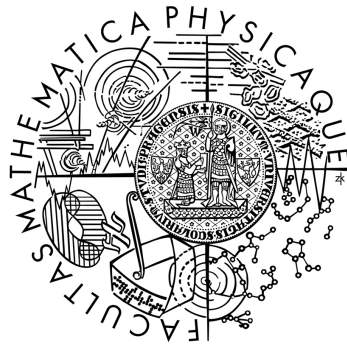


Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Alena Peterová

Konstrukce modelů pomocí CSP

Katedra algebry

Vedoucí bakalářské práce: RNDr. David Stanovský, Ph.D.

Studijní program: Matematika

Studijní obor: obecná matematika

Praha 2011

Děkuji vedoucímu bakalářské práce RNDr. Davidu Stanovskému, Ph.D., za ochotu, s jakou mi udělil mnoho cenných rad a užitečných připomínek. Dále děkuji všem autorům MiniZincu, Gecode a TPTP za možnost volného použití jejich díla.

Prohlašuji, že jsem tuto bakalářskou práci vypracovala 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 Praze dne 25. 5. 2011

Alena Peterová

Název práce: Konstrukce modelů pomocí CSP

Autor: Alena Peterová

Katedra: Katedra algebry

Vedoucí bakalářské práce: RNDr. David Stanovský, Ph.D., Katedra algebry

Abstrakt: V této práci se věnujeme algoritmům na konstrukci konečných modelů pro množiny axiomů logiky 1. řádu s cílem navrhnout a implementovat novou metodu, založenou na převodu na problém splnitelnosti omezení (CSP). V teoretické části představíme standardní metodu MACE, používající převod úloh na SAT, a pokročilejší techniky zvyšující její efektivitu: dělení klauzulí, definici termů a statickou redukci symetrií. Následuje návrh alternativní metody, která podobným způsobem převádí úlohy na CSP. Nově navrhuje techniku redukce symetrií i pro binární funkce. Poté popíšeme implementaci alternativní metody pomocí CSP-modelovacího jazyka MiniZinc a CSP-solveru Gecode. Na závěr porovnáme výkonnost vytvořeného nástroje na hledání modelů s nejuspěšnějšími zástupci standardních metod, systémy Paradox a Mace4.

Klíčová slova: logika 1. řádu, metoda MACE, CSP, MiniZinc, redukce symetrií

Title: Model building using CSP

Author: Alena Peterová

Department: Department of Algebra

Supervisor: RNDr. David Stanovský, Ph.D., Department of Algebra

Abstract: In the present work, we study algorithms for building finite models of sets of first-order axioms with the aim of proposing and implementing a new method, based on translation onto constraint satisfaction problem (CSP). In the theoretical part, we describe the standard MACE-style method, based on translating problems onto SAT, and advanced techniques that improve the effectiveness of this method: clause splitting, term definitions and static symmetry reduction. Next, we propose an alternative method, which translates problems onto CSP in a similar way. In addition, we have newly proposed a static symmetry reduction technique for binary functions. Next, we describe an implementation of the alternative method using a CSP-modelling language MiniZinc and a CSP-solver Gecode. Finally, we compare performance of our model finder against state-of-the-art representatives of standard methods, systems Paradox and Mace4.

Keywords: first-order logic, MACE-style method, CSP, MiniZinc, symmetry reduction

Obsah

Úvod	2
1 Základní pojmy a notace	4
1.1 Predikátová logika	4
1.1.1 Syntaxe	4
1.1.2 Sémantika	5
1.2 Výroková logika	6
1.3 Problém splňování podmínek	7
2 Metoda MACE	8
2.1 Základní myšlenka	8
2.1.1 Zavedení výrokových proměnných	10
2.1.2 Flattening	10
2.1.3 Vytvoření instancí	11
2.2 Srovnání s metodou SEM a LNH	13
2.3 Pokročilé techniky	14
2.3.1 Dělení klauzulí	14
2.3.2 Definice termů	16
2.3.3 Statická redukce symetrií	18
2.3.4 Postupné prohledávání	20
2.3.5 Odvození sortů	20
3 Metoda převodu na CSP	22
3.1 MiniZinc	22
3.1.1 Základy syntaxe	23
3.1.2 Prohledávací anotace	25
3.1.3 FlatZinc	26
3.2 Užití pokročilých technik	27
3.3 Převod do MiniZinc	29
3.4 Gecode	31
3.5 Vytvořený nástroj	31
4 Výsledky	33
4.1 Testování	33
4.2 Přehled výsledků	34
4.3 Komentář výsledků	35
Závěr	37
Seznam použité literatury	38
Seznam www odkazů	40
Seznam použitých zkratek	41
Příloha A	42

Úvod

Tato bakalářská práce se věnuje úloze konstrukce konečných modelů v predikátové logice 1. řádu (FOL – first-order logic). Vstupem typické úlohy je sada axiomů v jazyce FOL. Úkolem je najít nějakou konečnou množinu a na ní interpretaci každé funkce a relace tak, aby splňovaly zadané axiomy. Cílem této bakalářské práce je navrhnout a implementovat novou metodu hledání konečných modelů, která bude spočívat v převodu na problém splňování podmínek (CSP – constraint satisfaction problem) a ve které se použijí některé speciální techniky standardních metod konstrukce konečných modelů.

