a omezujících podmínek z [10].

Budeme pracovat ve vícesortové klasické logice prvního řádu. V celém textu budeme používat pevně danou signaturu, která se skládá z nekonečné množiny sort $\mathcal{S}$, množiny funkčních symbolů $\mathcal{F}$, množiny predikátových symbolů $\mathcal{P}$ a funkce arita : $\mathcal{F} \cup \mathcal{P} \to \mathcal{S}^\star$, kde $\mathcal{S}^\star$ značí množinu konečných posloupností sort. Funkce arita přiřazuje každému funkčnímu symbolu neprázdnou posloupnost sort $\langle S_1, \ldots, S_n, S\rangle$, kde $n \geq 0$ je počet argumentů, $S_1, \ldots, S_n$ jsou sorty argumentů a S je sorta výsledku. Funkce arita přiřazuje každému predikátovému symbolu posloupnost sort $\langle S_1, \ldots, S_n\rangle$, kde $n \geq 0$ je počet argumentů a $S_1, \ldots, S_n$ jsou sorty argumentů.

Pro každou neprázdnou posloupnost sort a obsahuje $\mathcal{F}$ nekonečně mnoho symbolů s aritou a. Pro každou posloupnost sort a obsahuje $\mathcal{P}$ nekonečně mnoho symbolů s aritou a.

Dále je každé sortě S přiřazena nekonečná množina proměnných $\mathcal{X}_S$ a tyto množiny jsou navzájem disjunktní. $\mathcal{X}$ je množina všech proměnných, tj. $\bigcup_{S\in\mathcal{S}} \mathcal{X}_S$. Množiny $\mathbb{N}_0$, $\mathcal{S}$, $\mathcal{F}$, $\mathcal{P}$, $\mathcal{X}$ jsou navzájem disjunktní.

Termy sorty S jsou definovány induktivně:

- proměnná sorty S je term sorty S,
- je-li f funkční symbol s aritou $\langle S_1, \ldots, S_n, S\rangle$ a t_i term sorty S_i pro každé $i \in \{1, \ldots, n\}$, pak $f(t_1, \ldots, t_n)$ je term sorty S.

Atomy jsou definovány induktivně:

- jsou-li s a t termy stejné sorty, pak $s \approx t$ je atom,
- je-li P predikátový symbol s aritou $\langle S_1, \ldots, S_n\rangle$ a t_i term sorty S_i pro každé $i \in \{1, \ldots, n\}$, pak $P(t_1, \ldots, t_n)$ je atom.

Literál je atom nebo jeho negace. Klauzule je disjunkce literálů. Řekneme, že sorta S se vyskytuje v termu, atomu, literálu, klauzuli, množině klauzulí, pokud

se tam vyskytuje proměnná sorty S nebo funkční či predikátový symbol, kde S je sorta argumentu nebo výsledku.

Sorty budeme značit písmenem S, funkční symboly písmeny c, f a g, predikátové symboly písmenem P a proměnné písmeny x, y a z. Termy budeme značit písmeny s a t, atomy písmenem A, literály písmenem L, klauzule písmenem C a množiny klauzulí písmenem N.

Nyní zadefinujeme interpretaci. Mějme množiny $\mathcal{S}' \subseteq \mathcal{S}$, $\mathcal{F}' \subseteq \mathcal{F}$ a $\mathcal{P}' \subseteq \mathcal{P}$, kde $\mathcal{S}'$ obsahuje alespoň sorty vyskytující se v aritách symbolů z $\mathcal{F}'$ a $\mathcal{P}'$. Funkce $\mathcal{I}$ definovaná na množině $\mathcal{S}' \cup \mathcal{F}' \cup \mathcal{P}'$ je interpretace, jestliže

- každé sortě z $\mathcal{S}'$ přiřadí neprázdnou množinu (doménu),
- každému funkčnímu symbolu $f \in \mathcal{F}'$ s aritou $\langle S_1, \ldots, S_n, S\rangle$ přiřadí funkci $f^{\mathcal{I}} : \mathcal{I}(S_1) \times \cdots \times \mathcal{I}(S_n) \to \mathcal{I}(S)$
- a každému predikátovému symbolu $P \in \mathcal{P}'$ s aritou $\langle S_1, \ldots, S_n\rangle$ přiřadí relaci $P^{\mathcal{I}} \subseteq \mathcal{I}(S_1) \times \cdots \times \mathcal{I}(S_n)$.

Interpretace je konečná, jsou-li $\mathcal{S}'$, $\mathcal{F}'$ a $\mathcal{P}'$ konečné a je-li každá doména konečná. Konečná interpretace je číselná, pokud je každá doména počáteční úsek $\mathbb{N}_0$, jinak řečeno, pokud je každá doména rovna $\{0, \ldots, n-1\}$, kde n je velikost domény. Pokud není řečeno jinak, bude $\mathcal{I}$ v dalším textu označovat interpretaci definovanou na množině $\mathcal{S}' \cup \mathcal{F}' \cup \mathcal{P}'$.

Buď $\mathcal{I}$ interpretace a β funkce definovaná na množině $\bigcup_{S\in\mathcal{S}'} \mathcal{X}_S$. Funkci β nazveme ohodnocením proměnných, pokud každé proměnné $x \in \mathcal{X}_S$ přiřazuje prvek domény S.

Je-li R term nebo atom nebo literál nebo klauzule, jenž obsahuje pouze sorty z $\mathcal{S}'$, funkční symboly z $\mathcal{F}'$ a predikátové symboly z $\mathcal{P}'$, pak hodnotu R při interpretaci $\mathcal{I}$ a ohodnocení proměnných β značíme $\mathcal{I}_\beta(R)$. Pro term t je $\mathcal{I}_\beta(t)$ definováno následovně:

$$\mathcal{I}_\beta(x) = \beta(x),$$
$$\mathcal{I}_\beta\Big(f(t_1, \ldots, t_n)\Big) = \mathcal{I}\Big(f\Big)\Big(\mathcal{I}_\beta(t_1), \ldots, \mathcal{I}_\beta(t_n)\Big).$$

Pro literál L je $\mathcal{I}_\beta(L)$ definováno takto:

$$\mathcal{I}_\beta(s \approx t) = \begin{cases} 1 & \text{pokud } \mathcal{I}_\beta(s) = \mathcal{I}_\beta(t), \\ 0 & \text{jinak,} \end{cases}$$

$$\mathcal{I}_\beta\Big(P(t_1, \ldots, t_n)\Big) = \begin{cases} 1 & \text{pokud } \Big(\mathcal{I}_\beta(t_1), \ldots, \mathcal{I}_\beta(t_n)\Big) \in \mathcal{I}(P), \\ 0 & \text{jinak,} \end{cases}$$

$$\mathcal{I}_\beta(\neg A) = 1 - \mathcal{I}_\beta(A).$$

A nakonec definice $\mathcal{I}_\beta(C)$ pro klauzuli C:

$$\mathcal{I}_\beta(L_1 \vee \cdots \vee L_n) = \max\Big\{0, \mathcal{I}_\beta(L_1), \ldots, \mathcal{I}_\beta(L_n)\Big\}.$$

Skutečnost, že $\mathcal{I}_\beta(R) = 1$, kde R je atom, literál nebo klauzule, zapisujeme

$$\mathcal{I}_\beta \models R$$

a říkáme, že R je splněné v interpretaci $\mathcal{I}$ při ohodnocení proměnných β. Platí-li $\mathcal{I}_\beta \models R$ pro libovolné β, říkáme, že $\mathcal{I}$ je model R, a píšeme

$$\mathcal{I} \models R.$$

Model $\mathcal{I}$ je konečný resp. číselný, je-li interpretace $\mathcal{I}$ konečná resp. číselná. $\mathcal{I}$ je model množiny klauzulí N, pokud je $\mathcal{I}$ model každé klauzule z N. R je vždy splněné, pokud je každá interpretace, jenž interpretuje sorty a symboly z R, model R. Naopak R není nikdy splněné, neexistuje-li interpretace $\mathcal{I}$ a ohodnocení proměnných β, že R je splněné v $\mathcal{I}$ při β.

Interpretace $\mathcal{I}$ a $\mathcal{I}'$ nad $\mathcal{S}' \cup \mathcal{F}' \cup \mathcal{P}'$ jsou izomorfní (značíme $\mathcal{I} \simeq \mathcal{I}'$), pokud pro každou sortu $S \in \mathcal{S}'$ existuje bijekce $i_S : \mathcal{I}(S) \to \mathcal{I}'(S)$, že

- pro každý funkční symbol $f \in \mathcal{F}'$ s aritou $\langle S_1, \ldots, S_n, S\rangle$ a pro každé $d_j \in \mathcal{I}(S_j)$, kde $j \in \{1, \ldots, n\}$, platí

$$i_S\Big(\mathcal{I}(f)(d_1, \ldots, d_n)\Big) = \mathcal{I}'\Big(f\Big)\Big(i_{S_1}(d_1), \ldots, i_{S_n}(d_n)\Big)$$

- a pro každý predikátový symbol $P \in \mathcal{P}'$ s aritou $\langle S_1, \ldots, S_n\rangle$ a pro každé $d_j \in \mathcal{I}(S_j)$, kde $j \in \{1, , \ldots, , n\}$, platí

$$(d_1, \ldots, d_n) \in \mathcal{I}(P) \iff \Big(i_{S_1}(d_1), \ldots, i_{S_n}(d_n)\Big) \in \mathcal{I}'(P).$$

Připomeňme vlastnosti izomorfních interpretací:

(1) Každá konečná interpretace je izomorfní číselné interpretaci.

(2) Je-li $\mathcal{I} \simeq \mathcal{I}'$ a N množina klauzulí, pak $\mathcal{I}$ je model N právě tehdy, když $\mathcal{I}'$ je model N.

V tomto okamžiku máme potřebné definice, abychom specifikovali chování programu – jaký problém program řeší:

Definice 2.1. Program pracuje ve dvou režimech – hledání jednoho modelu a hledání všech neizomorfních modelů. V obou režimech je vstupem programu sorta S a konečná množina klauzulí N, kde se vyskytuje pouze sorta S. V obou režimech jsou výstupem konečné modely N s jedinou sortou S, které interpretují právě symboly z N.

V režimu hledání jednoho modelu jsou navíc vstupem přirozená čísla $n \leq n'$. Výstupem v tomto režimu je model s doménou velikosti m, kde $n \leq m \leq n'$, pokud existuje.

V režimu hledání všech neizomorfních modelů je navíc vstupem programu přirozené číslo n. Výstupem v tomto režimu jsou všechny navzájem neizomorfní modely s doménou velikosti n.

Z (1) a (2) plyne, že pro každý vstup programu existuje výstup obsahující pouze číselné modely. Aniž by to mělo vliv na úplnost programu, může program hledat pouze číselné modely. Navíc číselných interpretací, jenž jsou kandidáty na model, je pouze konečně mnoho (důvodem je, že číselných interpretací s jedinou sortou S obsahujících právě symboly z N a doménou velikosti n je pouze konečně mnoho), díky čemuž lze hledané modely získat metodou generuj a testuj a stačí k tomu primitivní rekurze.

Zbývající definice se nám budou hodit při detekci izomorfních modelů. Zobrazení kanonických reprezentantů ρ pro číselné interpretace je zobrazení z číselných interpretací do číselných interpretací splňující

- $\rho(\mathcal{I}) \simeq \mathcal{I}$
- a $\mathcal{I} \simeq \mathcal{I}' \iff \rho(\mathcal{I}) = \rho(\mathcal{I}')$.

Orientovaný barevný graf (dále jen graf) je trojice $G = (V, E, c)$, kde V je konečná množina vrcholů, $E \subseteq V \times V$ je množina hran a $c : V \to \mathbb{N}_0$ je obarvení vrcholů. Grafy $G = (V, E, c)$ a $G' = (V, E', c')$ jsou izomorfní (značíme $G \simeq G'$), pokud existuje bijekce $i : V \to V$, že

- $c'\Big(i(v)\Big) = c(v)$
- a $(u, v) \in E \iff \Big(i(u), i(v)\Big) \in E'$.

Zobrazení kanonických reprezentantů ρ pro grafy je zobrazení z grafů do grafů splňující

- $\rho(G) \simeq G$
- a $G \simeq G' \iff \rho(G) = \rho(G')$.

Poznámka 2.1. K definicím.

- Některé transformace problému zavádějí nové symboly – v takovém případě je možné rozšířit signaturu nebo zajistit, že stávající signatura obsahuje dostatek nepoužitých symbolů. Domníváme se, že druhá možnost, ač méně obvyklá, je přehlednější – v celém textu používáme jednu signaturu, která je zkonstruována tak, že pro každou aritu vždy existuje nepoužitý symbol. Díky tomu, že signatura je pouze jedna, není třeba ji zmiňovat.
- Proč je množina sort $\mathcal{S}'$ v interpretaci zadána explicitně, proč není implicitně určena na základě interpretovaných symbolů? Interpretovat pouze sorty obsažené ve funkčních a predikátových symbolech nestačí, je třeba interpretovat i sorty proměnných.
- Proč je na vstupu programu sorta zadána explicitně, proč není implicitně určena z klauzulí na vstupu? Vstupní množina klauzulí nemusí žádnou sortu obsahovat (prázdná množina nebo množina s prázdnou klauzulí), v takovém případě není jednoznačně určeno, jakou sortu má model interpretovat, a číselných interpretací, jenž jsou kandidáty na model, je nekonečně mnoho.
- Vstup i výstup programu jsou jednosortové, proč výklad komplikovat vícesortovou logikou? Některé problémy s jednou sortou lze přeformulovat jako rychleji vyřešitelné problémy s více sortami.

KAPITOLA 3

Známé metody hledání modelů

Jak, jsme viděli v minulé kapitole, najít model velikosti n nebo ukázat, že neexistuje, je snadné – stačí použít metodu generuj a testuj. Tuto metodu lze jednoduše implementovat, kvůli své pomalosti je však prakticky nepoužitelná. V této kapitole budeme studovat pokročilejší metody pro hledání modelů, které jsou obvykle rychlejší než metoda generuj a testuj. Začneme popisem základních variant metod MACE a SEM, poté popíšeme modifikace obou metod. Metody MACE a SEM interně pracují s „klauzulemi“ bez proměnných a interpretace reprezentují explicitně, na konci kapitoly pak zmíníme metody, které interně pracují s klauzulemi s proměnnými a interpretace reprezentují implicitně.

3.1 Metody MACE a SEM

Obě metody řeší následující úlohu: Vstupem úlohy je konečná množina klauzulí N a velikost domény n_S pro každou sortu $S \in \mathcal{S}'$, kde $\mathcal{S}'$ je množina obsahující právě sorty z N. Výstupem úlohy jsou číselné modely nad $\mathcal{S}' \cup \mathcal{F}' \cup \mathcal{P}'$, kde domény mají velikosti n_S a $\mathcal{F}'$ resp. $\mathcal{P}'$ je množina obsahující právě funkční resp. predikátové symboly z N.

Všimněme si, že zadání úlohy jednoznačně určuje domény sort (domény jsou počátečním úsekem $\mathbb{N}_0$ a známe i jejich velikosti). Díky tomu jsou hledané modely jednoznačně určeny pouze funkcemi a relacemi. Navíc víme, nad jakými množinami jsou funkce i relace definovány. Označme $D_S = \{0, \ldots, n_S - 1\}$ doménu sorty $S \in \mathcal{S}'$. Každému funkčnímu symbolu $f \in \mathcal{F}'$ arity $\langle S_1, \ldots, S_n, S\rangle$ pak model přiřazuje funkci $f^{\mathcal{I}} : D_{S_1} \times \cdots \times D_{S_n} \to D_S$ a každému predikátovému symbolu $P \in \mathcal{P}'$ arity $\langle S_1, \ldots, S_n\rangle$ je přiřazena relace $P^{\mathcal{I}} \subseteq D_{S_1} \times \cdots \times D_{S_n}$.

Tyto funkce a relace můžeme reprezentovat tabulkou. Funkce $f^{\mathcal{I}}$ přiřazená symbolu f s aritou $\langle S_1, \ldots, S_n, S\rangle$ je reprezentována tabulkou o rozměrech $n_{S_1} \times \cdots \times n_{S_n}$, kde buňky tabulky obsahují prvky z D_S.

$$\mathcal{C}_f = \Big\{ f(v_1, \ldots, v_n) \;\Big|\; v_j \in D_{S_j}, \text{ kde } j \in \{1, \ldots, n\} \Big\}$$

je množina buněk tabulky pro funkci $f^{\mathcal{I}}$. Buňka $f(v_1, \ldots, v_n)$ obsahuje hodnotu v (zapisujeme $f(v_1, \ldots, v_n) = v$) právě tehdy, když $f^{\mathcal{I}}(v_1, \ldots, v_n) = v$.

Relace $P^{\mathcal{I}}$ přiřazená predikátovému symbolu P s aritou $\langle S_1, \ldots, S_n\rangle$ je reprezentována tabulkou o rozměrech $n_{S_1} \times \cdots \times n_{S_n}$, kde buňky tabulky obsahují

prvky 0 nebo 1.

$$\mathcal{C}_P = \left\{ P(v_1, \ldots, v_n) \;\middle|\; v_j \in D_{S_j}, \text{ kde } j \in \{1, \ldots, n\} \right\}$$

je množina buněk tabulky pro relaci $P^{\mathcal{I}}$. Fakt, že buňka $P(v_1, \ldots, v_n)$ obsahuje hodnotu v, zapisujeme $P(v_1, \ldots, v_n) = v$. Buňka $P(v_1, \ldots, v_n)$ obsahuje hodnotu 1 právě tehdy, když $(v_1, \ldots, v_n) \in P^{\mathcal{I}}$.

V dalším textu budeme hledání modelů metodami MACE a SEM chápat jako vyplňování tabulek funkcí a relací.