Problém splňování podmínek (CSP) je obecnější než konstrukce modelů predikátové logiky. CSP úloha se skládá pouze z proměnných, ze seznamu hodnot, které mohou proměnné nabývat, a ze sady podmínek. Vyřešit danou úlohu znamená nalézt nějaké ohodnocení proměnných, které zadané podmínky splňuje. Protože CSP má široké možnosti využití v praxi, existuje a neustále se rozvíjí mnoho systémů, které CSP úlohy umí řešit (tzv. CSP-solverů).

Konstrukce konečných modelů má velký význam v oblasti automatického dokazování vět. Pro mnohé automatické dokazovače je výhodné spojit síly s hledačem konečných modelů. Zatímco dokazovač se snaží odvodit věty ze zadaných axiomů a předpokladů, hledač hledá protipříklady, tj. modely splňující axiomy a negace odvozovaných vět. K úspěšným systémům na tomto poli patří v současné době nástroje W. McCuna, dokazovač Prover9 spolu s hledačem Mace4, nástupci staršího systému Otter s hledačem Mace2. „Metoda MACE“ je (podle nástroje Mace2) název jedné ze standardních metod konstrukce konečných modelů ve FOL, a to konstrukce pomocí převodu na problém splnitelnosti booleovských formulí (SAT – Boolean satisfiability problem). Základy této metody jsou popsány v sekci 2.1.

Mezi nejúspěšnější systémy používající metodu MACE patří Paradox [3]. Mým úkolem bylo prostudovat jeho algoritmus převodu na SAT a navrhnout analogický algoritmus pro převod na CSP. Pokročilými technikami systému Paradox se zabývá sekce 2.3.

Metodu převodu do CSP jsem implementovala pomocí nezávislého CSP mode-

lovacího jazyka MiniZinc [10], kterému se blíže věnuje sekce 3.1. Návrh algoritmu převodu do jazyka MiniZinc popisuje oddíl 3.3. K řešení vzniklých CSP úloh byl použit volně distribuovaný software Gecode, přiblížený v části 3.4.

Testování účinnosti vytvořeného nástroje na hledání konečných modelů bylo provedeno na vybraných úlohách z knihovny TPTP – Thousands of Problems for Theorem Provers. Zaměřila jsem se hlavně na algebraické úlohy. Výsledky testování a srovnání se systémy Mace4 a Paradox obsahuje kapitola 4.

1. Základní pojmy a notace

V této kapitole jsou uvedeny základní pojmy a notace užívané ve zbytku práce.

Množina \mathbb{N} je množina přirozených čísel $\{1, 2, 3, \dots\}$. Prvky množiny \mathbb{N} značíme m, n, k . Množina \mathcal{B} je množina booleovských pravdivostních hodnot $\{\text{false}, \text{true}\}$.

Kartézský součin $\underbrace{D \times \dots \times D}_n$ zapisujeme zkráceně D^n .

1.1 Predikátová logika

Koncepce predikátové logiky 1. řádu (FOL) je převzata z přednášky Výroková a predikátová logika J. Mlčka.

1.1.1 Syntaxe

Jazyk ve FOL je tvořen symboly logickými a mimologickými. Mezi logické symboly patří logické spojky (*konjunkce* \wedge , *disjunkce* \vee , *negace* \neg), univerzální a existenční kvantifikátor, *proměnné* x, y, z, w , *rovnost* $=$ a zkratka \neq zastupující negaci rovnosti. Mimologické symboly jazyka jsou buď funkční, které značíme f, g , nebo relační, které značíme P, Q, R .

Každý relační symbol R a každý funkční symbol f má danou nezápornou celočíselnou *aritu* $ar(R), ar(f)$, která určuje počet jeho argumentů. Je-li arita funkčního symbolu nula, nazýváme ho *konstanta* a značíme a, b, c .

Term t je vytvořený z funkčních symbolů, konstant a proměnných. Term je buď proměnná, nebo nějaký funkční symbol, jehož argumenty jsou termy ($f(t_1, \dots, t_{ar(f)})$). Speciálně samotná konstanta je také term.

Atomická formule je buď rovnost dvou termů ($t_1 = t_2$), nebo relační symbol, jehož argumenty jsou termy ($R(t_1, \dots, t_{ar(R)})$).

Formule φ jsou vystaveny z atomických formulí přidáváním kvantifikátorů či spojováním pomocí logických spojek. Pro speciální typy formulí se zavádí další tři pojmy. *Literál* l je atomická formule nebo její negace. *Klauzule* C je disjunkce konečného počtu literálů, které se pak pro přehlednost označují jako *disjunkty* příslušné klauzule. Dále říkáme, že formule je v konjunktivní normální formě (CNF), pokud je vytvořená jako konjunkce klauzulí.