3.1.1 MACE – základní varianta

Metoda MACE je popsána v [20]. Jak jsme již řekli v úvodu, metoda MACE převádí problém na SAT. Napřed vytvoříme instanci SATu, jejíž řešení odpovídají 1 ku 1 tabulkám funkcí[1] a relací:

1) Pro každou funkci $f^{\mathcal{I}}$, jejíž tabulku chceme vyplnit:
   a) Pro každou buňku $f(v_1, \ldots, v_n)$ a hodnotu v, kterou daná buňka může obsahovat, přidáme výrokovou proměnnou $A_{f(v_1,\ldots,v_n)=v}$.
   b) Pro každou buňku $f(v_1, \ldots, v_n)$ a dvojici hodnot $v < v'$, které buňka může obsahovat, přidáme klauzuli $\neg A_{f(v_1,\ldots,v_n)=v} \vee \neg A_{f(v_1,\ldots,v_n)=v'}$.
   c) Pro každou buňku $f(v_1, \ldots, v_n)$ a všechny hodnoty $0, \ldots, v$, jenž může obsahovat, přidáme klauzuli $A_{f(v_1,\ldots,v_n)=0} \vee \cdots \vee A_{f(v_1,\ldots,v_n)=v}$.
2) Pro každou relaci $P^{\mathcal{I}}$, jejíž tabulku chceme vyplnit: Pro každou buňku $P(v_1, \ldots, v_n)$ přidáme výrokovou proměnnou $A_{P(v_1,\ldots,v_n)=1}$.

Na základě řešení instance SATu můžeme vyplnit tabulky funkcí a relací: Výrok V o buňce tabulky bude pravdivý právě tehdy, když výroková proměnná A_V má v řešení hodnotu 1. Klauzule z kroku 1b) zajišťují, že každá buňka obsahuje nejvýše jednu hodnotu. Klauzule z kroku 1c) zajišťují, že každá buňka obsahuje alespoň jednu hodnotu.

Tabulky, které takto získáme, nejsou nutně modely N. Aby tomu tak bylo, rozšíříme instanci SATu o další výrokové klauzule, které získáme zakódováním klauzulí z N do výrokové logiky. Uvažme například klauzuli $C_1 = \{f(x, y) \approx x \vee c \approx x\}$, kde f je funkční symbol arity $\langle S_1, S_2, S_1 \rangle$, c je funkční symbol arity $\langle S_1 \rangle$ a sorta S_1 resp. S_2 má doménu velikosti 3 resp. 2. Aby tabulky symbolů f a c byly modelem klauzule C_1, musí platit následující podmínky:

$$\begin{aligned}
f(0,0) &= 0 \text{ nebo } c = 0,\\
f(1,0) &= 1 \text{ nebo } c = 1,\\
f(2,0) &= 2 \text{ nebo } c = 2,\\
f(0,1) &= 0 \text{ nebo } c = 0,\\
f(1,1) &= 1 \text{ nebo } c = 1,\\
f(2,1) &= 2 \text{ nebo } c = 2.
\end{aligned}$$

[1] Pro funkce v této práci používáme přímé kódování. Existují však i jiné možnosti. Pokud bychom například chtěli mít méně výrokových proměnných pro funkce, mohli bychom použít logaritmické kódování. Nicméně množství proměnných pro funkce nebývá problém, neboť se obvykle pracuje pouze s malými doménami. Naopak problém je horší propagace logaritmického kódování. Různá kódování včetně přímého a logaritmického jsou popsána v [13].

Uvedené podmínky se skládají z výroků, jenž přímo odpovídají výrokovým proměnným. Díky tomu můžeme podmínky snadno zakódovat do výrokových klauzulí:

$$\begin{aligned} &A_{f(0,0)=0} \vee A_{c=0}, \\ &A_{f(1,0)=1} \vee A_{c=1}, \\ &A_{f(2,0)=2} \vee A_{c=2}, \\ &A_{f(0,1)=0} \vee A_{c=0}, \\ &A_{f(1,1)=1} \vee A_{c=1}, \\ &A_{f(2,1)=2} \vee A_{c=2}. \end{aligned}$$

Klauzuli C_1 jsme tedy zakódovali do 6 výrokových klauzulí, které přidáme do naší instance SATu. Bohužel, ne každou klauzuli jde zakódovat takto přímočaře. Například, aby tabulky symbolů byly modelem klauzule $C_2 = f(c, y) \approx c$, musí platit podmínky

$$\begin{aligned} f(c, 0) &= c, \\ f(c, 1) &= c. \end{aligned}$$

Na rozdíl od podmínek pro C_1 výroky v podmínkách pro C_2 neodpovídají přímo výrokovým proměnným. Odpovídaly by, kdybychom znali přesnou hodnotu c, jenže tu neznáme. Víme však, že c má jednu z hodnot 0, 1, nebo 2, nabízí se tedy použít podmiňování. Pokud má c hodnotu 0, pak musí platit podmínky

$$\begin{aligned} f(0, 0) &= 0, \\ f(0, 1) &= 0, \end{aligned}$$

pokud má c hodnotu 1, musí platit

$$\begin{aligned} f(1, 0) &= 1, \\ f(1, 1) &= 1, \end{aligned}$$

a nakonec, pokud má c hodnotu 2, musí platit

$$\begin{aligned} f(2, 0) &= 2, \\ f(2, 1) &= 2. \end{aligned}$$

Tyto podmínky již do výrokových klauzulí zakódujeme snadno:

$$\begin{aligned} A_{c=0} &\implies A_{f(0,0)=0}, \\ A_{c=0} &\implies A_{f(0,1)=0}, \\ A_{c=1} &\implies A_{f(1,0)=1}, \\ A_{c=1} &\implies A_{f(1,1)=1}, \\ A_{c=2} &\implies A_{f(2,0)=2}, \\ A_{c=2} &\implies A_{f(2,1)=2}. \end{aligned}$$

Klauzule, které lze zakódovat do výrokových klauzulí bez podmiňování, nazveme ploché:

Definice 3.1. Klauzule je plochá, pokud má každý atom jeden z následujících tvarů

- $f(x_1, \ldots, x_n) \approx y$ (případně $y \approx f(x_1, \ldots, x_n)$),
- $P(x_1, \ldots, x_n)$,
- nebo $x \approx y$,

kde $f, P, x, x_1, \ldots, x_n, y$ jsou vhodně zvolené symboly a proměnné.

Pomocí těchto dvou pravidel můžeme zploštit každou klauzuli C:

- Je-li t term, jenž se vyskytuje v C, a x proměnná, jenž se v C nevyskytuje, pak klauzuli C můžeme nahradit novou klauzulí $t \not\approx x \vee C$.
- Je-li L literál z C tvaru $t \not\approx x$ (resp. $x \not\approx t$), pak můžeme C upravit tak, že jeden výskyt t mimo L nahradíme x.

Navíc má nová klauzule stejné modely jako původní klauzule.

Použití pravidel pro zplošťování si ukážeme na klauzuli C_2. Klauzule C_2 není plochá, neboť obsahuje atom $f(c, y) \approx c$, který nemá požadovaný tvar (atom může obsahovat nejvýše jeden výskyt funkčního nebo predikátového symbolu). Potřebujeme tedy nahradit c proměnnou. Napřed použijeme první pravidlo a ke klauzuli C_2 přidáme literál $c \not\approx x$, tj. dostaneme novou klauzuli $C_2' = c \not\approx x \vee f(c, y) \approx c$. Na klauzuli C_2' pak můžeme použít dvakrát druhé pravidlo a nahradit dva výskyty c proměnnou x, čímž dostaneme plochou klauzuli $C_2'' = c \not\approx x \vee f(x, y) \approx x$.

Kdybychom zakódovali C_2'' do výrokových klauzulí, dostali bychom stejné klauzule, jako při kódování C_2 s pomocí podmiňování. V dalším textu již budeme používat výhradně zplošťování, důvodem je, že zplošťování může plně nahradit podmiňování[2]. Později zplošťování rozšíříme o další pravidla, díky nimž v některých případech dostaneme méně výrokových klauzulí, než kdybychom použili podmiňování.

Kódujeme-li plochou klauzuli C do výrokových klauzulí, počet výrokových klauzulí závisí exponenciálně na počtu proměnných v C. Nepříjemnou vlastností zplošťování je, že jeho první pravidlo zvyšuje počet proměnných v klauzuli. Tato nepříjemnost je motivací pro metodu, jenž si nyní představíme.

3.1.2 SEM – základní varianta

Metoda SEM je popsána v [32]. Na rozdíl od metody MACE metoda SEM problém nikam nepřevádí, ale rovnou ho sama řeší. Pro popis metody SEM budeme předpokládat $v, v' \in \mathbb{N}_0$ a c je buňka.

Metoda SEM sestává ze dvou funkcí Search a Propagate. Tyto funkce reprezentují stav řešeného problému jako pětici $(A_\mathcal{F}, A_\mathcal{P}, U_\mathcal{F}, U_\mathcal{P}, E)$:

[2]Podmiňování je vlastně zplošťování přímo integrované do procesu, který transformuje klauzule z N na výrokové klauzule. My jsme tento složitější proces rozdělili na dva jednodušší procesy – prvním je zplošťování a druhým je transformace plochých klauzulí z N na výrokové klauzule. Nyní můžeme oba procesy vylepšovat zvlášť, případně vložit další procesy mezi ně, což by jinak nebylo možné.

- Množiny $A_{\mathcal{F}}$ a $A_{\mathcal{P}}$ obsahují buňky, jimž byla přiřazena hodnota. $A_{\mathcal{F}}$ je množina dvojic (c, v), kde c je buňka funkce a v je hodnota přiřazená této buňce. $A_{\mathcal{P}}$ je množina jejíž prvky mají tvar c nebo $\neg c$, kde c je buňka relace. Je-li $c \in A_{\mathcal{P}}$, pak je buňce c přiřazena hodnota 1, je-li $\neg c \in A_{\mathcal{P}}$, pak je buňce c přiřazena hodnota 0.

- Množiny $U_{\mathcal{F}}$ a $U_{\mathcal{P}}$ obsahují buňky, jimž nebyla přiřazena hodnota. $U_{\mathcal{F}}$ je množina dvojic (c, D), kde c je buňka funkce a D je množina hodnot, jenž lze přiřadit buňce. $U_{\mathcal{P}}$ je množina buněk relací.

- E je množina podmínek, které zbývá splnit. Každá podmínka je disjunkce komponent, těmto komponentám budeme říkat literály.

Invariantem je, že každá buňka má právě jeden výskyt v právě jedné z množin $A_{\mathcal{F}}, A_{\mathcal{P}}, U_{\mathcal{F}}, U_{\mathcal{P}}$. Konflikt nastává v okamžiku, kdy se v E objeví podmínka bez literálů nebo kdy se v $U_{\mathcal{F}}$ objeví dvojice tvaru $(c, \emptyset)$. Model je nalezen v okamžiku, kdy jsou všechny podmínky splněny (množina E je prázdná) a každé buňce je přiřazena hodnota (množiny $U_{\mathcal{F}}$ a $U_{\mathcal{P}}$ jsou prázdné). Model je určen množinami $A_{\mathcal{F}}$ a $A_{\mathcal{P}}$.

Funkce SEARCH je definována následovně:

function SEARCH($A_{\mathcal{F}}, A_{\mathcal{P}}, U_{\mathcal{F}}, U_{\mathcal{P}}, E$)
  $r \leftarrow$ PROPAGATE($A_{\mathcal{F}}, A_{\mathcal{P}}, U_{\mathcal{F}}, U_{\mathcal{P}}, E$)
  if $r \neq$ `conflict` **then**
    $(A_{\mathcal{F}}, A_{\mathcal{P}}, U_{\mathcal{F}}, U_{\mathcal{P}}, E) \leftarrow r$
    if $U_{\mathcal{F}} \cup U_{\mathcal{P}} = \emptyset$ **then**
      print model $(A_{\mathcal{F}}, A_{\mathcal{P}})$
    else
      $c \leftarrow$ vyber buňku z $U_{\mathcal{F}} \cup U_{\mathcal{P}}$
      if $(c, D) \in U_{\mathcal{F}}$ pro nějaké D **then**
        for $v \in D$ **do**
          SEARCH($A_{\mathcal{F}} \cup \{(c, v)\}, A_{\mathcal{P}}, U_{\mathcal{F}} \setminus \{(c, D)\}, U_{\mathcal{P}}, E$)
        end for
      else
        SEARCH($A_{\mathcal{F}}, A_{\mathcal{P}} \cup \{c\}, U_{\mathcal{F}}, U_{\mathcal{P}} \setminus \{c\}, E$)
        SEARCH($A_{\mathcal{F}}, A_{\mathcal{P}} \cup \{\neg c\}, U_{\mathcal{F}}, U_{\mathcal{P}} \setminus \{c\}, E$)
      end if
    end if
  end if
end function

Vstupem funkce SEARCH je již zmíněná pětice $(A_{\mathcal{F}}, A_{\mathcal{P}}, U_{\mathcal{F}}, U_{\mathcal{P}}, E)$. Výstupem funkce SEARCH jsou modely, pro výpis modelů se používá příkaz **print model**.

Na začátku metoda SEM zavolá funkci SEARCH s argumenty

$$
\begin{aligned}
A_{\mathcal{F}} &= \emptyset, \\
A_{\mathcal{P}} &= \emptyset, \\
U_{\mathcal{F}} &= \Big\{(c, D_S) \;\Big|\; f \in \mathcal{F}', S \text{ je sorta výsledku } f, c \in \mathcal{C}_f\Big\}, \\
U_{\mathcal{P}} &= \Big\{c \;\Big|\; P \in \mathcal{P}', c \in \mathcal{C}_P\Big\}, \\
E &= \left\{e \;\middle|\; \begin{array}{c} e \text{ vznikne z } C \in N \text{ použitím substituce, která každou} \\ \text{proměnnou } x \text{ z } C \text{ nahradí hodnotou z } D_S, \text{ kde } S \text{ je sorta } x \end{array}\right\}.
\end{aligned}
$$

Výstupem takového volání jsou hledané modely. Funkce SEARCH používá pomocnou funkci PROPAGATE. Jejím vstupem je opět pětice $(A_{\mathcal{F}}, A_{\mathcal{P}}, U_{\mathcal{F}}, U_{\mathcal{P}}, E)$. Funkce PROPAGATE pak na danou pětici opakovaně aplikuje následující pravidla, dokud nenastane konflikt nebo dokud se pětice mění:

- Pro každé $(c, v) \in A_{\mathcal{F}}$ nahradit všechny výskyty c v E hodnotou v.
- Pro každé v, v' takové, že $v \neq v'$:
  - Z E odebrat podmínky, jenž obsahují literál $v \approx v$ nebo literál $v \not\approx v'$ nebo literál z $A_{\mathcal{P}}$.
  - Z každé podmínky v E odebrat literály tvaru $v \not\approx v$, literály tvaru $v \approx v'$ a literály tvaru $\neg L$, kde $L \in A_{\mathcal{P}}$.
- Pro každou podmínku s jedním literálem tvaru $c \approx v$ (resp. $v \approx c$), kde $(c, D) \in U_{\mathcal{F}}$ pro nějaké D, odebrat (c, D) z $U_{\mathcal{F}}$ a přidat (c, v) do $A_{\mathcal{F}}$.
- Pro každou podmínku s jedním literálem tvaru $c \not\approx v$ (resp. $v \not\approx c$), kde $(c, D) \in U_{\mathcal{F}}$ pro nějaké D, odebrat v z D.
- Pro každou podmínku s jedním literálem tvaru c, kde $c \in U_{\mathcal{P}}$, odebrat c z $U_{\mathcal{P}}$ a přidat c do $A_{\mathcal{P}}$.
- Pro každou podmínku s jedním literálem tvaru $\neg c$, kde $c \in U_{\mathcal{P}}$, odebrat c z $U_{\mathcal{P}}$ a přidat $\neg c$ do $A_{\mathcal{P}}$.
- Pro každou dvojici $(c, \{v\}) \in U_{\mathcal{F}}$ odebrat $(c, \{v\})$ z $U_{\mathcal{F}}$ a přidat (c, v) do $A_{\mathcal{F}}$.

V případě, že nastal konflikt, funkce PROPAGATE vrátí hodnotou `conflict`, v opačném případě je vrácena nová pětice.

3.1.3 Lepší propagace pro SEM

Popsali jsme základní varianty metod MACE a SEM, nyní se podíváme na jejich modifikace. První modifikací, kterou si ukážeme, bude lepší propagace pro metodu SEM.

Obsahuje-li E podmínku s jedním literálem tvaru $c \approx v$ (resp. $v \approx c$) a je-li $(c, D) \in U_{\mathcal{F}}$ pro nějaké D, pak funkce PROPAGATE přiřadí buňce c hodnotu v bez ohledu na to, zda $v \in D$. Pokud $v \notin D$, dojde v každém případě ke konfliktu – lepší by tedy bylo přiřazení vůbec neprovádět a okamžitě hlásit konflikt.

První dvě pravidla funkce PROPAGATE slouží ke zjednodušování množiny podmínek E. Tato pravidla využívají množiny $A_{\mathcal{F}}$ a $A_{\mathcal{P}}$ a nevyužívají množinu E. Například, máme-li

$$E = \Big\{ f(g(0)) \approx 1, f(g(0)) \approx g(0) \Big\}$$

a obsahuje-li $U_{\mathcal{F}}$ dvojice $(f(1), D)$ a $(g(0), D)$, kde $D = \{0, 1, 2, 3\}$, pak funkce PROPAGATE nic neudělá. Kdybychom však propagaci rozšířili o pravidlo demodulace, mohli bychom první podmínku $f(g(0)) \approx 1$ použít ke zjednodušení druhé podmínky $f(g(0)) \approx g(0)$, čímž bychom dostali:

$$E = \Big\{ f(g(0)) \approx 1, 1 \approx g(0) \Big\}.$$

S těmito podmínkami si již poradí i původní funkce PROPAGATE – buňkám $f(1)$ a $g(0)$ přiřadí hodnotu 1 a podmínky eliminuje.

Demodulace používá rovnosti, mohli bychom využít i nerovnosti? Je-li

$$E = \Big\{ f(2, 3) \approx 4, f(2, g(5)) \not\approx 4 \Big\},$$

pak buňce $g(5)$ nelze přiřadit hodnotu 3. Stejný závěr vyvodíme, když prohodíme rovnost s nerovností

$$E' = \Big\{ f(2, 3) \not\approx 4, f(2, g(5)) \approx 4 \Big\}$$

nebo když informace $f(2, 3) = 4$ resp. $f(2, 3) \neq 4$ bude pocházet z $A_{\mathcal{F}}$ resp. $U_{\mathcal{F}}$ místo z E resp. E'.

Obecně, obsahuje-li E podmínky tvaru

$$\begin{aligned} f(v_1, \ldots, v_n) &\approx v, \\ f(v_1, \ldots, v_{i-1}, c, v_{i+1}, \ldots, v_n) &\not\approx v, \end{aligned}$$

kde $v_1, \ldots, v_n \in \mathbb{N}_0$, pak odvodíme, že buňka c nemůže obsahovat hodnotu v_i. Stejný závěr lze učinit, když v první podmínce bude nerovnost a ve druhé rovnost nebo když informace $f(v_1, \ldots, v_n) = v$ resp. její negace nebude pocházet z E, ale z $A_{\mathcal{F}}$ resp. $U_{\mathcal{F}}$. Rovněž lze prohodit levou a pravou stranu rovnosti anebo nerovnosti. Analogické pravidlo lze formulovat i pro relace. Tento druh propagace je implementován v hledači modelů Mace4 [21], kde se nazývá negativní propagace.

V zobecňování bychom mohli pokračovat. Například z následujících podmínek

$$E = \Big\{ \neg P(0, g(2), g(2)), P(0, g(2), 1) \Big\}$$

bychom chtěli odvodit, že buňce $g(2)$ nelze přiřadit hodnotu 1. K tomu nám negativní propagace, jak jsme ji formulovali, nestačí. Místo ní bychom mohli použít pravidlo: Pokud E obsahuje podmínky tvaru $P(s_1, \ldots, s_n)$ a $\neg P(t_1, \ldots, t_n)$, pak buňkám nesmíme přiřadit hodnoty, aby $s_i = t_i$ pro všechna $i \in \{1, \ldots, n\}$. Je zřejmé, že jednu podmínku z E lze nahradit literálem z $A_{\mathcal{P}}$. Například je-li

$$E = \Big\{ P(g(1), g(1)) \Big\}$$

a $\neg P(0, 0) \in A_{\mathcal{P}}$, pak buňce $g(1)$ nelze přiřadit hodnotu 0.

3.1.4 Opakované použití výrokových klauzulí pro MACE

Předpokládejme, že chceme opakovaně hledat modely metodou MACE s tím, že při každém hledání zvětšíme domény některých sort.

Naivním řešením je vytvořit úplně novou instanci SATu pro každé hledání. To je ovšem zbytečné, neboť nová instance obsahuje skoro všechny výrokové klauzule z původní instance. Výjimkou jsou klauzule vytvořené v bodu 1c) pro funkce $f^{\mathcal{I}}$, kde sorta výsledku f patří mezi sorty, jejichž doménu zvětšujeme, a klauzule z nich odvozené například pomocí učení klauzulí [25].

S lepším řešením přišel program Paradox [9]. Před každým hledáním modelu se vytvoří nová výroková proměnná A. Literál A se přidá do klauzulí vytvářených v bodu 1c), které bude třeba odstranit. Jelikož je učení klauzulí založeno na rezoluci, literál A se rozšíří i do naučených klauzulí, které byly odvozeny z klauzulí, jenž bude třeba odstranit. SAT řešič je spuštěn s předpokladem $\neg A$, tudíž literál A neovlivní nalezený model. Pro následující hledání s většími doménami jsou klauzule obsahující literál A odstraněny přidáním jednotkové klauzule A.

Opakované hledání modelů metodou MACE se používá pro implementaci režimu hledání jednoho modelu, kdy jsou dána čísla $n \leq n'$ z $\mathbb{N}$ a hledáme model s doménou velikosti m, že $n \leq m \leq n'$.

3.1.5 Další pravidla pro zplošťování

Počet různých proměnných v klauzuli $C \in N$ ovlivňuje, do kolika výrokových klauzulí bude C zakódována metodou MACE a do kolika podmínek metodou SEM. Obecně počet výrokových klauzulí a počet podmínek závisí exponenciálně na počtu různých proměnných. Například, obsahuje-li klauzule C 3 proměnné z $\mathcal{X}_{S_1}$ a 5 proměnných z $\mathcal{X}_{S_2}$ a je-li $n_{S_1} = 4$ a $n_{S_2} = 8$, pak počet výrokových klauzulí resp. počet podmínek bude $4^3 \cdot 8^5 = 2^{21}$.

Jak je vidět, i pro klauzuli s pouhými 8 proměnnými může vzniknout velké množství výrokových klauzulí resp. podmínek. Připomeňme navíc, že zplošťování, které je třeba provést u metody MACE, přidává do klauzulí nové proměnné. Následující tři modifikace se tedy snaží transformovat N tak, aby klauzule v N obsahovaly méně proměnných.

Začneme tím, že vylepšíme zplošťování. Oproti pravidlům pro zplošťování, která jsme uvedli, obsahuje program Paradox [8] navíc následující pravidla:

- Klauzuli C, jenž obsahuje literál L tvaru $x \not\approx y$, můžeme upravit tak, že z ní odstraníme literál L a všechny výskyty x nahradíme y.
- Klauzuli $C = L_1 \vee \cdots \vee L_n$, kde literál L_i je tvaru $x \approx y$ a literál L_j je tvaru $t \not\approx y$, můžeme nahradit klauzulí $C' = t \approx x \vee \bigvee_{k \in \{1,\ldots,n\} \setminus \{i,j\}} L_k$, pokud C' neobsahuje y. Poznamenejme, že pravidlo rovněž platí, když prohodíme levou a pravou stranu některých rovností anebo nerovností.

První pravidlo je obsaženo i v článku [9], druhé je obsaženo pouze v implementaci.

3.1.6 Definice termů

Abychom zploštili klauzuli

$$f(g(c)) \approx x,$$

musíme do ní přidat dvě nové proměnné:

$$c \not\approx z \vee g(z) \not\approx y \vee f(y) \approx x.$$

Je-li velikost každé domény n, pak tato klauzule bude zakódována do n^3 výrokových klauzulí o 3 literálech.

Článek [9] ukazuje způsob, jak to udělat úsporněji. Pro term $g(c)$ zavedeme zkratku c'. Původní klauzuli nahradíme klauzulí

$$f(c') \approx x$$

a přidáme klauzuli

$$c' \approx g(c),$$

která zaručí, že c' je zkratka za $g(c)$. Zploštěním těchto klauzulí dostaneme

$$c' \not\approx y \vee f(y) \approx x,$$
$$c \not\approx y \vee g(y) \not\approx x \vee c' \approx x.$$

Počet výrokových klauzulí bude $2 \cdot n^2$ a počet literálů $5 \cdot n^2$.

Jako zkratka slouží funkční symbol[3], který se nevyskytuje v N. Zkratku můžeme zavést za libovolný term bez proměnných a lze s ní nahradit více výskytů daného termu i v různých klauzulích.

3.1.7 Rozdělování klauzulí

Třetí modifikace, která snižuje počty proměnných v klauzulích, je rozdělování klauzulí. Na rozdíl od předchozích dvou modifikací se používá nejen pro metodu MACE, ale i pro metodu SEM. Princip této metody je použít více klauzulí k nahrazení jedné klauzule $C \in N$. Samozřejmě, každá z nových klauzulí, jimiž nahrazujeme klauzuli C, musí obsahovat méně proměnných než C.

Začneme příkladem. Rozdělíme klauzuli z N

$$C = P_1(x) \vee P_2(y) \vee P_3(x, z)$$

na dvě klauzule s menším množstvím proměnných. První dva literály C dáme do první klauzule a zbylý literál C do druhé klauzule:

$$C_1 = P(x) \vee P_1(x) \vee P_2(y)$$
$$C_2 = \neg P(x) \vee P_3(x, z).$$

P je predikátový symbol, jenž se nevyskytuje v N. Propojovací literály $P(x)$ a $\neg P(x)$ zajišťují, že každý model původní klauzule C jde převést na model C_1 a C_2 rozšířením o vhodnou interpretaci P. Pro převod modelu opačným směrem

[3]Modely, jenž jsou výstupem programu, musí interpretovat právě symboly z původní vstupní množiny klauzulí N. Pomocné symboly zavedené při transformování množiny N jako například zkratka c' nesmí být součástí výstupních modelů. Naopak, pokud symbol z původní vstupní množiny N v důsledku určitých transformací z N vypadne, musí být k výstupnímu modelu přidán. Příkladem transformace, která může vyřadit symboly z N, je zjednodušování – odstraňuje klauzule, jenž jsou vždy splněné (klauzule obsahující literál $t \approx t$ nebo literály L a $\neg L$ nebo literály $t \approx s$ a $s \not\approx t$), odstraňuje klauzule, jenž jsou subsumovány jinými klauzulemi, a z klauzulí odstraňuje literály, jenž nejsou nikdy splněné ($t \not\approx t$).

stačí odstranit interpretaci P. Klauzuli se třemi proměnnými jsme rozdělili na dvě klauzule po dvou proměnných. Navíc klauzuli C_1 můžeme rozdělit na dvě klauzule po jedné proměnné.

Obecně můžeme klauzuli $C \in N$ s literály $L_1, \ldots, L_n$ rozdělit na dvě klauzule

$$C_1 = P(x_1, \ldots, x_n) \vee L_1 \vee \cdots \vee L_i,$$
$$C_2 = \neg P(x_1, \ldots, x_n) \vee L_{i+1} \vee \cdots \vee L_n,$$

kde P je predikátový symbol, jenž se nevyskytuje v N, a $x_1, \ldots, x_n$ jsou proměnné, jenž se vyskytují v $L_1, \ldots, L_i$ a zároveň v $L_{i+1}, \ldots, L_n$. Rozdělení provádíme pouze v případě, že obě klauzule mají méně proměnných než C.

Otázkou je, jak rozdělit literály původní klauzule C mezi C_1 a C_2. Algoritmus z článku [30] od autora programu Gandalf prochází všechny podmnožiny proměnných z C a pro každou podmnožinu $\mathcal{X}'$ zkouší literály z C rozdělit do dvou skupin tak, aby proměnné sdílené oběma skupinami $x_1, \ldots, x_n$ byly právě proměnné z $\mathcal{X}'$. Podmnožiny jsou procházeny v pořadí od nejmenší po největší – díky tomu algoritmus najde podmnožinu minimální velikosti a predikátový symbol P má minimální počet argumentů.

Program Paradox [9] používá jiný postup. Řekneme, že dvě proměnné jsou v klauzuli spojené, pokud existuje literál, v němž se obě vyskytují. Paradox hledá proměnnou x, jenž je spojena s minimem proměnných. Do klauzule C_1 dá právě literály obsahující x. Rozdělení se provede pouze, pokud x není spojena se všemi proměnnými (kdyby x byla spojena se všemi proměnnými, klauzule C_1 by obsahovala stejný počet proměnných jako C).

3.1.8 Redukce symetrií

Doposud jsme při hledání jednoho modelu nebo všech neizomorfních modelů prozkoumávali všechna možná ohodnocení buněk. Nyní si ukážeme, že to není třeba, že prozkoumáním určitých ohodnocení získáme informace i o jiných ohodnoceních, která pak prozkoumávat už nemusíme.

Například, když prozkoumáme interpretaci $\mathcal{I}$, nemusíme již prozkoumávat interpretace, které jsou s $\mathcal{I}$ izomorfní. Toto jednoduché pozorování lze zobecnit pro stavy řešeného problému. Mějme dva stavy řešeného problému z metody SEM $(A_\mathcal{F}, A_\mathcal{P}, U_\mathcal{F}, U_\mathcal{P}, E)$ a $(A'_\mathcal{F}, A'_\mathcal{P}, U'_\mathcal{F}, U'_\mathcal{P}, E')$. Řekneme, že tyto stavy jsou izomorfní, pokud zpermutováním prvků domén[4] dostaneme z trojice $(A_\mathcal{F}, A_\mathcal{P}, E)$ trojici $(A'_\mathcal{F}, A'_\mathcal{P}, E')$. Použité permutace tvoří dohromady jednu symetrii. Jestliže jsou dva stavy řešeného problému izomorfní, získáme jejich prohledáním izomorfní modely.

Dalším příkladem jsou symetrické symboly. Uvažme množinu klauzulí N, která obsahuje predikátové symboly P a P' takové, že jejich záměnnou se N nezmění. Pokud najdeme model N, tak záměnnou interpretací P a P' získáme jiný model N, který v obecném případě není izomorfní původnímu modelu. Toto pozorování lze použít i naopak, například, pokud zjistíme, že žádný model neobsahuje nějakou interpretaci pro P, pak automaticky dostaneme, že žádný model neobsahuje tuto interpretaci ani pro P'.

[4]Buňky v $A_\mathcal{F}, A_\mathcal{P}$ i podmínky E obsahují prvky domén. Pokud máme pro každou doménu permutaci, můžeme tyto permutace použít k přejmenování prvků domén, jenž se vyskytují $(A_\mathcal{F}, A_\mathcal{P}, E)$.

Chceme-li metodu SEM modifikovat, aby neprozkoumávala některá ohodnocení, o nichž získá nebo získala informace z jiných ohodnocení, máme dvě možnosti:

- Můžeme upravit množinu podmínek E tak, aby některé symetrie přestaly existovat,
- nebo upravit funkci SEARCH, aby neprohledávala symetrické stavy.

První možnosti se říká statická modifikace, neboť pracuje pouze na začátku před spuštěním funkce SEARCH. Druhé možnosti se říká dynamická modifikace, jelikož pracuje za běhu funkce SEARCH.

Statická modifikace u metody MACE upraví výrokové klauzule, dynamická modifikace upravuje samotný SAT řešič. Nyní popíšeme konkrétní modifikace.

LNH

Modifikace LNH [31] je založena na následujícím principu: Uspořádáme-li všechny buňky funkcí do posloupnosti $c_1, \ldots, c_n$, tak ke každému modelu existuje[5] izomorfní model, kde je každé buňce c_i přiřazena buď nějaká použitá hodnota, nebo nejnižší nepoužitá hodnota. Mezi použité hodnoty pro buňku c_i patří hodnoty, jenž se vyskytují jako argument buněk $c_1, \ldots, c_i$, a hodnoty přiřazené buňkám $c_1, \ldots, c_{i-1}$. Pozor, hodnoty z různých domén chápeme jako různé hodnoty.

Buňka c_i tedy může bez omezení nabývat hodnot, jenž se vyskytují mezi argumenty $c_1, \ldots, c_i$, a také nejnižší hodnoty v_1, která se mezi argumenty $c_1, \ldots, c_i$ nevyskytuje. Jsou-li $v_1 < \cdots < v_{n'}$ hodnoty, jenž jde přiřadit buňce c_i a jenž se nevyskytují mezi argumenty $c_1, \ldots, c_i$, pak buňka c_i může obsahovat hodnotu v_k pro $k \geq 2$ pouze tehdy, pokud nějaká buňka před c_i obsahuje hodnotu v_{k-1}.

Z výše uvedeného vyplývá, jak LNH implementovat jako statickou modifikaci. Pro buňku c_i a hodnotu v_k, kde $k \in \{2, \ldots, n'\}$, přidáme podmínku tvaru

$$c_i \approx v_k \implies \bigvee_{\substack{j \in \{1,\ldots,i-1\},\\ c_i \text{ a } c_j \text{ mají stejnou}\\ \text{sortu výsledku}}} c_j \approx v_{k-1}.$$

c_j označuje buňky nalevo od c_i, které mají stejnou sortu výsledku jako c_i, buňky s jinou sortou výsledku nabývají hodnot z jiných domén, tudíž by rovnost s v_{k-1} nemohla nastat.

Díky těmto podmínkám, nemohou některé buňky nabývat určitých hodnot. Například, máme-li buňky $c, f(0,0), f(0,1), \ldots$ a jednu doménu velikosti 3, pak buňka c může nabývat pouze hodnoty 0. Důvodem jsou dvě podmínky, které jsme pro ni vytvořili – jelikož žádné buňky nejsou nalevo od c, je pravá strana implikace s předpokladem $c = 1$ resp. $c = 2$ prázdná.

[5] Takový model můžeme snadno zkonstruovat. Ohodnocené buňky zapíšeme do posloupnosti $c_1 = v_1, \ldots, c_n = v_i$. Tuto posloupnost procházíme zleva doprava a kdykoliv najdeme $c_i = v_i$, kde v_i je nepoužitá hodnota, ale není to nejnižší nepoužitá hodnota v, tak v celé posloupnosti zaměníme v a v_i. Tato záměna se nedotkne buněk před c_i, neboť tam se hodnoty v a v_i nevyskytují. U buňky c_i se změní pouze přiřazená hodnota. Záměna však pokazí pořadí buněk po c_i, pokud se tam v a v_i vyskytují jako argumenty. Požadovaný model získáme, až dojdeme na konec posloupnosti. Ze záměn můžeme složit permutace, čímž dostaneme izomorfismus modelů.

Buňka $f(0,0)$ může nabývat pouze hodnot 0, 1. Důvodem je opět podmínka, kterou jsme pro ni vytvořili:

$$f(0,0) \approx 2 \implies c \approx 1.$$

Pravá strana implikace nikdy neplatí, neboť $c = 0$.