Potřebujeme-li zdůraznit, jaké proměnné v sobě formule φ obsahuje, píšeme $\varphi(x_1, \dots, x_n)$. Při takovém zápisu je pak již zaručeno, že jiné proměnné než x_1, \dots, x_n se ve formuli φ nevyskytují. Pro přehlednost se někdy množina proměnných $\{x_1, \dots, x_n\}$ označuje pouze jako \bar{x} , neboli $\varphi(\bar{x})$ a $\varphi(x_1, \dots, x_n)$ jsou ekvivalentní zápisy téhož.

Substituovat termy t_1, \dots, t_n za proměnné x_1, \dots, x_n do formule $\varphi(x_1, \dots, x_n)$ znamená nahradit každý výskyt proměnné x_i termem t_i pro každé $i = 1, \dots, n$. Výsledek substituce se symbolicky zapisuje jako $\varphi(t_1, \dots, t_n)$ a nazývá se *instance* formule φ . Podobně je možné ve formuli substituovat term za term. Značení $\varphi[t_1/t_2]$ znamená, že každý výskyt termu t_1 ve formuli φ byl nahrazen termem t_2 . Případně zápis $\varphi[t]$ říká, že se ve formuli φ alespoň jednou vyskytuje term t .

Za *axiom* může být zvolena libovolná formule. Množinu axiomů značíme T .

V dalším textu se již nebudeme zabývat kvantifikátory, neboť v oblasti konstrukce konečných modelů lze každou formuli ekvivalentně přepsat bez kvantifikátorů, pouze za cenu přidání nových konstantních symbolů. Proto nadále považujeme všechny formule za bezkvantifikátorové.

1.1.2 Sémantika

Interpretací každého funkčního a relačního symbolu získáváme *strukturu* \mathcal{S} nad vytvořeným jazykem. Nosnou množinu struktury \mathcal{S} nazýváme *univerzum* a značíme D (podle angl. domain). Na konkrétní podobě prvků univerza nezáleží, jak je uvedeno níže, proto pro konečná univerza velikosti n volíme $D = \{1, \dots, n\}$.

Každý funkční symbol f s aritou n interpretujeme jako *n -ární funkci* $f^{\mathcal{S}}$, $f^{\mathcal{S}} : D^n \rightarrow D$, která každé n -tici argumentů z D přiřazuje právě jednu hodnotu z D . Speciálně každá konstanta odpovídá jedné z hodnot D .

Každý relační symbol R s aritou n interpretujeme jako *n -ární relaci* $R^{\mathcal{S}}$, do které každá n -tice argumentů z D buď náleží, nebo nenáleží. Relaci $R^{\mathcal{S}}$ můžeme chápat jako funkci $R^{\mathcal{S}} : D^n \rightarrow \mathcal{B}$, jejíž hodnoty odpovídají tomu, zda daná n -tice relaci $R^{\mathcal{S}}$ náleží (true), nebo nenáleží (false).

Nepovede-li to k nedorozumění, nebudou v dalším textu výslovně odlišovány relační symboly R a funkční symboly f jazyka od jejich interpretací $R^{\mathcal{S}}, f^{\mathcal{S}}$

v konkrétní struktuře \mathcal{S} . Z kontextu bude patrné, o který objekt se jedná.

Ohodnocení proměnných ve struktuře \mathcal{S} je funkce $e : Var \rightarrow D$ definovaná na množině proměnných Var . *Hodnota* termu t ve struktuře \mathcal{S} při ohodnocení e závisí na hodnotách, které e nabývá na proměnných v termu t , a také na hodnotách funkcí, ze kterých se term t skládá. Podobně *platnost* formule φ ve struktuře \mathcal{S} při ohodnocení e závisí na hodnotách termů obsažených ve φ při ohodnocení e a na interpretaci relací, ze kterých je formule φ složena.

Formule ve struktuře \mathcal{S} *platí*, pokud platí při každém ohodnocení proměnných v \mathcal{S} . Struktura je *modelem* množiny axiomů, pokud v ní každý axiom platí. Axiom či množina axiomů jsou *splnitelné*, pokud existuje nějaký jejich model.

Dvě struktury S_1, S_2 stejného jazyka jsou *izomorfní*, pokud mezi jejich univerzy D_1, D_2 existuje bijekce $\pi : D_1 \leftrightarrow D_2$, pro každý funkční symbol f platí

$$\forall a_1, \dots, a_{ar(f)} \in D : \pi (f^{S_1}(a_1, \dots, a_{ar(f)})) = f^{S_2} (\pi(a_1), \dots, \pi(a_{ar(f)}))$$

a podobně pro každý relační symbol R platí

$$\forall a_1, \dots, a_{ar(R)} \in D : R^{S_1}(a_1, \dots, a_{ar(R)}) \Leftrightarrow R^{S_2} (\pi(a_1), \dots, \pi(a_{ar(R)}))$$

Přímo z definice izomorfismu je zřejmé, že izomorfní struktury mají stejnou velikost univerza a je-li struktura S modelem množiny axiomů T , pak každá struktura S' izomorfní s S je také modelem T . Proto při hledání modelů záleží pouze na velikosti univerza, nikoli na jeho konkrétních prvcích.