Skutečnosti, že některé buňky nenabývají určitých hodnot, můžeme využít v metodě MACE při generování výrokových klauzulí. Jednak pro tyto buňky není třeba generovat některé výrokové proměnné. Dále klauzule z bodu 1c) pro tyto buňky mohou obsahovat méně literálů a není třeba je odstraňovat při zvětšení domény. Viz také [9].

Velmi důležitou roli hraje pořadí buněk. Například, kdybychom dali buňku $f(0,1)$ jako první, tj. $f(0,1), c, f(0,0), \ldots$, tak by každá buňka mohla nabýt každé hodnoty z domény. Výroková klauzule z bodu 1c) pro buňku $f(0,1)$ by však platila i po zvětšení domény na velikost 4 – klauzuli bychom nemuseli odstraňovat.

Pro metodu MACE se LNH obvykle implementuje jako statická modifikace – je to jednodušší než upravovat SAT řešič. Na druhé straně pro metodu SEM se LNH obvykle implementuje dynamicky – ve funkci SEARCH stačí nahradit

for $v \in D$ **do**
  SEARCH$(A_{\mathcal{F}} \cup \{(c,v)\}, A_{\mathcal{P}}, U_{\mathcal{F}} \setminus \{(c,D)\}, U_{\mathcal{P}}, E)$
end for

pomocí (liší se pouze první řádek)

for $v \in D' \cup D''$ **do**
  SEARCH$(A_{\mathcal{F}} \cup \{(c,v)\}, A_{\mathcal{P}}, U_{\mathcal{F}} \setminus \{(c,D)\}, U_{\mathcal{P}}, E)$
end for

kde D' obsahuje hodnoty z D, jenž se vyskytují v $A_{\mathcal{F}}$ nebo v argumentech c, a D'' obsahuje nejmenší hodnotu z $D \setminus D'$, pokud taková hodnota existuje. D' je množina použitých hodnot a D'' je buď prázdná množina, nebo obsahuje nejmenší nepoužitou hodnotu.

Pořadí buněk pro LNH je tedy dáno pořadím, v němž byly buňky ohodnoceny, což znamená, že v různých větvích prohledávacího stromu může být různé pořadí buněk pro LNH. Abychom mohli omezit výběr hodnot, tj. omezit velikost D', je vhodné při výběru neohodnocených buněk z $U_{\mathcal{F}}$ dávat přednost buňkám, jejichž argumenty se již vyskytují v $A_{\mathcal{F}}$.

XLNH

Modifikace XLNH [2] redukuje symetrie u problémů s funkčním symbolem f arity $\langle S, S\rangle$. XLNH lze implementovat jako dynamickou modifikaci metody SEM:

Implementace postupně generuje všechna navzájem neizomorfní ohodnocení všech buněk funkcí bez argumentů a všech buněk funkce $f^{\mathcal{I}}$. S každým takovým ohodnocením $A_{\mathcal{F}}$ je pak spuštěna upravená funkce SEARCH, která vygeneruje ohodnocení pro zbylé buňky. Při ohodnocování zbylých buněk využívá upravená funkce SEARCH rozklad $f^{\mathcal{I}}$ na cykly k redukci dalších symetrií. Pokud $f^{\mathcal{I}}$ není bijekce, použije se její bijektivní restrikce.

Článek [2] jednak ukazuje, jak systematicky generovat neizomorfní ohodnocení všech buněk funkcí bez argumentů a všech buněk funkce $f^{\mathcal{I}}$. Dále ukazuje, jak použít $f^{\mathcal{I}}$ k redukci dalších symetrií.

Na rozdíl od modifikace LNH se XLNH staticky neimplementuje. Důvodem je velké množství podmínek resp. výrokových klauzulí, jenž by bylo nutné přidat.

DASH

Představené modifikace LNH a XLNH nejsou obecně schopny eliminovat všechny izomorfní modely. Pokud bychom něčeho takového chtěli dosáhnout v metodě SEM, mohli bychom si pamatovat všechny prohledané stavy řešeného problému a před prohledáváním nového stavu, se podívat, zda již nebyl prohledán izomorfní stav. Taková modifikace by vskutku eliminovala veškeré izomorfní modely a mohla by tak nahradit modifikace LNH a XLNH – viz [1]. Problém spočívá v tom, že stavů řešeného problému je velmi mnoho, tudíž hledat mezi uloženými stavy izomorfní stav může dlouho trvat a navíc se tolik stavů ani nemusí vejít do paměti běžného počítače.

S lepším řešením přišli autoři modifikace DASH [16]. Jelikož je na začátku množina podmínek E invariantní při zpermutování prvků domén, není třeba tuto množinu uvažovat. Buňky, jimž přiřadila hodnoty funkce PROPAGATE, rovněž není třeba uvažovat, stačí uvažovat pouze buňky, jimž přiřadila hodnoty funkce SEARCH. A nakonec, není třeba si pamatovat všechny stavy, některé stavy jsou redundantní.

Modifikace DASH začíná ohodnocením konstant. Konstanty jsou ohodnoceny jako při LNH – jsou uspořádány do posloupnosti a každé je buď přiřazena hodnota použitá nějakou předchozí konstantou, nebo nejmenší nepoužitá hodnota. Hodnoty přiřazené konstantám nejsou záměnné, ostatní hodnoty jsou záměnné. Prohledané stavy reprezentuje DASH schémata. Schéma je množina obsahující prvky z $A_{\mathcal{F}}$ a $A_{\mathcal{P}}$, jimž byla přiřazena hodnota funkcí SEARCH. Před prohledáváním jiného stavu[6] s ohodnocením $A'_{\mathcal{F}}$ a $A'_{\mathcal{P}}$ se zkusí najít schéma, jemuž tento stav odpovídá. Nalezení takového schématu znamená, že byl prohledán izomorfní stav řešeného problému, tudíž stav s ohodnocením $A'_{\mathcal{F}}$ a $A'_{\mathcal{P}}$ prohledávat nemusíme. Po prohledání stavu[7] přidáme schéma, jenž daný stav reprezentuje, a odebereme schémata přidaná pro jeho podstavy[8].

Dvě schémata jsou izomorfní, pokud jedno schéma dostaneme z druhého zpermutováním prvků domén mimo prvků, jenž byly přiřazeny konstantám. Stav s ohodnocením $A'_{\mathcal{F}}$ a $A'_{\mathcal{P}}$ odpovídá schématu, pokud prvky nějakého izomorfního schématu leží v množině $A'_{\mathcal{F}} \cup A'_{\mathcal{P}}$.

V praxi se ukázalo, že snaha eliminovat všechny izomorfní modely metodou DASH se nevyplatí. Proto autoři [16] v závislosti na hloubce rekurze SEARCH omezují, jaká schémata se zkouší.

DASH se staticky neimplementuje, důvody jsou stejné jako pro XLNH.

3.1.9 Omezení velikosti domén

Dalším trikem, jak zmenšit prohledávaný prostor, je omezit velikost domén některých sort [9]. Předpokládejme, že hledáme model, kde $n > k$ je velikost domény D_S, všechny použité funkční symboly $c_1, \ldots, c_k$ se sortou výsledku S jsou bez

[6] Po volání PROPAGATE ve funkci SEARCH – použití propagace zvyšuje šanci na nalezení schématu, jemuž stav odpovídá.

[7] Na konci funkce SEARCH, ale jen pokud nenastal konflikt, tj. $r \neq$ `conflict`.

[8] Schémata přidaná v rekurzivních voláních pro stavy, jenž vznikly z aktuálního stavu.

argumentů a navíc klauzule neobsahují literál tvaru $t \approx t'$, kde t, t' jsou termy sorty S. Takový model existuje právě tehdy, když existuje model, kde doména D_S má velikost k.

Pokud platí předpoklady, lze doménu D_S velikosti n zmenšit odebráním hodnot, jenž nejsou přiřazeny žádné z buněk $c_1, \ldots, c_k$. Naopak, když máme model s doménou D_S velikosti k, jenž obsahuje prvek v, můžeme tuto doménu rozšířit o hodnoty $v_1, \ldots, v_j$, přičemž hodnotu každé nové buňky c získáme tak, že výskyty hodnot $v_1, \ldots, v_j$ v argumentech buňky c nahradíme hodnotou v, čímž dostaneme jinou buňku c', hodnotu c definujeme jako hodnotu c'.

Pozorování můžeme upravit pro případ, kdy klauzule navíc obsahují literály $t \approx t'$, kde t, t' jsou termy sorty S a alespoň jeden z nich není proměnná. V takovém případě model s doménou D_S velikosti n existuje právě tehdy, když existuje model s doménou D_S velikosti $k + 1$.

Rovnosti termů sorty S nepředstavují žádný problém při zmenšování domény D_S z velikosti n na $k+1$. Při zvětšování domény víme, že existuje hodnota $v \in D_S$, která není přiřazena žádné z buněk $c_1, \ldots, c_k$. Hodnoty nových buněk definujeme stejným způsobem jako u předchozího pozorování, rozdíl je akorát v tom, že hodnota v není libovolná hodnota z D_S. Klauzule, které obsahují rovnosti termů sorty S, budou splněny i pro hodnoty přidané do domény, jelikož jsou splněny i pro hodnotu v.

Poznamenejme, že ač je klauzule

$$x \approx c$$

ekvivalentní s klauzulí

$$x \approx y \vee y \not\approx c,$$

tak u první klauzule se podaří omezit velikost sorty a u druhé klauzule se to nepodaří – druhá klauzule obsahuje rovnost proměnných. Postup, jak model s menší doménou převést na model s větší doménou, nezaručuje vygenerování všech neizomorfních modelů. Omezování velikostí domén tedy použijeme pouze v režimu, kdy hledáme jediný model.

3.1.10 Inference sort

Metody MACE a SEM a jejich modifikace podporují vstupy s více než jednou sortou. Vstupní množina klauzulí N však obsahuje nejvýše jednu sortu. Nyní se podíváme, jak vstupní množinu N transformovat na množinu N', která potenciálně obsahuje více sort než N.

Pokud bylo původním úkolem najít model N, kde je velikost jediné domény n, tak nyní budeme hledat model N', kde je velikost každé domény n. Takový vícesortový model snadno převedeme na model jednosortový – stačí zapomenout, že argumenty a hodnoty buněk jsou z různých sort.

Motivací pro N' s více sortami je skutečnost, že některé modifikace mají větší šanci omezit prohledávaný prostor, když N' obsahuje více sort. Například redukce symetrií LNH má k dispozici více nepoužitých hodnot, neboť hodnoty z různých domén chápeme jako různé. Jiným příkladem je omezování velikosti domén – velikost původní domény nemusí jít omezit, ale velikost některé z nových domén už ano.

Vraťme se k samotnému převodu N na N'. Používáme techniku z [9]. Označme

$$s = \{s_{a,n} \mid a \in \mathcal{F} \cup \mathcal{P} \cup \mathcal{X}, n \in \mathbb{N}\}$$

množinu, kde $s_{a,n} \neq s_{a',n'}$, pokud $a \neq a'$ nebo $n \neq n'$. Poznamenejme, že prvky množiny s nejsou termy.

Množina N' vznikne z množiny N tak, že

- každý funkční symbol f arity $\langle S_1, \ldots, S_n, S\rangle$ nahradíme funkčním symbolem f' arity $\langle \eta(s_{f,1}), \ldots, \eta(s_{f,n}), \eta(s_{f,n+1})\rangle$,
- každý predikátový symbol P arity $\langle S_1, \ldots, S_n\rangle$ nahradíme predikátovým symbolem P' arity $\langle \eta(s_{P,1}), \ldots, \eta(s_{P,n})\rangle$
- a každou proměnnou $x \in \mathcal{X}_S$ nahradíme proměnnou $x' \in \mathcal{X}_{\eta(s_{x,1})}$,

kde $\eta : s \to \mathcal{S}$ je takové zobrazení, že N' obsahuje klauzule a pro každé zobrazení σ s touto vlastností platí $\exists\sigma' : \sigma' \circ \eta = \sigma$.

N' by neobsahovala klauzule v případě, že by nesouhlasily sorty – term sorty S by byl na pozici, kde je vyžadován term sorty $S' \neq S$. Zobrazení η není určeno jednoznačně, jeho jádro

$$E = \ker \eta = \{(a, a') \in s^2 \mid \eta(a) = \eta(a')\}$$

však ano. Při konstrukci zobrazení η můžeme postupovat tak, že algoritmem union-find nejprve najdeme jeho jádro E (ekvivalentní budou jen ty prvky s, které ekvivalentní být musí, aby souhlasily sorty). Následně pak vezmeme libovolné zobrazení $\eta : s \to \mathcal{S}$ s jádrem E.

3.2 Další metody hledání modelů

Metody MACE a SEM jsou úspěšné při řešení mnoha praktických problémů – viz například výsledky soutěže CASC [28]. Na druhé straně existuje i řada poměrně jednoduchých problémů, s nimiž si tyto metody neporadí. Jedním z takových problémů je najít model následujících klauzulí:

$$f(x_1, \ldots, x_{128}) \approx c,$$
$$c \not\approx c'.$$

Klauzule mají zjevně model s doménou velikosti 2, kde funkce $f^{\mathcal{I}}$ je konstantní. Metody MACE a SEM však žádný model nenajdou. Problém spočívá v explicitní reprezentaci tabulky funkce $f^{\mathcal{I}}$. Pro doménu velikosti 2 obsahuje tato tabulka 2^{128} buněk, takže se ani nevejde do paměti počítače.

Další nepříjemnost metod MACE a SEM uvádí [15]. Mějme tyto klauzule:

$$f(x) \approx x,$$
$$P(f(g(x))).$$

V logice prvního řádu, lze druhou klauzuli zjednodušit pomocí první klauzule na $P(g(x))$. Pokud klauzule transformujeme na podmínky pro SEM, dostaneme:

$$f(v) \approx v,$$
$$P(f(g(v))),$$

kde v je prvek domény, tj. číslo z $\mathbb{N}_0$. Pak ovšem podmínky vzešlé z první klauzule již nejde[9] použít ke zjednodušení podmínek z druhé klauzule, důvodem je, že $g(v)$ není číslo.

Oba příklady naznačují, že by některé problémy bylo vhodné řešit přímo v logice prvního řádu. Budeme tedy postupovat podobně jako u metody MACE, ale místo do výrokové logiky převedeme problém do logiky prvního řádu, kde z funkčních symbolů jsou pouze symboly bez argumentů [5].

Začneme zploštěním všech klauzulí. Na rozdíl od zploštění, jenž jsme popsali pro metodu MACE, budeme navíc požadovat, aby rovnost termů byla pouze mezi proměnnými.

Dále všechny funkční symboly nahradíme predikátovými symboly. Pro každý funkční symbol f vezmeme dosud nepoužitý predikátový symbol P_f stejné arity. Každý literál tvaru $f(x_1, \ldots, x_n) \not\approx x$ (resp. $x \not\approx f(x_1, \ldots, x_n)$) nahradíme literálem $\neg P(x_1, \ldots, x_n, x)$.

Poté pro každý prvek v každé domény D_S vezmeme dosud nepoužitý funkční symbol $c_{S,v}$ arity $\langle S \rangle$. Pro každé dva funkční symboly $c_{S,v} \neq c_{S,v'}$ přidáme klauzuli $c_{S,v} \not\approx c_{S,v'}$.

Každá funkce má v každém bodě právě jednu hodnotu – tento invariant garantovaly u metody MACE klauzule přidané v bodech 1b) a 1c). Nyní nám stačí garantovat pouze to, že každá „funkce“ (reprezentovaná relací) má v každém bodě alespoň jednu hodnotu, což pro každý funkční symbol f z původního problému vyjádříme klauzulí

$$\bigvee_{\substack{v \in D_S, \\ \text{kde } S \text{ je sorta výsledku } f}} P_f(x_1, \ldots, x_n, c_{S,v}). \tag{3.1}$$

Díky tomu, že se P_f ve všech klauzulích kromě klauzule (3.1) vyskytuje pouze v negaci, můžeme nalezené modely transformovat tak, že P_f bude reprezentovat skutečnou funkci.

Jelikož takto zakódovaný problém neobsahuje funkční symboly s argumenty, jsou domény Herbrandových interpretací konečné. Abychom tedy našli konečný model, stačí problém předat metodě pro hledání Herbrandových modelů. Popsané kódování používají například programy FM-Darwin [5] a iProver[10] [18].

Samozřejmě metody pro hledání Herbrandových modelů lze použít i bez popsaného kódování, pak ovšem nalezené modely nemusí mít konečné domény. Rozsáhlý přehled metod automatického dokazování v logice prvního řádu založených na budování modelů je [7]. Tento přehled mj. zmiňuje i metody Model Evolution [6] a Inst-Gen [19], jenž jsou implementovány v programech FM-Darwin a iProver.

[9] Metodu SEM by bylo možné upravit, aby u plně interpretovaných funkcí držela informaci, zda jsou konstantní nebo zda se vůči nějakému argumentu chovají jako identita. Takové funkce by pak bylo možné použít ke zjednodušování množiny podmínek E.

[10] Pouze v režimu hledání konečných modelů.

KAPITOLA 4

Implementace

Cílem této kapitoly je popsat náš hledač konečných modelů, který se jmenuje Crossbow. Crossbow implementuje metodu MACE a některé její modifikace. Kromě modifikací obsažených v minulé kapitole jsou implementovány i úplně nové, dosud nevyzkoušené, modifikace. Tato kapitola začíná popisem nových modifikací metody MACE. Poté následuje stručný popis samotné implementace.

Hlavním důvodem, proč jsme se rozhodli založit program Crossbow právě na metodě MACE, byly výsledky metody MACE při řešení problémů – například program Paradox [8] vyhrál 6 ročníků soutěže CASC [28]. Další příjemnou vlastností metody MACE je, že funguje s existujícími SAT řešiči – není třeba vytvářet specializovaný řešič jako u metody SEM.