1.2 Výroková logika

Výroková logika je fragment predikátové logiky. Jazyk výrokové logiky neobsahuje proměnné, funkční symboly ani relační symboly s kladnou aritou. U struktur ve výrokové logice tedy nemá smysl uvažovat univerzum; jediné, co je třeba interpretovat, jsou nulární relační symboly.

Každá nulární relace odpovídá jedné z hodnot $\{\text{true}, \text{false}\}$. Nulární relace jsou výstižně nazývány *výrokovými proměnnými*. Výrokové formule, nebo také *booleovské formule*, se skládají pouze z výrokových proměnných a logických spojek.

Konstrukce modelu množiny axiomů T tedy spočívá pouze v nalezení nějakého ohodnocení výrokových proměnných, které splňuje všechny axiomy. Existuje-li takové ohodnocení výrokových proměnných, říkáme, že T je *splnitelná*.

1.3 Problém splňování podmínek

Definice problému splňování podmínek (CSP) vychází z přednášky M. Marótiho [4].

Jazyk CSP obsahuje proměnné, které budeme nazývat *rozhodovací proměnné* (angl. decision variables), a podmínky (angl. constraints).

Model CSP úlohy je trojice (V, D, \mathcal{C}) , kde V je konečná množina rozhodovacích proměnných, D je konečná množina hodnot, které mohou proměnné nabývat, a \mathcal{C} je seznam *podmínek*. Každá podmínka z \mathcal{C} je reprezentována dvojicí (S, R) , kde S je n -prvková podmnožina V a R je n -ární relace na D .

Podmínka (S, R) je *splněna* při ohodnocení $e : V \rightarrow D$, pokud $(e(x_1), \dots, e(x_n))$ náleží relaci R .

Zobrazení $e : V \rightarrow D$ je řešení modelu (V, D, \mathcal{C}) , pokud je při něm každá podmínka z \mathcal{C} splněna.

Protože CSP slouží k formulaci úloh, které v sobě obsahují nějaké rozhodování na základě omezujících podmínek, je také SAT speciálním případem CSP. Výrokové proměnné chápeme jako rozhodovací proměnné, množina hodnot pro proměnné je \mathcal{B} a seznam podmínek \mathcal{C} je tvořen dvojicemi $(\{x_1, \dots, x_n\}, \varphi(x_1, \dots, x_n))$, kde φ je booleovská formule ze SAT úlohy.

V CSP lze formulovat mnoho NP-úplných problémů, kromě uvedeného problému SAT také například 3-obarvitelnost grafu [4].

2. Metoda MACE

Existuje více způsobů konstrukce konečných modelů, přičemž pro obecné FOL úlohy se standardně používají dvě metody. *Metoda MACE*, jejíž podstatou je převod na SAT, je podrobně popsána v této kapitole. Mezi její představitele patří například systémy Mace2 a Paradox. *Metoda SEM*, pojmenovaná podle nástroje SEM (System for Enumerating Models) [14], je založena na postupném vyplňování tabulek funkcí a relací. Metodu SEM používá kromě systému SEM i Mace4 [8].

Pro speciální třídy FOL úloh se uplatňují i jiné metody. Například pro úlohy EPR (Effectively Propositional), v nichž formule neobsahují žádné funkční symboly, je nejúčinnější metodou (podle výsledků soutěží CASC) tzv. Inst-Gen kalkulus systému iProver [5].

Nicméně cíl této práce jsou obecné FOL úlohy s konečným modelem a na tomto poli se nejvíce prosazuje metoda MACE. Její výhodou je možnost použití již existujícího nástroje na řešení SAT, tzv. SAT-solveru, přičemž díky velkému vědeckému zájmu o problém SAT je nabídka výkonných SAT-solverů široká. Základní princip metody MACE je popsán v sekci 2.1.

Výčet hlavních nedostatků základní metody MACE, stručné porovnání s metodou SEM a idea tzv. *heuristiky nejmenšího čísla* (LNH – least-number heuristic) jsou obsahem sekce 2.2.

Sekce 2.3 se věnuje pokročilým technikám, které navrhli Claessen a Sörensson při implementaci metody MACE v systému Paradox [3]. Paradox je nejúspěšnější představitel metody MACE, například v letech 2007–2010 se pokaždé stal vítězem soutěže CASC v kategorii hledačů konečných modelů. Paradox využívá efektivní a dobře integrovatelný SAT-solver MiniSat, jehož spoluautorem je právě N. Sörensson.

2.1 Základní myšlenka

Hlavním zdrojem informací o metodě MACE byl dokument [3].