4.1 Další modifikace metody MACE

4.1.1 Zplošťování

Jako první popíšeme algoritmus zplošťování používaný programem Crossbow. Pravidla pro zplošťování klauzulí uvedená v předchozí kapitole lze použít mnoha různými způsoby. Například klauzuli

$$f(x) \approx c \vee g(x) \approx c \tag{4.1}$$

můžeme zploštit na

$$f(x) \not\approx y \vee g(x) \not\approx z \vee y \approx c \vee z \approx c \tag{4.2}$$

nebo na

$$c \not\approx y \vee f(x) \approx y \vee g(x) \approx y. \tag{4.3}$$

V prvním případě byly přidány dvě nové proměnné, ve druhém případě pouze jedna nová proměnná. Náš algoritmus se pochopitelně snaží přidat co nejméně proměnných.

Vstupem algoritmu je klauzule, jenž se má zploštit. Výstupem algoritmu je buď plochá klauzule, nebo rozhodnutí, že se má klauzule odstranit, neboť je vždy splněná.

Algoritmus se skládá ze sedmi kroků, které se, pokud není řečeno jinak, vykonávají postupně. Každý krok se skládá z předpokladu a akce. Akce se provádí

pouze v případě, že je splněn předpoklad. Pokud není splněn předpoklad, pokračuje se dalším krokem. Některé kroky využívají tzv. nové proměnné, což jsou proměnné, které se v klauzuli nevyskytovaly předtím, než se daný krok začal vykonávat.

1) **Předpoklad:** Klauzule obsahuje literál $t \approx t$ nebo literály L a $\neg L$ nebo literály $t \approx s$ a $s \not\approx t$.

   Akce: Klauzule se odstraní, zplošťování končí.

2) **Předpoklad:** Klauzule obsahuje literál L tvaru $t \not\approx t$.

   Akce: Literál L se odstraní z klauzule. Dále se pokračuje krokem 2).

3) **Předpoklad:** Klauzule obsahuje literál L tvaru $x \not\approx y$.

   Akce: Z klauzule se odstraní literál L a všechny výskyty x se nahradí y. Dále se pokračuje krokem 1).

4) **Předpoklad:** Klauzule obsahuje literál L tvaru $t \not\approx x$ (resp. $x \not\approx t$) a t se vyskytuje mimo L.

   Akce: Všechny výskyty t mimo výskytu v L se nahradí x. Dále se pokračuje krokem 1).

5) **Předpoklad:** Klauzule obsahuje atom $P(\ldots, t, \ldots)$ nebo $f(\ldots, t, \ldots) \approx s$ (resp. $s \approx f(\ldots, t, \ldots)$), kde t není proměnná.

   Akce: Do klauzule se přidá literál $t \not\approx x$, kde x je nová proměnná. Dále se pokračuje krokem 4).

6) **Předpoklad:** Klauzule obsahuje literál L tvaru $t \not\approx s$, kde t a s nejsou proměnné.

   Akce: Z klauzule se odstraní literál L a přidají se tam literály $t \not\approx x$ a $x \not\approx s$, kde x je nová proměnná. Dále se pokračuje krokem 4).

7) Nechť $t_1 \approx t_2, \ldots, t_{2n-1} \approx t_{2n}$ jsou všechny literály z klauzule, jenž mají tvar $t \approx s$, kde t ani s nejsou proměnné. (Tj. t_i pro každé $i \in \{1, \ldots, 2n\}$ není proměnná.)

   Předpoklad: $n \geq 1$.

   Akce: Vytvoří se neorientovaný graf s vrcholy $t_1, \ldots, t_{2n}$. Mezi vrcholy t_i a t_j vede hrana právě tehdy, když literál $t_i \approx t_j$ je v klauzuli. Najde se minimální vrcholové pokrytí $t_{i_1}, \ldots, t_{i_k}$. Do klauzule se přidá literál $t_{i_j} \not\approx x_j$ pro každé $j \in \{1, \ldots, k\}$, kde $x_1, \ldots, x_k$ jsou navzájem různé nové proměnné. Dále se pokračuje krokem 4).

Například u klauzule (4.1) se v pravidle 7) vytvoří graf s vrcholy $f(x), c, g(x)$ a hranami $\{f(x), c\}$ a $\{g(x), c\}$. Minimální vrcholové pokrytí tohoto grafu je vrchol c, proto se do klauzule přidá literál $c \not\approx y$. Dále se pokračuje krokem 4), který nahradí výskyty c mimo literál $c \not\approx y$, čímž dostaneme klauzuli (4.3).

Akci v pravidle 6) bychom mohli změnit tak, že by se literál L neodstraňoval a pouze by se přidal jeden z literálů $t \not\approx x$, nebo $x \not\approx s$. Pravidlo 4) by pak s pomocí přidaného literálu změnilo literál L na nepřidaný literál.

Všimněme si asymetrie mezi rovností a nerovností neproměnných. Pravidlo 6) přidáním jedné nové proměnné získá dva literály $t \not\approx x$ a $x \not\approx s$, které může použít pravidlo 4) k nahrazení termů t a s proměnnou x.

Na druhé straně pravidlo 7) použité na rovnost neproměnných $t \approx s$ přidáním jedné nové proměnné získá pouze jeden z literálů $t \not\approx x$, nebo $s \not\approx x$, které může použít pravidlo 4). Abychom získali i druhý literál, museli bychom přidat další novou proměnnou. Na rozdíl od pravidla 6) tedy pravidlo 7) musí pečlivě zvažovat, zda chce literál pro levou stranu rovnosti, nebo pro pravou stranu rovnosti, nebo literály pro obě strany rovnosti.

Například pro zploštění klauzule

$$c \approx c' \vee c \approx c_1 \vee c \approx c_2 \vee c' \approx c'_1 \vee c' \approx c'_2,$$

kde $c, c_1, c_2, c', c'_1, c'_2$ jsou funkční symboly bez argumentů, přidá pravidlo 7) literály $c \not\approx x_1$ a $c' \not\approx x_2$, jenž obsahují dvě nové proměnné x_1 a x_2. Kdyby však první literál klauzule byl nerovnost, tj. $c \not\approx c'$, tak by pravidlu 6) stačila pouze jedna nová proměnná.

4.1.2 Odzploštování

Jelikož preferujeme klauzule s malým počtem proměnných, snažili jsme se náš zplošťovací algoritmus navrhnout tak, aby zaváděl co nejmenší množství nových proměnných. Ovšem i tak existují klauzule, které jde zploštit lépe. Příkladem takové klauzule je (4.2). Tato klauzule již je plochá a náš algoritmus ji nijak nezmění. Víme však, že existuje ekvivalentní plochá klauzule, jenž obsahuje méně proměnných, tou klauzulí je (4.3).

Abychom při vstupu (4.2) dostali výstup (4.3), provedeme odzploštování před zplošťováním. Odzploštování je založeno na stejných pravidlech jako zploštování (viz sekce 3.1.1). Rozdíl je v tom, že odzploštování používá pravidla naopak:

- Je-li L literál z C tvaru $t \not\approx x$ (resp. $x \not\approx t$) takový, že x není obsažena v t ani v ostatních literálech C mimo L, pak můžeme literál L odstranit[1] z klauzule C.

- Je-li L literál z C tvaru $t \not\approx x$ (resp. $x \not\approx t$), pak můžeme C upravit tak, že jeden výskyt x mimo L nahradíme t.

Zkombinováním obou opačných pravidel dostaneme pravidlo odzploštování:

- Je-li L literál z C tvaru $t \not\approx x$ (resp. $x \not\approx t$) takový, že x není obsažena v t, pak můžeme všechny výskyty x mimo výskytu v L nahradit t a literál L odstranit.

Odzploštování klauzule C se provádí tak, že se na ni aplikuje uvedené pravidlo, dokud se klauzule mění.

[1] Ve skutečnosti se nejedná o inverzní pravidlo k prvnímu pravidlu zplošťování. Jednak proto, že při zplošťování se přidávaly literály tvaru $t \not\approx x$, nyní se však odebírají i literály tvaru $x \not\approx t$. Dále proto, že při zplošťování se přidával literál $t \not\approx x$ pouze v případě, kdy se term t vyskytoval v klauzuli C. Nyní se literál L odstraňuje, i když se term t v ostatních literálech mimo L nevyskytuje.

Například z klauzule (4.2) vytvoří odzplošťování klauzuli (4.1), kterou pak zplošťování převede na klauzuli (4.3).

Poznamenejme, že náš algoritmus zplošťování neobsahuje následující pravidlo z programu Paradox, jenž jsme uvedli v sekci 3.1.5:

- Klauzuli $C = L_1 \vee \cdots \vee L_n$, kde literál L_i je tvaru $x \approx y$ a literál L_j je tvaru $t \not\approx y$, můžeme nahradit klauzulí $C' = t \approx x \vee \bigvee_{k \in \{1,\ldots,n\} \setminus \{i,j\}} L_k$, pokud C' neobsahuje y.

Místo něj lze totiž použít odzplošťování (term t neobsahuje proměnnou y) následované zplošťováním.

4.1.3 Redundantní klauzule

Abychom posílili propagaci SAT řešiče, přidáme ke vstupní množině klauzulí N další klauzule, jenž plynou z N.

Program Crossbow tyto klauzule získá pomocí dokazovače E [24]. Z klauzulí odvozených dokazovačem E vybere malé klauzule a přidá je do N.

4.1.4 Komutativní funkce

Pokud víme, že funkce $f^{\mathcal{I}}$ je komutativní, můžeme pro buňky $f(v_1, v_2)$ a $f(v_2, v_1)$ používat stejné výrokové proměnné. To znamená, že při vytváření instance SATu (viz 3.1.1) budeme v kroku 1) přidávat výrokové proměnné a klauzule pouze pro buňky $f(v_1, v_2)$, kde $v_1 \leq v_2$. Každou výrokovou proměnnou $A_{f(v_1,v_2)=v}$, kde $v_1 > v_2$, nahradíme proměnnou $A_{f(v_2,v_1)=v}$.

Program Crossbow mezi vstupními klauzulemi a mezi všemi klauzulemi z dokazovače E hledá klauzuli ekvivalentní s klauzulí $f(x, y) \approx f(y, x)$. Pokud takovou klauzuli najde, označí funkci $f^{\mathcal{I}}$ jako komutativní a při vytváření instance SATu se použije optimalizace uvedená v předchozím odstavci.

Podobně bychom mohli postupovat i u symetrických relací, to však program Crossbow nedělá.

4.1.5 Převod pro Gecode

Chceme-li se vyhnout zplošťování klauzulí a s ním spojenému nárůstu počtu proměnných, můžeme klauzule kódovat do bohatšího jazyka, než je SAT. Program Crossbow umí klauzule kódovat do jazyka omezujících podmínek řešiče Gecode [14]. Při kódování se používají booleovské proměnné, celočíselné proměnné, pole booleovských i celočíselných proměnných, celočíselné konstanty a jejich pole. Pole jsou indexovány od 0. Booleovské proměnné značíme písmenem b, celočíselné proměnné nebo konstanty značíme písmenem i, pole booleovských proměnných značíme písmenem B a pole celočíselných proměnných nebo konstant značíme písmenem I. Z podmínek se používají:

- EQ(i_1, i_2, b). Podmínka je splněna právě tehdy, když $i_1 = i_2 \wedge b = 1$ nebo $i_1 \neq i_2 \wedge b = 0$.

- LOWER-EQ(i_1, i_2). Podmínka je splněna právě tehdy, když $i_1 \leq i_2$.

- Clause(B_P, B_N). Podmínka je splněna právě tehdy, když je nějaká booleovská proměnná z B_P ohodnocena 1 nebo je nějaká booleovská proměnná z B_N ohodnocena 0.

- Element(H, i, h), kde H resp. h je buď pole booleovských proměnných resp. booleovská proměnná, nebo pole celočíselných proměnných resp. celočíselná proměnná. i je vždy celočíselná proměnná. Podmínka je splněna právě tehdy, když hodnota proměnné v poli H s indexem i je stejná jako hodnota proměnné h.

- Linear(I_c, I_x, i), kde I_c a I_x jsou pole stejné délky, I_c je pole konstant, I_x je pole proměnných a i je konstanta. Podmínka je splněna právě tehdy, když

$$\sum_{k=0}^{|I_c|-1} I_c(k) \cdot I_x(k) = i.$$

- Precede(I_x, I_c), kde I_x je pole proměnných a I_c je pole konstant. Podmínka je splněna právě tehdy, když pro každé $j \in \{0, \ldots, |I_x| - 1\}$ a pro každé $k \in \{1, \ldots, |I_c| - 1\}$ platí

$$I_x(j) = I_c(k) \implies \exists j' \in \{0, \ldots, j-1\} : I_x(j') = I_c(k-1).$$

  Jinak řečeno, kdykoliv má proměnná z I_x hodnotu i z $I_c(1, \ldots, \star)$, tak v I_x existuje proměnná s nižším indexem a s hodnotou, jenž je v poli I_c těsně před i.

Začneme tím, že ukážeme, jak do uvedených podmínek zakódovat ploché klauzule. Následně výklad rozšíříme i o neploché klauzule a o LNH.

Ploché klauzule

Pro zakódování plochých klauzulí stačí podmínky Eq a Clause. Napřed pro každou funkci $f^{\mathcal{I}}$ a každou její buňku $c \in \mathcal{C}_f$ přidáme celočíselnou proměnnou i_c s doménou D_S, kde S je sorta výsledku f. Dále pro každou relaci $P^{\mathcal{I}}$ a každou její buňku $c \in \mathcal{C}_P$ přidáme booleovskou proměnnou b_c. Při přidávání proměnných pro buňky uplatňuje program Crossbow trik s komutativními funkcemi.

Nyní přidáme podmínky pro klauzule. Mějme klauzuli $C \in N$ a ohodnocení proměnných β. Pokud C obsahuje literál L s atomem $x \approx y$ takový, že L je splněný při ohodnocení β, tak pro klauzuli C a ohodnocení β žádné podmínky nepřidáváme. Dále předpokládejme, že klauzule žádný takový literál neobsahuje. Označme $L_1, \ldots, L_j$ všechny literály z klauzule, jenž neobsahují atom tvaru $x \approx y$. Každému literálu z L_k z $L_1, \ldots, L_j$ přiřadíme booleovskou proměnnou b_k:

- Obsahuje-li L_k atom tvaru $f(x_1, \ldots, x_n) \approx x$ (resp. $x \approx f(x_1, \ldots, x_n)$), tak b_k bude nová booleovská proměnná a přidáme podmínku

$$\text{Eq}(i_{f(\beta(x_1),\ldots,\beta(x_n))}, \beta(x), b_k).$$

- Obsahuje-li L_k atom tvaru $P(x_1, \ldots, x_n)$, tak b_k bude existující proměnná

$$b_{P(\beta(x_1),\ldots,\beta(x_n))}$$

  a žádnou podmínku nebudeme přidávat.

Následně přidáme podmínku $\textsc{Clause}(B_P, B_N)$, kde pole proměnných B_P obsahuje právě ty proměnné b_k, jenž jsou přiřazeny literálům bez negace. Naopak pole proměnných B_N obsahuje právě ty proměnné b_k, jenž jsou přiřazeny literálům s negací.

Abychom dostali podmínky pro všechny klauzule, použijeme uvedený postup na každou klauzuli $C \in N$ se všemi ohodnoceními β, jenž se liší na proměnných v klauzuli C.

Ukažme si, jak získat podmínky pro klauzuli $P(x) \vee f(x) \not\approx y \vee x \approx y$ s jedinou doménou velikosti 2. Uvedený postup musíme aplikovat na 4 ohodnocení $\beta_1, \beta_2, \beta_3$ a β_4, která splňují:

$$\begin{aligned} \beta_1(x) &= 0, & \beta_1(y) &= 0, \\ \beta_2(x) &= 0, & \beta_2(y) &= 1, \\ \beta_3(x) &= 1, & \beta_3(y) &= 0, \\ \beta_4(x) &= 1, & \beta_4(y) &= 1. \end{aligned}$$

Jelikož je literál $x \approx y$ splněný při ohodnoceních β_1 a β_4, nebudeme pro tato ohodnocení žádné podmínky přidávat. Pro obě zbylá ohodnocení označíme literály, jenž neobsahují atom rovnosti:

$$\begin{aligned} L_1 &= P(x), \\ L_2 &= f(x) \not\approx y. \end{aligned}$$

Pro ohodnocení β_2 přiřadíme literálu L_1 existující booleovskou proměnnou $b_{P(0)}$, literálu L_2 přiřadíme novou booleovskou proměnnou $b_2^{\beta_2}$ a přidáme podmínku $\textsc{Eq}(i_{f(0)}, 1, b_2^{\beta_2})$. Nakonec pro ohodnocení β_2 přidáme podmínku

$$\textsc{Clause}([b_{P(0)}], [b_2^{\beta_2}]).$$

Pro ohodnocení β_3 přiřadíme literálu L_1 existující booleovskou proměnnou $b_{P(1)}$, literálu L_2 přiřadíme novou booleovskou proměnnou $b_2^{\beta_3}$ a přidáme podmínku $\textsc{Eq}(i_{f(1)}, 0, b_2^{\beta_3})$. Nakonec pro ohodnocení β_3 přidáme podmínku

$$\textsc{Clause}([b_{P(1)}], [b_2^{\beta_3}]).$$

Pro klauzuli $P(x) \vee f(x) \not\approx y \vee x \approx y$ jsme dohromady přidali 4 podmínky.

Pro literály s atomem tvaru $f(x_1, \ldots, x_n) \approx x$ (resp. $x \approx f(x_1, \ldots, x_n)$), není třeba v některých situacích přidávat novou booleovskou proměnnou a podmínku. Uvažme například klauzuli $f(x) \not\approx x \vee y \approx c$ s jedinou doménou velikosti 2.