Úkolem je zkonstruovat konečný model množiny FOL axiomů. Velikost hledaného modelu obecně není zadána, ovšem tato metoda s velikostí modelu potřebuje

pracovat. V důsledku se proto místo jedné úlohy řeší zvlášť pro různá $n \in \mathbb{N}$ úlohy, které se skládají z axiomů původní úlohy a z požadavku na velikost univerza. Typicky probíhá hledání modelu od velikosti $n = 2$, v případě nenalezení modelu se vždy n o jedna zvětší. Jakmile je pro nějaké n model nalezen, úloha je vyřešena.

Konstrukce modelu určené velikosti (tj. s určeným univerzem D) pak probíhá ve čtyřech krocích:

1. Množina axiomů FOL úlohy se převede na množinu axiomů v CNF tvaru, která je splnitelná právě tehdy, když je splnitelná původní množina.
2. CNF axiomy se převedou na sadu booleovských axiomů, které tak tvoří novou úlohu ve výrokové logice („SAT úloha“). Přitom SAT úloha je splnitelná, právě když je množina CNF axiomů splnitelná.
3. SAT úloha se vyřeší pomocí SAT-solveru.
4. Je-li SAT úloha splnitelná, z nalezeného ohodnocení výrokových proměnných se rekonstruuje model původní úlohy. Není-li splnitelná, model neexistuje.

Algoritmy na převod obecných FOL formulí do CNF jsou dobře známé, kvalitní výsledky dává například překladač FLOTTER spojený s automatickým dokazovačem SPASS [13]. Vzniklé CNF formule se pak mohou rozdělit na jednotlivé klauzule, kterými se v množině axiomů nahradí. Tím je vytvořena množina klauzulí, jejíž splnitelnost odpovídá splnitelnosti původní množiny FOL axiomů.

Klíčový krok konstrukce je ten druhý – **převod axiomů na SAT**. Poslední dvě fáze konstrukce jsou pak spíše záležitostí konkrétní implementace, takže není třeba se jim podrobněji věnovat.

Jazyk SAT úlohy obsahuje pouze výrokové proměnné. Převod úlohy na SAT proto spočívá v zavedení vhodných výrokových proměnných a následném zakódování axiomů do booleovských formulí. Proces kódování se dále dělí na dvě části; tzv. *flattening* a *vytvoření instancí* (angl. instantiating). Převodu na SAT se věnují následující tři oddíly.

2.1.1 Zavedení výrokových proměnných

Zaváděné výrokové proměnné SAT úlohy se označují tak, aby byla zřejmá jejich vazba s původní FOL úlohou. Ve výrokové logice však jejich označení žádné speciální vlastnosti neimplikuje.

Pro každou relaci R a každý vektor argumentů $(a_1, \dots, a_{ar(R)}) \in D^{ar(R)}$ je zavedena jedna nová výroková proměnná, která bude odpovídat platnosti relace $R(a_1, \dots, a_{ar(R)})$. Tuto výrokovou proměnnou nazveme „ $R(a_1, \dots, a_{ar(R)})$ “.

Pro každou funkci f , každý vektor argumentů $(a_1, \dots, a_{ar(f)}) \in D^{ar(f)}$ a každý prvek $b \in D$ je zavedena nová výroková proměnná, která bude mít hodnotu true právě tehdy, když hodnota $f(a_1, \dots, a_{ar(f)})$ je rovna b . Název této výrokové proměnné je „ $f(a_1, \dots, a_{ar(f)}) = b$ “.

2.1.2 Flattening

Flattening je proces, při kterém se převádějí obecné FOL klauzule na tzv. *ploché* (angl. shallow) klauzule, přičemž množina vzniklých plochých klauzulí je splnitelná, právě když je splnitelná množina původních klauzulí.

Definice. *Literál je plochý, pokud je v některém z těchto tvarů:*

1. $R(x_1, \dots, x_{ar(R)})$ nebo $\neg R(x_1, \dots, x_{ar(R)})$
2. $f(x_1, \dots, x_{ar(f)}) = y$ nebo $f(x_1, \dots, x_{ar(f)}) \neq y$
3. $x = y$

kde x_1, x_2, \dots, x, y jsou proměnné. V opačném případě je literál neplochý.

Definice. *Klauzule je plochá, pokud obsahuje pouze ploché literály.*

Algoritmus 1 (Flattening).

vstup: klauzule C

výstup: plochá klauzule C_P

1. **while** $\exists x, y$ proměnné, $\exists C_0$ klauzule, že C je tvaru $C_0 \vee x \neq y$ **do**
 $C := C_0[y/x]$
2. **while** $\exists t$ term, který není proměnná a vyskytuje se v C **do**
najdi proměnnou x , která se nevyskytuje v C

$$C := C[t/x] \vee t \neq x$$

3. **return** C

Věta 1. *Algoritmus 1 najde pro každou klauzuli C plochou klauzuli C_P , která je splnitelná právě tehdy, když je C splnitelná.*

Důkaz. Z definice plochého literálu plyne, že všechny možné tvary neplochého literálu jsou tyto:

1. $(\neg)R(t_1, \dots, t_{ar(R)})$, kde některý z termů $t_1, \dots, t_{ar(R)}$ není proměnná
2. $(\neg)f(t_1, \dots, t_{ar(f)}) = t_0$, kde některý z termů $t_0, \dots, t_{ar(f)}$ není proměnná
3. $x \neq y$ pro proměnné x, y

Algoritmus 1 v 1. kroku nahrazuje všechny neploché literály tvaru 3 a v 2. kroku všechny neploché literály tvaru 1 a 2. Během 2. kroku se do klauzule žádné nové neploché literály tvaru 3 nepřidají, neboť term t v připojeném literálu $t \neq x$ není proměnná. Protože termy nemohou být donekonečna zanořené, najde algoritmus v konečném čase klauzuli, která obsahuje pouze ploché literály.

Klauzule C tvaru $C_0(x, y, \dots) \vee x \neq y$ platí v každém ohodnocení e s vlastností $e(x) \neq e(y)$, neboť pak je splněn disjunkt $x \neq y$. Klauzule C je tedy splnitelná právě tehdy, když platí $C_0(x, y, \dots)$ pro všechna ohodnocení, která se na proměnných x a y rovnají. Neboli C je splnitelná, právě když $C_0[x, y/x, \dots]$ je splnitelná.

Podobně se dá dokázat, že je klauzule $C[t]$ splnitelná, právě když je splnitelná $C[t/x] \vee t \neq x$.

Po každém kroku algoritmu je tedy získaná klauzule splnitelná, právě když je C splnitelná, a tvrzení platí. \square

2.1.3 Vytvoření instancí

Z FOL axiomů ve tvaru plochých klauzulí se v této fázi sestrojí axiomy nové SAT úlohy jako sjednocení množin, získaných podle následujícího algoritmu.

Algoritmus 2 (Vytvoření instancí)

vstup: plochá klauzule $C(x_1, \dots, x_m)$, množina \mathcal{F} funkcí vyskytujících se v C

výstup: množina T booleovských formulí

0. $D := \{1, \dots, n\}$
1. $T := \{C(a_1, \dots, a_m) \mid (a_1, \dots, a_m) \in D^m\}$
2. **while** $\exists \varphi \in T, \exists a \in D$ takové, že φ obsahuje $a = a$ **do**
 $T := T \setminus \{\varphi\}$
3. **while** $\exists \varphi \in T, \exists \varphi_0, \exists a, b \in D, a \neq b$ takové, že φ je tvaru $\varphi_0 \vee a = b$ **do**
 $T := (T \setminus \{\varphi\}) \cup \{\varphi_0\}$
4. **forall** $f \in \mathcal{F}, a_1, \dots, a_{ar(f)} \in D$ **do**
 označ \bar{a} vektor $(a_1, \dots, a_{ar(f)})$
 $T := T \cup \{„f(\bar{a}) = 1“ \vee „f(\bar{a}) = 2“ \vee \dots \vee „f(\bar{a}) = n“\}$
for $b_1, b_2 \in D$ **do**
 $T := T \cup \{„f(\bar{a}) \neq b_1“ \vee „f(\bar{a}) \neq b_2“\}$
5. **return** T

Množina T získaná algoritmem 2 obsahuje dva typy formulí. V 1. kroku jsou do T zařazeny všechny instance klauzule C . Ve 4. kroku jsou přidány formule popisující korektnost definice funkcí, tj. vlastnost, že funkce nabývá pro každý vektor argumentů právě jednu hodnotu.

Věta 2. *Pro každou plochou klauzuli C sestrojí algoritmus 2 množinu booleovských formulí T , která je splnitelná právě tehdy, když pro C existuje n -prvkový model.*

Důkaz. Instanci ploché klauzule můžeme chápat jako booleovskou formuli, protože *instanciací* každého plochého literálu l vznikne buď nějaká výroková proměnná „ l “, její negace \neg „ l “, nebo literál $a = b$. V 2. a 3. kroku algoritmu jsou všechny výskyty literálů $a = b$ z klauzulí množiny T odstraněny, takže výsledná T se skládá pouze z booleovských formulí.

Je-li klauzule C splnitelná, platí pro každé ohodnocení svých proměnných, tj. každá instance C platí. Splnitelnost formulí přidanych ve 4. kroku vyplývá z existence n -prvkového modelu klauzule C , neboť v něm je každá funkce podle zadaných pravidel korektně interpretována.

Naopak, je-li množina T splnitelná, lze sestrojit n -prvkový model klauzule C . Interpretace všech funkcí a relací je dána tím, které výrokové proměnné v jednotlivých instancích C platí, a díky formulím přidáním ve 4. kroku je interpretace

funkcí korektní. Klausule C je splnitelná, protože platí všechny její instance. \square

2.2 Srovnání s metodou SEM a LNH

Hlavní nedostatek metody MACE se projevuje u úloh, jejichž axiomy obsahují mnoho proměnných. Problematické jsou axiomy obsahující hluboko zanořené termy, neboť do nich se ve fázi flattening přidá velký počet nových proměnných.