Označme literály klauzule

$$\begin{aligned} L_1 &= f(x) \not\approx x, \\ L_2 &= y \approx c. \end{aligned}$$

Následující tabulka ukazuje proměnné a podmínky, jenž je třeba přidat pro oba literály pro ohodnocení $\beta_1, \beta_2, \beta_3$ a β_4:

	L_1	L_2
β_1	$\textsc{Eq}(i_{f(0)}, 0, b_1^{\beta_1})$	$\textsc{Eq}(i_c, 0, b_2^{\beta_1})$
β_2	$\textsc{Eq}(i_{f(0)}, 0, b_1^{\beta_2})$	$\textsc{Eq}(i_c, 1, b_2^{\beta_2})$
β_3	$\textsc{Eq}(i_{f(1)}, 1, b_1^{\beta_3})$	$\textsc{Eq}(i_c, 0, b_2^{\beta_3})$
β_4	$\textsc{Eq}(i_{f(1)}, 1, b_1^{\beta_4})$	$\textsc{Eq}(i_c, 1, b_2^{\beta_4})$

Z tabulky je patrné, že pro literál L_1, pro ohodnocení β_2 můžeme použít stejnou proměnnou a podmínku jako pro ohodnocení β_1. Podobně pro ohodnocení β_3 můžeme použít stejnou proměnnou a podmínku jako pro ohodnocení β_4. Pro literál L_2 můžeme rovněž ušetřit dvě booleovské proměnné a dvě podmínky.

Obecné klauzule

Pro každý funkční symbol f arity $\langle S_1, \ldots, S_n, S \rangle$ přidáme pole I_f celočíselných proměnných. Délka pole bude $\prod_{j=1}^{n} n_{S_j}$. Pole bude obsahovat proměnné pro buňky funkce $f^{\mathcal{I}}$. Proměnná pro buňku $f(v_1, \ldots, v_n) \in \mathcal{C}_f$ bude v poli I_f pod indexem $\textsc{Horner}(v_1, \ldots, v_n, n_{S_1}, \ldots, n_{S_n})$. Funkce Horner je definována následovně:

function $\textsc{Horner}(v_1, \ldots, v_n, n_{S_1}, \ldots, n_{S_n})$
  $j \leftarrow 0$
  for $k = 1, \ldots, n$ **do**
    $j \leftarrow j \cdot n_{S_k} + v_k$
  end for
  return j
end function

Pro každý predikátový symbol P arity $\langle S_1, \ldots, S_n \rangle$ přidáme pole B_P booleovských proměnných. Délka pole bude $\prod_{j=1}^{n} n_{S_j}$. Pole bude obsahovat booleovské proměnné pro buňky $P^{\mathcal{I}}$. Proměnná pro buňku $P(v_1, \ldots, v_n) \in \mathcal{C}_P$ bude v poli B_P pod indexem $\textsc{Horner}(v_1, \ldots, v_n, n_{S_1}, \ldots, n_{S_n})$.

Díky právě přidaným polím můžeme zakódovat neplochou klauzuli $f(c) \approx c$. Zavedeme pomocnou proměnnou i a přidáme podmínku $\textsc{Element}(I_f, i_c, i)$, která zajistí, že proměnná i obsahuje hodnotu funkce $f^{\mathcal{I}}$ v bodě $c^{\mathcal{I}}$. Jelikož se tato hodnota musí rovnat hodnotě $c^{\mathcal{I}}$, přidáme stejně jako v případě plochých klauzulí pomocnou booleovskou proměnnou b a podmínku $\textsc{Eq}(i, i_c, b)$. Nakonec přidáme podmínku $\textsc{Clause}([b], [])$.

Důvodem, proč pro zakódování klauzule potřebujeme pole I_f, je, že nevíme, jakou buňku $f(c)$ přesně označuje (v případě plochých klauzulí jsme to věděli), a tudíž do podmínky Eq nemůžeme dát proměnnou přímo pro tuto buňku. Víme pouze to, že proměnná pro buňku $f(c)$ je v poli I_f a má index $c^{\mathcal{I}}$ – k získání její hodnoty použijeme podmínku Element.

V obecném případě budeme potřebovat zjistit, jakou buňku označuje term $f(t_1, \ldots, t_n)$ resp. atom $P(t_1, \ldots, t_n)$ při ohodnocení proměnných β. Předpokládejme, že máme k dispozici funkci Index takovou, že volání $\textsc{Index}(\beta, t_1, \ldots, t_n)$ vrátí index proměnné pro buňku $f(t_1, \ldots, t_n)$ resp. $P(t_1, \ldots, t_n)$ v poli I_f resp. B_P při ohodnocení β. Index je buď celočíselná proměnná, nebo konstanta. S pomocí funkce Index vytvoříme funkce:

- $\textsc{Term-Value}(\beta, t)$. Vrací celočíselnou proměnnou nebo konstantu s hodnotou termu t při ohodnocení proměnných β.
- $\textsc{Var-For-Atom}(\beta, A)$. Vrací booleovskou proměnnou, jenž obsahuje hodnotu atomu A při ohodnocení proměnných β.

Je-li t term tvaru x, funkce $\textsc{Term-Value}(\beta, t)$ vrátí konstantu $\beta(x)$. Je-li t term tvaru $f(t_1, \ldots, t_n)$, funkce $\textsc{Term-Value}(\beta, t)$ zavolá $\textsc{Index}(\beta, t_1, \ldots, t_n)$,

čímž dostane index do pole I_f. Pokud je index konstanta, vrátí funkce proměnnou z pole I_f s daným indexem. V opačném případě je index proměnná i. Pak funkce TERM-VALUE zavede novou pomocnou proměnnou i', přidá podmínku ELEMENT(I_f, i, i') a vrátí proměnnou i'.

Je-li A atom tvaru $t \approx s$, VAR-FOR-ATOM(β, A) zavolá TERM-VALUE(β, t) a TERM-VALUE(β, s), čímž dostane proměnné nebo konstanty i a i' s hodnotami t a s. Dále VAR-FOR-ATOM zavede novou pomocnou proměnnou b, přidá podmínku EQ(i, i', b) a vrátí proměnnou b. Je-li A atom tvaru $P(t_1, \ldots, t_n)$, postupuje funkce VAR-FOR-ATOM(β, A) analogicky jako funkce TERM-VALUE v případě, že term není proměnná.

Funkce INDEX$(\beta, t_1, \ldots, t_n)$ pro každý term t_j, kde $j \in \{1, \ldots, n\}$ zavolá funkci TERM-VALUE(β, t_j), čímž dostane proměnné nebo konstanty $i_1, \ldots, i_n$. Mají-li termy $t_1, \ldots, t_n$ sorty $S_1, \ldots, S_n$, pak INDEX musí vrátit hodnotu HORNER$(i_1, \ldots, i_n, n_{S_1}, \ldots, n_{S_n})$. Pokud jsou všechny $i_1, \ldots, i_n$ konstanty, tak vrácená hodnota bude rovněž konstanta. Pokud je mezi $i_1, \ldots, i_n$ proměnná, pak funkce INDEX zavede pomocnou proměnnou i, pomocí podmínky LINEAR zajistí, že i má hodnotu HORNER$(i_1, \ldots, i_n, n_{S_1}, \ldots, n_{S_n})$, a vrátí proměnnou i.

Při konstrukci podmínek CLAUSE se literálům přiřadí booleovské proměnné vrácené funkcí VAR-FOR-ATOM.

Převod neploché klauzule do podmínek řešiče Gecode si ukážeme na klauzuli $f(g(x), y, g(x)) \approx y$, kde proměnné x, y mají sortu s doménou velikosti 2 a sorta výsledku g má doménu velikosti 3.

Začneme s podmínkami pro ohodnocení β_1. Abychom přiřadili booleovskou proměnnou jedinému literálu z klauzule, zavoláme funkci VAR-FOR-ATOM$(\beta_1$, $f(g(x), y, g(x)) \approx y)$. Tato funkce následně zavolá

$$\text{TERM-VALUE}(\beta_1, f(g(x), y, g(x))) \quad \text{a}$$
$$\text{TERM-VALUE}(\beta_1, y).$$

Druhé volání vrátí hodnotu proměnné y při ohodnocení β_1, což je konstanta 0. První volání napřed spočte hodnoty termů $x, g(x), y$. Hodnotou termů x a y je konstanta 0. Hodnota termu $g(x)$ je nultá proměnná z pole I_g, tj. $i_{g(0)}$. Na základě těchto hodnot musí funkce INDEX spočítat index proměnné v poli I_f, jenž obsahuje hodnotu termu $f(g(x), y, g(x))$. Víme, že hodnota indexu je dána HORNER$(i_{g(0)}, 0, i_{g(0)}, 3, 2, 3)$. Zavedeme-li pro index novou pomocnou proměnnou i_1, pak musí platit

$$(i_{g(0)} \cdot 2 + 0) \cdot 3 + i_{g(0)} = i_1.$$

Úpravou dostaneme rovnici

$$7 \cdot i_{g(0)} + (-1) \cdot i_1 = 0,$$

jenž zakódujeme pomocí podmínky LINEAR$([i_{g(0)}, i_1], [7, -1], 0)$. Funkce INDEX vrátí proměnnou i_1. Funkce TERM-VALUE zavede novou pomocnou proměnnou i'_1 a pomocí podmínky ELEMENT(I_f, i_1, i'_1) zajistí, že i'_1 bude obsahovat hodnotu termu $f(g(x), y, g(x))$. Funkce TERM-VALUE$(\beta_1, f(g(x), y, g(x)))$ vrátí proměnou i'_1.

Nyní jsme zpět ve funkci VAR-FOR-ATOM a známe hodnoty termů na levé a pravé straně. Jelikož se tyto hodnoty mají rovnat, přidá VAR-FOR-ATOM novou

booleovskou proměnnou b_1, podmínku $\text{Eq}(i'_1, 0, b_1)$ a vrátí b_1. Nakonec je přidána podmínka $\text{Clause}([b_1], [])$. Stejně lze postupovat pro ohodnocení β_2, β_3 a β_4.

Pomocné proměnné zavedené pro podmínky Linear, Element a Eq lze používat opakovaně (pro podmínky Eq viz konec sekce Ploché klauzule).

LNH

Redukci symetrií LNH implementujeme prostřednictvím podmínek Lower-Eq a Precede. Popíšeme LNH pro funkční symboly se sortou výsledku S. Předpokládáme, že všechny hodnoty z D_S jsou nepoužité.

Buňky funkčních symbolů se sortou výsledku S uspořádáme do posloupnosti. Buňky, jenž nemají žádný argument sorty S, budou před buňkami, jenž mají alespoň jeden argument sorty S. Buňky s argumenty sorty S budou uspořádány tak, že je-li buňka c před buňkou c', tak maximální argument z D_S buňky c není větší než maximální argument z D_S buňky c'.

Pro buňky funkčních symbolů $c_1, \ldots, c_n$, jenž nemají argument z D_S, se přidá podmínka

$$\begin{aligned}\text{Precede}([&i_{c_1}, \ldots, i_{c_n}],\\ &[0, \ldots, \min\{n, n_S\} - 1]).\end{aligned}$$

Pro buňky $c'_1, \ldots, c'_{n'}$, kde maximální argument z D_S je 0, se přidá podmínka

$$\begin{aligned}\text{Precede}([&i_{c_1}, \ldots, i_{c_n},\\ &i_{c'_1}, \ldots, i_{c'_{n'}}],\\ &[1, \ldots, \min\{m + n + n', n_S\} - 1]),\end{aligned}$$

kde $m = 0$, pokud $n > 0$, jinak $m = 1$. Pro buňky $c''_1, \ldots, c''_{n''}$ s maximálním argumentem 1 se přidá podmínka

$$\begin{aligned}\text{Precede}([&i_{c_1}, \ldots, i_{c_n},\\ &i_{c'_1}, \ldots, i_{c'_{n'}},\\ &i_{c''_1}, \ldots, i_{c''_{n''}}],\\ &[2, \ldots, \min\{m + n + n' + n'', n_S\} - 1]).\end{aligned}$$

Pro buňky, kde je maximální argument z D_S vyšší, se postupuje analogicky. Podmínka Lower-Eq se používá pro omezování velikosti domén.

4.1.6 Redundantní podmínky pro Gecode

Abychom posílili propagaci řešiče Gecode, přidáme redundantní globální podmínky. Mezi vstupními klauzulemi a klauzulemi z dokazovače E budeme hledat axiomy grup, kvazigrup a involutorních funkcí. Tabulky násobení v grupách a kvazigrupách jsou latinské čtverce, pro každý řádek a sloupec takové tabulky přidáme jednu podmínku All-Different, jenž vynutí, že daný řádek nebo sloupec obsahuje každou hodnotu právě jednou. Tabulky involutorních funkcí obsahují každou hodnotu právě jednou, pro každou involutorní funkci přidáme jednu podmínku All-Different.

4.1.7 Hledání všech neizomorfních modelů

V této sekci popíšeme, jak najít více modelů a jak odfiltrovat izomorfní modely. Řešič Gecode prozkoumává ohodnocení proměnných systematicky a snadno tak najde všechna ohodnocení splňující zadané podmínky. U SAT řešičů je situace obvykle odlišná, proto po nalezení každého modelu přidáme klauzuli, která zakáže ohodnocení výrokových proměnných, jenž vedou na daný model. Program Crossbow přidává klauzuli, jenž pro každou funkci z modelu a každou její buňku c s hodnotou v obsahuje literál $\neg A_{c=v}$ a pro každou relaci z modelu a každou její buňku s hodnotou v obsahuje buňku $A_{c=1}$, pokud $v = 0$, nebo $\neg A_{c=1}$, pokud $v = 1$.

Nyní dokážeme postupně generovat různé modely. LNH v obecném případě však nezabrání situaci, že vygenerovaný model je izomorfní jinému modelu, který byl vygenerován dříve. Abychom tyto izomorfní modely odstranili, mohli bychom pro každý nalezený model zkusit najít izomorfismus mezi právě nalezeným modelem a dříve nalezenými modely. Nevýhoda tohoto řešení spočívá v tom, že s rostoucím počtem nalezených modelů bude růst i čas potřebný na nalezení izomorfismu.

Jiné řešení je použít zobrazení kanonických reprezentantů pro číselné interpretace. Pomocí něj každý číselný model převedeme na kanonický číselný model. Díky tomu, že kanonické číselné modely jsou izomorfní právě tehdy, když se rovnají, můžeme velmi rychle detekovat, zda je nově nalezený model izomorfní již dříve nalezenému modelu.

K implementaci zobrazení kanonických reprezentantů pro číselné interpretace využijeme implementaci zobrazení kanonických reprezentantů pro orientované barevné grafy (dále jen grafy). Číselnou interpretaci převedeme na graf, ke grafu najdeme kanonický graf a na základě kanonického grafu zkonstruujeme kanonickou číselnou interpretaci. K nalezení kanonického grafu využívá program Crossbow knihovnu bliss [17].

Konstrukce grafu z interpretace:

- Pro každou doménu D_S vezmeme dosud nepoužitou barvu i_{D_S} a pro každý prvek domény $v \in D_S$ přidáme vrchol u_v barvy i_{D_S}.

- Pro každou relaci $P^{\mathcal{I}}$, kde P je arity $\langle S_1, \ldots, S_n \rangle$ a $n > 0$, vezmeme n dosud nepoužitých barev, označme je $i_1, \ldots, i_n$. Pro každou buňku $P(v_1, \ldots, v_n)$ relace $P^{\mathcal{I}}$ s hodnotou 1 přidáme nové vrcholy $u_1^P, \ldots, u_n^P$, kde vrchol u_k^P má barvu i_k pro $k \in \{1, \ldots, n\}$. Nově přidané vrcholy propojíme hranami (u_{k-1}^P, u_k^P) pro $k \in \{2, \ldots, n\}$. Dále propojíme hranami vrcholy pro argumenty s jejich hodnotami, tj. pro každé $k \in \{1, \ldots, n\}$ přidáme hranu (u_k^P, u_{v_k}), kde u_{v_k} je vrchol pro hodnotu v_k z domény D_{S_k}.

- Pro každou funkci $f^{\mathcal{I}}$, kde f je arity $\langle S_1, \ldots, S_n, S_0 \rangle$, vezmeme $n+1$ dosud nepoužitých barev, označme je $i_0, \ldots, i_n$. Pro každou buňku $f(v_1, \ldots, v_n)$ funkce $f^{\mathcal{I}}$ s hodnotou v_0 přidáme nové vrcholy $u_0^f, \ldots, u_n^f$, kde vrchol u_k^f má barvu i_k pro $k \in \{0, \ldots, n\}$. Nově přidané vrcholy propojíme hranami (u_{k-1}^f, u_k^f) pro $k \in \{1, \ldots, n\}$. Dále propojíme hranami vrcholy pro argumenty a vrchol pro hodnotu buňky s jejich hodnotami, tj. pro každé $k \in \{0, \ldots, n\}$ přidáme hranu (u_k^f, u_{v_k}), kde u_{v_k} je vrchol pro hodnotu v_k z domény D_{S_k}.

Jsou-li dva modely izomorfní, pak[2] grafy zkonstruované pro tyto modely výše popsaným způsobem budou rovněž izomorfní.

Společně s kanonickým grafem bliss vydá i bijekci vrcholů, jenž dokazuje, že grafy jsou skutečně izomorfní. Restrikce této bijekce na vrcholy pro prvky domén určuje bijekci na prvcích domén. S pomocí této bijekce můžeme přejmenovat prvky domén, čímž dostaneme kanonický model.

4.1.8 SAT řešič s podporou dalších podmínek

Další možností, jak snížit počet proměnných nebo počet klauzulí nebo zlepšit propagaci, je rozšířit SAT řešič o další typy podmínek. Do SAT řešiče MiniSat 2.2 [11] jsme přidali speciální typ výrokových klauzulí, které jsou splněny právě tehdy, když obsahují právě jeden splněný literál. Nový řešič se jmenuje Josat. Díky speciálním klauzulím není třeba přidávat klauzule z bodu 1b) sekce 3.1.1.