Má-li klausule k proměnných, vytvoří se z ní při převodu na SAT n^k instancí, což odpovídá počtu různých vektorů v prostoru D^k . Po každém zvětšení velikosti hledaného modelu dochází k vygenerování velkého počtu nových booleovských formulí, což činí problémy při implementaci metody MACE. Neexistuje-li žádný dostatečně malý model, jsou paměťové nároky v uvedeném případě příliš velké a dojde k selhání hledače modelů.

Metoda SEM [14] tomuto problému předchází vynecháním fáze flattening. Instance se vytvářejí přímo z původních FOL axiomů a hledání vhodných interpretací probíhá postupným vyplňováním tabulek funkcí a relací v souladu se zadanými axiomy. Algoritmus vyplňování je prohledávání do hloubky (backtracking), jehož efektivita je vylepšena technikami *propagace podmínek* (v orig. constraint propagation) a *redukce symetrií* (v orig. symmetry reduction).

Myšlenka redukce symetrií vychází z pozorování, že všechny struktury izomorfní modelu nějaké množiny axiomů jsou také modelem této množiny. Při hledání však stačí najít jeden model. V rámci redukce symetrií se tedy usiluje o to, aby se neprocházely vždy všechny izomorfní struktury, ale jen zástupci tříd izomorfních struktur.

K redukci symetrií je v metodě SEM použita **heuristika nejmenšího čísla** (LNH). Vychází z této úvahy: Ve chvíli, kdy se vyplňuje jedna buňka tabulky hodnot funkce a stále ještě zbývá několik nepoužitých prvků univerza, není třeba zkoušet pro buňku všechny nepoužité prvky, ale pouze nejmenší z nich. Dosazení ostatních prvků by totiž vedlo jen k jinému izomorfnímu modelu.

Převod na SAT způsobí, že symetrie původní FOL úlohy již nejdou snadno rozpoznat a tedy ani redukovat. To je druhý nedostatek metody MACE.

Společnou překážkou obou standardních metod je nutnost hledat modely po-

stupně pro určenou velikost. Předpoklad velikosti se musí použít ve fázi vytváření instancí, kterou mají obě metody společnou.

Důvod, proč je i přes popsané nedostatky metoda MACE pro obecné úlohy obvykle efektivnější než metoda SEM, není jednoznačný. Významnou roli určitě hraje použití SAT-solveru, neboť výzkumu a vývoji efektivních algoritmů pro SAT se věnuje mnoho vědeckého úsilí, s čímž nemůže metoda vyvíjená jednotlivci reálně soupeřit.

2.3 Pokročilé techniky

Všechny systémy používající metodu MACE mají společný základ, který byl popsán v sekci 2.1. Rozdílná výkonnost různých systémů je dána tím, jaké pokročilejší metody používají ke zvýšení své efektivity. Na špičce stojí systém Paradox, který používá tyto inovativní techniky: *dělení klauzulí* (v orig. clause splitting), *definice termů* (v orig. term definitions), *statická redukce symetrií* (v orig. static symmetry reduction), *postupné prohledávání* (v orig. incremental search) a *odvození sortů* (v orig. sort inference).

Následující oddíly se těmto technikám věnují v míře, která odpovídá následnému využití v metodě konstrukce modelů pomocí CSP. Hlavním zdrojem je dokument [3].

2.3.1 Dělení klauzulí

Metoda dělení klauzulí má za cíl zmenšit počet proměnných v klauzulích tvořících axiomy, aniž by byla porušena splnitelnost úlohy. Je-li možné klauzuli *rozdělit*, nahradí se několika novými klauzulemi s menším počtem proměnných. Výsledkem je pak méně vygenerovaných booleovských formulí při vytváření instancí.

Definice. (*Dělení klauzulí*). *Nechť $C_1(\bar{x})$ a $C_2(\bar{y})$ jsou klauzule. Existuje-li alespoň jedna proměnná $x \in \bar{x} \setminus \bar{y}$ a alespoň jedna proměnná $y \in \bar{y} \setminus \bar{x}$, lze klauzuli $C_1(\bar{x}) \vee C_2(\bar{y})$ rozdělit. Výsledkem dělení jsou dvě nové klauzule:*

$$\begin{aligned} C_1(\bar{x}) \vee R(\bar{x} \cap \bar{y}) \\ C_2(\bar{y}) \vee \neg R(\bar{x} \cap \bar{y}) \end{aligned}$$

kde R je relační symbol nově zavedený do jazyka problému.

Příklad. Klauzuli $P(x, y, z) \vee Q(y, w)$ lze s pomocí nového unárního relačního symbolu R rozdělit na klauzule $P(x, y, z) \vee R(y)$ a $Q(y, w) \vee \neg R(y)$. \square

Věta 3. *Nechť C je dělitelná klauzule a T výsledek jejího dělení. Pak T je splnitelná množina, právě když C je splnitelná. Navíc klauzule z T obsahují méně proměnných než klauzule C .*

Důkaz. Nechť C je složená z klauzulí C_1 a C_2 a R je relační symbol přidáný při dělení klauzule C . Označme $R[e(\bar{x})] := R[x_1/e(x_1), \dots, x_n/e(x_n)]$ pro nějaké ohodnocení proměnných e .