Aktuální implementace nedovoluje uvnitř speciálních klauzulí užít negaci a každá výroková proměnná se může vyskytovat nejvýše v jedné speciální klauzuli.

4.2 Výběr programovacího jazyka

Při výběru programovacího jazyka byly klíčové následující vlastnosti: dostatečně velká základna uživatelů, automatická správa paměti, algebraické datové typy, pattern matching, bohatý statický typový systém s typovou inferencí.

Dostatečně velká základna uživatelů bývá zárukou rozvoje jazyka a nástrojů s jazykem spjatých. Uživatelé jazyka rovněž vytváří tlak na zachování zpětné kompatibility.

Automatická správa paměti je důležitá pro modularitu programu a usnadňuje sdílení struktur. Například i dokazovač E, jenž je napsaný v jazyce C, implementuje kvůli sdílení termů jednoduchý garbage collector typu Mark & Sweep.

Algebraické datové typy a pattern matching zlepšují čitelnost symbolických výpočtů. V hledači modelů Crossbow se zlepšení čitelnosti týká například kódu, jenž hledá axiomy komutativity, grup, kvazigrup a involutorních funkcí, nebo kódu, jenž transformuje klauzule.

Bohatý statický typový systém eliminuje některé chyby a typové anotace slouží jako dokumentace kódu. Na druhé straně nutnost psát typové anotace u každé lokální proměnné nebo u každého argumentu funkce zhoršuje čitelnost kódu, proto je požadována typová inference.

Výše zmíněné vlastnosti mají programovací jazyky Haskell, OCaml, F# a Scala. Jelikož preferujeme striktní vyhodnocování a jazyky s vedlejšími efekty, vyřadíme Haskell. Jelikož Scala nemá tak dobrou typovou inferenci jako OCaml nebo F#, vyřadíme i Scalu.

Zbývají tedy jazyky OCaml a F#. V obou jazycích jsme vytvořili prototyp hledače modelů a na základě prototypů jsme vybrali OCaml. Prototyp v OCamlu byl pod Linuxem rychlejší než prototyp v F#. Nemalou výhodou OCamlu oproti F# je systém modulů, kde se signatury porovnávají strukturálně, zatímco F# porovnává rozhraní nominálně. Strukturální porovnávání představuje velký přínos pro modularitu programu.

[2]Nejedná se o ekvivalenci. Může nastat situace, že grafy budou izomorfní i pro dva neizomorfní modely.

4.3 Architektura programu

Program Crossbow se skládá ze tří částí. První částí je knihovna tptp, která slouží pro načítání formulí logiky prvního řádu z formátu TPTP [29] a ukládání formulí logiky prvního řádu do formátu TPTP. Tuto knihovnu program Crossbow používá k načítání vstupu a k ukládání výstupu. Knihovnu tptp lze použít i mimo program Crossbow – není na něm nijak závislá.

Další částí je knihovna earray, jenž obsahuje pole s oprávněními. Na rozdíl od klasických polí z OCamlu, která jde měnit vždy, pole s oprávněními mohou být pouze pro čtení. Díky tomu můžeme pole s oprávněními použít pro reprezentaci neměnných struktur, například termů nebo atomů. Jelikož pole s oprávněními nejsou klasická pole, nefunguje na nich pattern matching. Tuto nepříjemnost knihovna earray řeší pomocí syntaktického rozšíření (PPX), které umožní použít syntax pattern matchingu pro klasická pole i pro pole s oprávněními.

Poslední částí je samotný hledač modelů. Ten ke své činnosti využívá SAT řešiče CryptoMiniSat [26], MiniSat [11] a Josat, řešič omezujících podmínek Gecode [14], dokazovač E [24] pro klauzifikaci a pro generování redundantních klauzulí a knihovnu bliss [17] pro hledání kanonických grafů.

Před spuštěním SAT řešiče resp. řešiče omezujících podmínek vykoná program Crossbow standardně následující akce:

- Klauzifikuje vstup, pokud na vstupu nejsou klauzule.
- Vygeneruje redundantní klauzule pomocí dokazovače E.
- Ve vstupních klauzulích a v klauzulích z dokazovače E najde axiomy komutativity. V případě, že se bude spouštět řešič omezujících podmínek, jsou navíc hledány i axiomy grup, kvazigrup a involutorních funkcí.
- Ke vstupním klauzulím přidá malé klauzule z dokazovače E.
- Provede odzplošťování.
- Definuje termy.
- Provede zplošťování, pokud se bude spouštět SAT řešič.
- Rozdělí klauzule.
- Inferuje sorty.
- Vygeneruje výrokové klauzule (včetně klauzulí pro LNH) resp. podmínky (včetně podmínek pro LNH).

SAT řešiče resp. řešiče omezujících podmínek lze snadno přidávat, stačí implementovat modul se signaturou `Sat_inst.Solver` resp. `Csp_solver.S`. S pomocí tohoto modulu vytvoří funktor `Sat_inst.Make` resp. `Csp_inst.Make` modul, který generuje výrokové klauzule resp. omezující podmínky. Důvodem, proč se používá jeden funktor pro SAT řešiče a jiný funktor pro řešiče omezujících podmínek, je podpora pro opakované použití výrokových klauzulí ve funktoru pro SAT řešiče a rozdílná implementace LNH.

KAPITOLA 5

Experimenty

V této kapitole srovnáme program Crossbow s jinými programy pro hledání modelů. Srovnání budeme provádět na splnitelných problémech s `cnf` formulemi z kolekce TPTP 6.1.0 [27].

Programy budeme spouštět na počítači s procesorem Intel Core i5-3570, operačním systémem openSUSE 13.1 a verzí Linuxu 4.1.2. Na vyřešení jednoho problému bude mít každý program k dispozici 11 GiB RAM a 2 minuty procesorového času[1].

Program Crossbow budeme srovnávat s následujícími programy:

- Mace4[2] [21],
- Paradox [8] (program byl upraven, aby ho bylo možné přeložit GHC 7.6.3),
- iProver 1.0 [18].

Ke kompilaci programů použijeme kompilátory GCC 4.8.1, OCaml 4.02.2 a GHC 7.6.3 s knihovnami z Haskell Platform 2013.2.

Pro překlad programu Crossbow použijeme následující programy a knihovny (dostupné z repozitáře OPAM [23]):

- batteries 2.3.1,
- cmdliner 0.9.7,
- menhir 20141215,
- ocamlfind 1.5.5,
- oclock 0.4.0,
- omake 0.9.8.6-0.rc1,
- ounit 2.0.0,

[1] Procesorový čas na jednotlivých jádrech se sčítá. To znamená, že program může vyčerpat časový limit například i za jednu minutu, pokud bude po dobu jedné minuty používat dvě jádra procesoru.

[2] Program Mace4 neumí hledat konečné modely s velikostí domény 1.

- pprint 20140424,
- ppx_tools 0.99.2,
- sexplib 112.24.01,
- sqlite 3.2.0.9,
- tptp 0.3.1,
- zarith 1.3.

Bohužel v našem srovnání nemáme hledače modelů založené na metodách Model Evolution a SGGS [7]. Implementace E-Darwin 1.4 [3] a MELIA 0.1.3 [4] metody Model Evolution se nám totiž nepodařilo zprovoznit a o žádné implementaci SGGS nevíme.

Nyní budeme prezentovat výsledky našeho srovnání. Začneme tabulkou, jenž obsahuje počty vyřešených problémů. V prvním sloupci tabulky jsou názvy kategorií problémů z kolekce TPTP. Kategorie, kde bylo ostře méně než 10 splnitelných problémů, byly sloučeny pod položku „Ostatní“. První řádek tabulky obsahuje názvy konfigurací[3] hledačů modelů. Pro cizí programy jsou názvy konfigurací:

- Mace. Program Mace4.
- Paradox. Program Paradox.
- iProver. Program iProver, hledání modelů.
- iProver/Fin. Program iProver, hledání konečných modelů.

Ostatní názvy konfigurací značí program Crossbow. Použitý řešič se pozná z prefixu názvu konfigurace (CMSat značí CryptoMiniSat). Pokud název konfigurace končí řetězcem „+E“, znamená to, že byl po dobu 5 sekund spuštěn dokazovač E za účelem generování redundantních klauzulí. V opačném případě nebyl dokazovač E spuštěn vůbec.

[3]Soubor `mereni/run_all_provers` na přiloženém DVD obsahuje přesné parametry, s nimiž byly hledače modelů spouštěny.

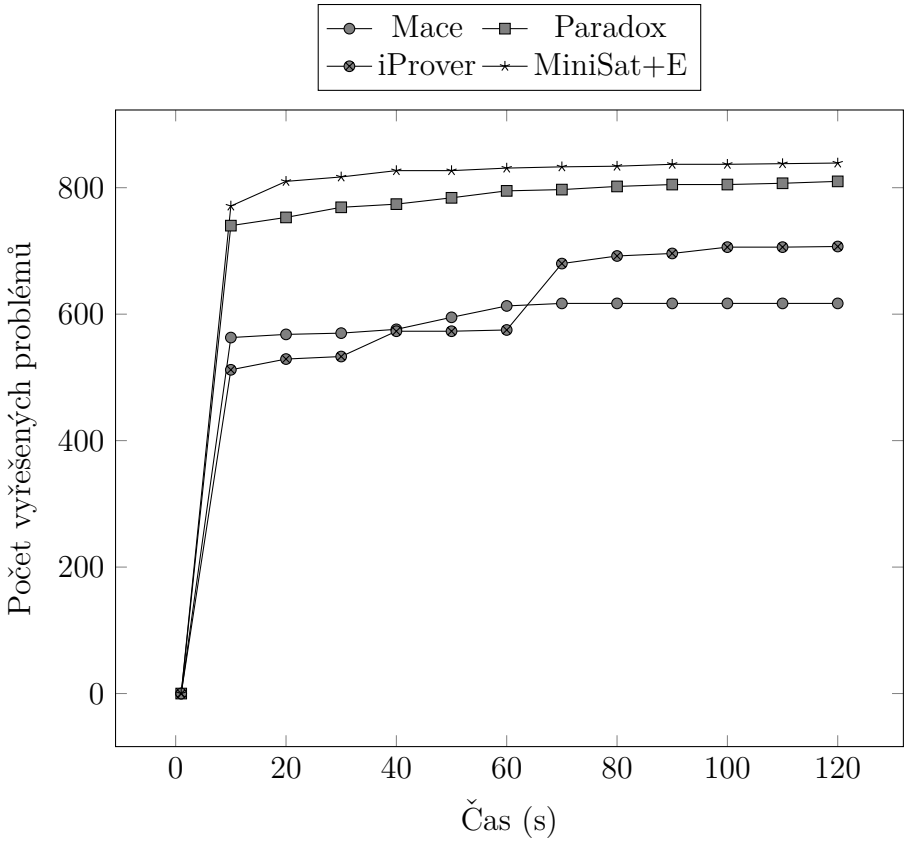
Tabulka s počty vyřešených problémů:

	Počet problémů	Mace	Paradox	iProver	iProver/Fin	CMSat	CMSat+E	MiniSat	MiniSat+E	Josat	Josat+E	Gecode	Gecode+E
ALG	37	14	25	1	1	**30**	**30**	**30**	**30**	**30**	**30**	1	1
BOO	21	14	16	10	10	**17**	**17**	**17**	**17**	**17**	**17**	16	15
CAT	10	8	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**
GEO	18	0	**7**	4	**7**	**7**	**7**	**7**	**7**	**7**	**7**	1	1
GRP	85	72	78	71	72	**80**	**80**	**80**	**80**	**80**	79	77	77
HWV	41	12	21	14	12	27	27	**28**	**28**	27	**28**	19	19
KRS	13	**8**	**8**	**8**	**8**	**8**	**8**	**8**	**8**	**8**	**8**	**8**	**8**
LAT	62	**62**	54	13	14	61	61	61	61	61	61	54	50
LCL	44	23	**43**	23	23	**43**	**43**	**43**	**43**	**43**	**43**	23	25
LDA	26	24	21	0	0	**26**	**26**	**26**	25	**26**	25	0	0
MGT	11	10	**11**	**11**	**11**	**11**	**11**	**11**	**11**	**11**	**11**	**11**	**11**
NLP	236	158	209	**230**	209	222	222	223	223	223	223	176	180
NUM	15	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**
PUZ	27	17	22	20	12	**23**	**23**	22	**23**	22	**23**	18	18
RNG	14	8	**10**	8	8	**10**	**10**	**10**	**10**	**10**	**10**	8	8
SET	30	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**
SWW	39	4	23	11	11	**24**	**24**	23	**24**	**24**	23	14	14
SYN	223	142	195	**221**	220	182	182	182	182	182	182	135	137
TOP	19	8	**19**	**19**	**19**	**19**	**19**	**19**	**19**	**19**	**19**	4	8
Ostatní	44	23	**28**	23	21	**28**	**28**	**28**	**28**	**28**	**28**	23	23
Celkem	1015	617	810	707	678	838	838	838	**839**	838	837	608	615

Jak je vidět, nejlépe si vedl program Crossbow s řešičem MiniSat a se zapnutým generováním redundantních klauzulí. Všimněme si, že program Crossbow výrazněji zaostává pouze na problémech z kategorií NLP a SYN. Důvodem je velikost explicitně reprezentovaných modelů. Modely z těchto kategorií mají při vypsání programem Crossbow do formátu TPTP velikost i několik gigabajtů. Například Josat našel v kategorii NLP více modelů než iProver, jenže modely problémů 75, 82-93 nestihl celé vypsat. Tyto problémy také ukazují výhody řešiče Josat – řešiči MiniSat došla paměť, zatímco řešič Josat dokázal modely najít. Domníváme se, že výhody řešiče Josat by se projevily ještě více, kdyby modely problémů měly větší domény.

Zklamáním je naopak řešič Gecode. Je pomalejší a spotřebovává více paměti než SAT řešiče. Důvodem jsou pravděpodobně použité podmínky – SAT řešiče pracují s klauzulemi efektivněji než Gecode.

Nyní se podíváme na graf ukazující vývoj počtu vyřešených problémů v čase:



Z grafu je patrné, že iProver po jedné minutě zkouší jinou metodu. Od osmdesáté sekundy se počet vyřešených problémů mění pouze velmi mírně – nezdá se, že by zvýšení časového limitu dramaticky změnilo výsledky.

Gecode podporuje jak ploché, tak i neploché klauzule. Následující tabulka ukazuje, že kvůli kategorii LAT[4] je lepší nezplošťovat:

	Počet problémů	Gecode+E	Gecode-F+E
ALG	37	1	**5**
BOO	21	15	**16**
CAT	10	**10**	**10**
GEO	18	**1**	**1**
GRP	85	**77**	76
HWV	41	**19**	18
KRS	13	**8**	**8**
LAT	62	**50**	34
LCL	44	25	**26**
LDA	26	**0**	**0**
MGT	11	**11**	10
NLP	236	**180**	179
NUM	15	**5**	**5**
PUZ	27	**18**	12
RNG	14	**8**	**8**
SET	30	**5**	**5**
SWW	39	**14**	13
SYN	223	**137**	**137**
TOP	19	**8**	**8**
Ostatní	44	**23**	**23**
Celkem	1015	**615**	594

Gecode+E označuje konfiguraci bez zplošťování, Gecode-F+E označuje konfiguraci se zplošťováním.

[4]Důvodem jsou klauzule, jejichž zploštění způsobí výrazný nárůst počtu proměnných.

Některé modifikace lze v programu Crossbow deaktivovat. Díky tomu můžeme zkoumat, jaký přínos tyto modifikace mají. Zde jsou výsledky pro definice termů:

	Počet problémů	MiniSat+E	MiniSat-NDef+E	Gecode+E	Gecode-NDef+E
ALG	37	**30**	**30**	1	1
BOO	21	**17**	**17**	15	15
CAT	10	**10**	**10**	**10**	**10**
GEO	18	**7**	**7**	1	1
GRP	85	**80**	79	77	74
HWV	41	**28**	**28**	19	19
KRS	13	**8**	**8**	**8**	**8**
LAT	62	**61**	54	50	50
LCL	44	**43**	**43**	25	25
LDA	26	**25**	24	0	0
MGT	11	**11**	**11**	**11**	10
NLP	236	**223**	**223**	180	180
NUM	15	**5**	**5**	**5**	**5**
PUZ	27	**23**	**23**	18	18
RNG	14	**10**	**10**	8	**10**
SET	30	**5**	**5**	**5**	**5**
SWW	39	**24**	21	14	13
SYN	223	**182**	**182**	137	137
TOP	19	**19**	**19**	8	3
Ostatní	44	**28**	**28**	23	23
Celkem	1015	**839**	827	615	607

Řetězec NDef v názvu konfigurace značí, že definice termů nebyly použity. Pro řešič MiniSat je největší přínos v kategorii LAT, důvodem jsou opět klauzule, jejichž zploštění způsobí výrazný nárůst počtu proměnných. Díky opakovanému používání pomocných proměnných zavedených pro podmínky Linear, Element, Eq a faktu, že Gecode+E nezplošťuje, nemá řešič Gecode s kategorií LAT obtíže, i když jsou definice termů vypnuté.

Ukazuje se, že detekce axiomů komutativity, grup, kvazigrup a involutorních funkcí nemá prakticky žádný přínos:

	Počet problémů	MiniSat+E	MiniSat-NDet+E	Gecode+E	Gecode-NDet+E
ALG	37	**30**	**30**	1	1
BOO	21	**17**	**17**	15	15
CAT	10	**10**	**10**	**10**	**10**
GEO	18	**7**	**7**	1	1
GRP	85	**80**	**80**	77	76
HWV	41	**28**	**28**	19	19
KRS	13	**8**	**8**	**8**	**8**
LAT	62	**61**	**61**	50	50
LCL	44	**43**	**43**	25	25
LDA	26	**25**	**25**	0	0
MGT	11	**11**	**11**	**11**	**11**
NLP	236	**223**	**223**	180	180
NUM	15	**5**	**5**	**5**	**5**
PUZ	27	**23**	**23**	18	18
RNG	14	**10**	**10**	8	8
SET	30	**5**	**5**	**5**	**5**
SWW	39	**24**	**24**	14	14
SYN	223	**182**	**182**	137	137
TOP	19	**19**	**19**	8	8
Ostatní	44	**28**	**28**	23	23
Celkem	1015	**839**	**839**	615	614

Řetězec NDet v názvu konfigurace značí, že detekce axiomů nebyla použita.

Odzplošťování nemá na problémy z kolekce TPTP vůbec žádný efekt:

	Počet problémů	MiniSat+E	MiniSat-NU+E	Gecode+E	Gecode-NU+E
ALG	37	**30**	**30**	1	1
BOO	21	**17**	**17**	15	15
CAT	10	**10**	**10**	**10**	**10**
GEO	18	**7**	**7**	1	1
GRP	85	**80**	**80**	77	77
HWV	41	**28**	**28**	19	19
KRS	13	**8**	**8**	**8**	**8**
LAT	62	**61**	**61**	50	50
LCL	44	**43**	**43**	25	25
LDA	26	**25**	**25**	0	0
MGT	11	**11**	**11**	**11**	**11**
NLP	236	**223**	**223**	180	180
NUM	15	**5**	**5**	**5**	**5**
PUZ	27	**23**	**23**	18	18
RNG	14	**10**	**10**	8	8
SET	30	**5**	**5**	**5**	**5**
SWW	39	**24**	**24**	14	14
SYN	223	**182**	**182**	137	137
TOP	19	**19**	**19**	8	8
Ostatní	44	**28**	**28**	23	23
Celkem	1015	**839**	**839**	615	615

Řetězec NU v názvu konfigurace značí, že odzplošťování nebylo použito.

Na rozdíl od předešlých dvou modifikací je přínos rozdělování klauzulí značný:

	Počet problémů	MiniSat+E	MiniSat-NS+E	Gecode+E	Gecode-NS+E
ALG	37	**30**	1	1	1
BOO	21	**17**	16	15	15
CAT	10	**10**	**10**	**10**	**10**
GEO	18	**7**	**7**	1	1
GRP	85	**80**	74	77	77
HWV	41	**28**	**28**	19	19
KRS	13	**8**	**8**	**8**	**8**
LAT	62	**61**	23	50	50
LCL	44	**43**	36	25	25
LDA	26	**25**	0	0	0
MGT	11	**11**	**11**	**11**	**11**
NLP	236	**223**	187	180	150
NUM	15	**5**	**5**	**5**	**5**
PUZ	27	**23**	16	18	18
RNG	14	**10**	8	8	8
SET	30	**5**	**5**	**5**	**5**
SWW	39	**24**	14	14	14
SYN	223	**182**	**182**	137	136
TOP	19	**19**	**19**	8	8
Ostatní	44	**28**	**28**	23	23
Celkem	1015	**839**	678	615	584

Řetězec NS v názvu konfigurace značí, že rozdělování klauzulí nebylo použito. Jelikož řešiči Gecode dáváme neploché klauzule, není přínos rozdělování klauzulí tak výrazný.

Nakonec se podíváme, jaký přínos má inference sort:

	Počet problémů	MiniSat+E	MiniSat-NSI+E	Gecode+E	Gecode-NSI+E
ALG	37	**30**	**30**	1	1
BOO	21	**17**	**17**	15	15
CAT	10	**10**	**10**	**10**	**10**
GEO	18	**7**	**7**	1	1
GRP	85	**80**	79	77	77
HWV	41	**28**	26	19	17
KRS	13	**8**	**8**	**8**	**8**
LAT	62	**61**	**61**	50	50
LCL	44	**43**	**43**	25	25
LDA	26	**25**	**25**	0	0
MGT	11	**11**	**11**	**11**	10
NLP	236	**223**	217	180	121
NUM	15	**5**	**5**	**5**	**5**
PUZ	27	**23**	22	18	17
RNG	14	**10**	**10**	8	8
SET	30	**5**	**5**	**5**	**5**
SWW	39	**24**	23	14	13
SYN	223	**182**	**182**	137	136
TOP	19	**19**	**19**	8	7
Ostatní	44	**28**	**28**	23	23
Celkem	1015	**839**	828	615	549

Řetězec NSI v názvu konfigurace značí, že inference sort nebyla použita. Inference sort má výrazný přínos pouze v kategorii NLP pro řešič Gecode. Bohužel neznáme přesné důvody[5], proč tomu tak je.

[5]Jelikož problémy z kategorie NLP obsahují značné množství funkčních symbolů, je možným důvodem velikost podmínek přidaných LNH pro Gecode. S větším množstvím sort (u některých problémů z kategorie NLP lze inferovat stovky sort) klesá velikost těchto podmínek.

KAPITOLA 6

Závěr

V této práci jsme vytvořili hledač konečných modelů Crossbow, který dokáže hledat jeden konečný model nebo všechny navzájem neizomorfní konečné modely dané velikosti. Implementovali jsme metodu MACE, některé její známé modifikace a také několik nových modifikací, jenž, pokud je nám známo, dosud nikdo nezkoušel. Domníváme se tedy, že zadání diplomové práce bylo splněno.

Ve srovnání s ostatními programy pro hledání modelů si program Crossbow nevede špatně. Experimenty z minulé kapitoly však naznačují, že program implementující metodu z dokazovače iProver společně s metodou MACE by si mohl vést ještě lépe[1].

Řada modifikací metody MACE neměla v našem experimentálním srovnání prakticky žádný efekt. To se týká nejen nově představených modifikací jako odzplošťování a přidávání redundantních klauzulí, ale i dříve známých modifikací jako definic termů nebo inference sort (inference sort výrazně pomohla pouze řešiči Gecode v kategorii NLP). Naopak velký přínos měla již dříve známá modifikace rozdělování klauzulí.

I přesto, že jsme se při návrhu kódování klauzulí do podmínek řešiče Gecode snažili omezit množství a velikost podmínek, spotřebovával řešič Gecode více paměti než SAT řešiče. Bylo by tedy vhodné přijít s kódováním, jenž je paměťově šetrnější. Kromě problémů se spotřebou paměti byl řešič Gecode obvykle i pomalejší než SAT řešiče. K vyšší rychlosti by možná pomohla v řešiči Gecode vestavěná redukce symetrií LDSB [22]. Bohužel se nám nepodařilo vymyslet způsob, jak ji použít, aniž by došlo k nárůstu množství proměnných a podmínek.

Při vývoji programu Crossbow jsme nepočítali s tím, že některé modely budou mít po uložení do formátu TPTP velikost několik gigabajtů. To se projevilo zejména v kategorii NLP, kde program Crossbow některé modely našel, ale nestihl je v daném časovém limitu celé vypsat. Tuto nepříjemnost můžeme napravit tak, že do souborů s modely přestaneme ukládat hodnoty buněk, jenž lze odvodit propagací.

Kromě metody MACE jsme věnovali značnou pozornost i metodě SEM. Její implementace, program Mace4, si však v našem experimentálním srovnání příliš dobře nevedla. Podle našeho názoru je hlavním důvodem absence učení klauzulí. Námětem na další práci je tedy implementace metody SEM s lepší propagací

[1] K volbě vhodné metody pro daný problém lze použít strojové učení.

pro SEM, s učením klauzulí a případně i se dvěma sledovanými literály. Takto implementovanou metodu SEM můžeme chápat jako krok mezi DPLL s učením klauzulí a metodami Model Evolution a SGGS.

Seznam použité literatury

1 AUDEMARD, Gilles; BENHAMOU, Belaid. Symmetry in finite model of first order logic. In. *Workshop on Symmetry and Constraint Satisfaction Problems – Affiliated to CP(SymCon)*. 2001, s. 01–08.

2 AUDEMARD, Gilles; HENOCQUE, Laurent. The eXtended Least Number Heuristic. In GORÉ, Rajeev; LEITSCH, Alexander; NIPKOW, Tobias (ed.). *Automated Reasoning*. 2001, s. 427–442. Lecture Notes in Computer Science. ISBN 978-3-54042-254-9.

3 BAUMGARTNER, Peter. *E-Darwin - A Theorem Prover for the Model Evolution Calculus with Equality* [online]. [cit. 2015-07-21]. Dostupný z WWW: ⟨`http://userpages.uni-koblenz.de/~bpelzer/edarwin/`⟩.

4 BAUMGARTNER, Peter. *MELIA* [online]. [cit. 2015-07-21]. Dostupný z WWW: ⟨`http://users.cecs.anu.edu.au/~baumgart/systems/MELIA/index.html`⟩.

5 BAUMGARTNER, Peter; FUCHS, Alexander; NIVELLE, Hans de; TINELLI, Cesare. Computing Finite Models by Reduction to Function-Free Clause Logic. *Journal of Applied Logic*. 2007, vol. 2007.

6 BAUMGARTNER, Peter; TINELLI, Cesare. The Model Evolution Calculus. In BAADER, Franz (ed.). *Automated Deduction – CADE-19*. 2003, s. 350–364. Lecture Notes in Computer Science. ISBN 978-3-54040-559-7.

7 BONACINA, Maria P.; FURBACH, U.; SOFRONIE-STOKKERMANS, V. On First-Order Model-Based Reasoning. *CoRR*. 2015, vol. abs/1502.02535.

8 CLAESSEN, Koen; SÖRENSSON, Niklas. *Paradox* [online]. [cit. 2012-11-11]. Dostupný z WWW: ⟨`http://www.cse.chalmers.se/~koen/code/folkung.tar.gz`⟩.

9 CLAESSEN, Koen; SÖRENSSON, Niklas. New Techniques that Improve MACE-style Finite Model Finding. In. *Proceedings of the CADE-19 Workshop: Model Computation - Principles, Algorithms, Applications*. 2003.

10 DECHTER, R. *Constraint Processing*. 2003. The Morgan Kaufmann Series in Artificial Intelligence Series. ISBN 978-1-55860-890-0.

11 EEN, Niklas; SORENSSON, Niklas. *MiniSat 2.2* [online]. [cit. 2012-11-19]. Commit cd3a2d653fc073585ef05e2ebb72ab015d851279. Dostupný z WWW: ⟨`https://github.com/niklasso/minisat`⟩.

12 ENDERTON, H. B. *A Mathematical Introduction to Logic*. 2001. ISBN 978-0-12238-452-3.

13 GAVANELLI, Marco. The Log-support Encoding of CSP into SAT. In. *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*. Providence, RI, USA: Springer-Verlag, 2007, s. 815–822. CP'07. ISBN 978-3-54074-969-1.

14 GECODE TEAM. *Gecode* [online]. [cit. 2015-04-12]. Dostupný z WWW: ⟨`http://www.gecode.org/`⟩.

15 HILLENBRAND, T.; WEIDENBACH, Christoph. Superposition for Bounded Domains. In BONACINA, Maria Paola; STICKEL, Mark E. (ed.). *Automated Reasoning and Mathematics*. 2013, s. 68–100. Lecture Notes in Computer Science. ISBN 978-3-64236-674-1.

16 JIA, Xiangxue; ZHANG, Jian. A Powerful Technique to Eliminate Isomorphism in Finite Model Search. In FURBACH, Ulrich; SHANKAR, Natarajan (ed.). *Automated Reasoning*. 2006, s. 318–331. Lecture Notes in Computer Science. ISBN 978-3-54037-187-8.

17 JUNTTILA, Tommi; KASKI, Petteri. *bliss: A Tool for Computing Automorphism Groups and Canonical Labelings of Graphs* [online]. [cit. 2015-07-16]. Dostupný z WWW: ⟨`http://www.tcs.hut.fi/Software/bliss/`⟩.

18 KOROVIN, Konstantin. iProver – An Instantiation-Based Theorem Prover for First-Order Logic (System Description). In ARMANDO, Alessandro; BAUMGARTNER, Peter; DOWEK, Gilles (ed.). *Automated Reasoning*. 2008, s. 292–298. Lecture Notes in Computer Science. ISBN 978-3-54071-069-1.

19 KOROVIN, Konstantin. Inst-Gen – A Modular Approach to Instantiation-Based Automated Reasoning. In VORONKOV, Andrei; WEIDENBACH, Christoph (ed.). *Programming Logics*. 2013, s. 239–270. Lecture Notes in Computer Science. ISBN 978-3-64237-650-4.

20 MCCUNE, William. *A Davis-Putnam Program and its Application to Finite First-Order Model Search: Quasigroup Existence Problems*. 1994.

21 MCCUNE, William. Mace4 Reference Manual and Guide. *CoRR*. 2003, vol. cs.SC/0310055.

22 MEARS, Christopher; GARCIA DE LA BANDA, Maria; DEMOEN, Bart; WALLACE, Mark. Lightweight dynamic symmetry breaking. *Constraints*. 2014, vol. 19, no. 3, s. 195–242. ISSN 1383-7133.

23 OCAMLPRO. *OPAM* [online]. [cit. 2015-07-21]. Dostupný z WWW: ⟨`http://opam.ocaml.org/packages/`⟩.

24 SCHULZ, Stephan. System Description: E 1.8. In MCMILLAN, Ken; MIDDELDORP, Aart; VORONKOV, Andrei (ed.). *Logic for Programming, Artificial Intelligence, and Reasoning*. 2013, s. 735–743. Lecture Notes in Computer Science. ISBN 978-3-64245-220-8.

25 SILVA, João P. Marques; SAKALLAH, Karem A. GRASP—A New Search Algorithm for Satisfiability. In. *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*. San Jose, California, USA: IEEE Computer Society, 1996, s. 220–227. ICCAD '96. ISBN 0-8186-7597-7.

26 SOOS, Mate. *CryptoMiniSat SAT solver 4.2.0* [online]. [cit. 2015-04-04]. Commit 9d0f57c745cc731238db394d62048234d3ba917d. Dostupný z WWW: ⟨`https://github.com/msoos/cryptominisat`⟩.

27 SUTCLIFFE, G. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*. 2009, roč. 43, č. 4, s. 337–362.

28 SUTCLIFFE, G.; SUTTNER, C. The State of CASC. *AI Communications*. 2006, vol. 19, no. 1, s. 35–48.

29 SUTCLIFFE, Geoff. *TPTP Syntax 6.1.0.1* [online]. [cit. 2015-07-20]. Dostupný z WWW: ⟨`http://www.cs.miami.edu/~tptp/TPTP/SyntaxBNF.html`⟩.

30 TAMMET, Tanel. Finite model building: improvements and comparisons. In. *In: Model Computation – Principles, Algorithms, Applications, CADE-19 Workshop W4*. 2003.

31 ZHANG, Jian. Constructing finite algebras with FALCON. *Journal of Automated Reasoning*. 1996, vol. 17, no. 1, s. 1–22. ISSN 0168-7433.

32 ZHANG, Jian; ZHANG, Hantao. SEM: A System for Enumerating Models. In. *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 1*. Montreal, Quebec, Canada: Morgan Kaufmann Publishers Inc., 1995, s. 298–303. IJCAI'95. ISBN 978-1-55860-363-9.

PŘÍLOHA A

Obsah DVD

Součástí práce je i DVD obsahující:

- Text této práce ve formátu PDF (soubor `dipl.pdf`).
- Zdrojový kód tohoto textu v LaTeXu (archiv `dipl.zip`).
- Zdrojový kód programu Crossbow (archiv `crossbow.zip`).
- Zdrojový kód knihovny tptp (archiv `tptp.zip`).
- Zdrojový kód dalších programů a knihoven (adresář `dalsi`):
  - archiv `folkung.tar.gz` obsahuje původní verzi programu Paradox staženou ze stránek autora [8],
  - archiv `paradox.zip` obsahuje modifikovanou verzi programu Paradox, kterou lze přeložit s GHC 7.6.3,
  - archiv `opam-mini-repository.tar.gz` obsahuje závislosti potřebné ke kompilaci programu Crossbow a knihovny tptp (doporučujeme však instalovat závislosti z oficiálního repozitáře OPAM [23]),
  - archiv `bow.zip` obsahuje náš prototyp hledače modelů v jazyce F#, ...
- Zkompilované programy (archiv `kompilovane.zip`).
- Problém `model-evolution/prob.p`, na němž nefungují hledače modelů E-Darwin 1.4 a MELIA 0.1.3 implementující metodu Model Evolution. Instrukce pro spouštění hledačů modelů jsou uloženy přímo v problému.
- Programy a skripty pro realizaci experimentů (adresář `mereni`):
  - v podadresáři `problems` jsou seznamy problémů z databáze TPTP, na nichž byly experimenty prováděny,
  - ke spouštění hledačů modelů byl použit skript `run_all_provers` – obsahuje parametry, s nimiž byly hledače modelů spouštěny, ...
- Modely nalezené při experimentech (adresář `modely`).

- Podrobné výsledky experimentů (adresář `vysledky`):
  - Shrnutí ve formátu PDF. Soubory, jejichž název má prefix `sum_times`, obsahují podrobné výsledky pro jednotlivé problémy. Tabulky v těchto souborech obsahují časy v sekundách potřebné na nalezení modelu. Pro nevyřešené problémy je v tabulce jedno z písmen „č“ (došel čas), „p“ (došla paměť), „x“ (jiný problém). Podrobnější informace lze získat v některých PDF prohlížečích umístěním ukazatele myši nad počet sekund nebo písmeno.
  - SQLite 3 databáze `REPORT` obsahuje výsledky, z nichž jsou vygenerována shrnutí ve formátu PDF.
- Problémy použité pro experimenty (archiv `TPTP-v6.1.0.tgz`).