Při ohodnocení e je platný právě jeden z literálů $R[e(\bar{x} \cap \bar{y})], \neg R[e(\bar{x} \cap \bar{y})]$. Pokud tedy obě klauzule množiny T při ohodnocení e platí, platí alespoň jedna z klauzulí $C_1[e(\bar{x})], C_2[e(\bar{y})]$. Z toho vyplývá, že i C při ohodnocení e platí.

Naopak, platí-li klauzule C při libovolném ohodnocení e , pak nanejvýš jeden z jejích disjunktů C_1, C_2 při ohodnocení e neplatí. Protože je relace R nově zavedený symbol při dělení, můžeme ji definovat libovolně. Neplatí-li $C_1[e(\bar{x})]$, definujeme hodnotu $R[e(\bar{x} \cap \bar{y})]$ jako true, v opačném případě jako false. V obou případech budou obě klauzule množiny T při ohodnocení e platné.

Počet proměnných v klauzulích z množiny T je přímým důsledkem definice dělení. Tyto klauzule obsahují proměnné \bar{x} resp. \bar{y} , což jsou podle předpokladu vlastní podmnožiny $\bar{x} \cup \bar{y}$, tj. množiny složené právě ze všech proměnných klauzule C . \square

Dělení klauzule lze obecně provést více způsoby a opakovaně, jsou-li podklauzule C_1, C_2 zvoleny vhodně. Paradox používá algoritmus, který o každé klauzuli rozhodne, zda je dělitelná, a případně dělení najde. Algoritmus je polynomiální v počtu proměnných původní klauzule a podle autorů dává dobré výsledky při testování.

Definice. *Řekneme, že dvě proměnné jsou v klauzuli C spojeny, pokud v C existuje literál, v němž se obě vyskytují.*

Algoritmus 3 (Dělení klauzule).

vstup: klauzule $C(\bar{z})$

výstup: NEDĚLITELNÁ, nebo dělení klauzule $C(\bar{z})$

1. najdi $x \in \bar{z}$ tak, že x je spojena s nejmenším počtem proměnných \bar{z}
2. $\bar{x} := \{z \in \bar{z} \mid z \text{ je spojena s } x\}$
3. **if** $\bar{x} = \bar{z}$ **then return** NEDEĚLITELNÁ
4. $C_1(\bar{x}) := l_1 \vee \dots \vee l_k$, kde l_i jsou všechny literály z C obsahující x
 $C_2(\bar{y}) := l'_1 \vee \dots \vee l'_k$, kde l'_i jsou všechny literály z C neobsahující x
5. zaveď do jazyka úlohy novou relaci R , $ar(R) = |\bar{x} \cap \bar{y}|$
6. **return** $C_1(\bar{x}) \vee R(\bar{x} \cap \bar{y}), C_2(\bar{y}) \vee \neg R(\bar{x} \cap \bar{y})$

Věta 4. Pro každou klauzuli C najde algoritmus 3 její dělení právě tehdy, když je možné ji rozdělit.

Důkaz. Dělení klauzule je možné provést, právě když existují dvě různé proměnné $x, y \in \bar{z}$, podle nichž lze množinu literálů klauzule rozdělit do dvou neprázdných množin: 1) literály neobsahující x a 2) literály obsahující x . Jsou-li všechny proměnné spojeny, existuje pro každou dvojici literál, v němž se společně vyskytuje. Takový literál by se nedal zařadit do žádné ze dvou množin, tedy dělení nelze provést.

Ukážeme, že je-li naopak dělení klauzule $C(\bar{z})$ možné, algoritmus 3 najde vlastní podmnožiny \bar{x}, \bar{y} množiny \bar{z} , které vyhoví definici $\bar{x} \setminus \bar{y} \neq \emptyset$ a $\bar{y} \setminus \bar{x} \neq \emptyset$, a že příslušné klauzule $C_1(\bar{x}), C_2(\bar{y})$ jsou neprázdné.

Nechť x je proměnná vybraná v 1. kroku algoritmu a $\bar{x} \subset \bar{z}$ množina určená v 2. kroku. Pak speciálně existuje proměnná $y \notin \bar{x}$, která se nevyskytuje s x ve stejném literálu. Libovolný literál obsahující y je ve 4. kroku zařazen do klauzule $C_2(\bar{y})$. Žádná z klauzulí $C_1(\bar{x}), C_2(\bar{y})$ proto není prázdná a zřejmě také platí $x \notin \bar{y}$. □

2.3.2 Definice termů

Cílem techniky definice termů je omezit počet proměnných přidaných do klauzulí ve 2. kroku flatteningu, kde se nahrazuje každý složitější term proměnnou.

Je-li t libovolný term neobsahující proměnné a složitější než konstanta (např. $f(a, g(b))$ pro konstanty a, b), má smysl jeho hodnotu *definovat* jako novou konstantu c . Klauzule $C[t]$ se pak nahradí klauzulemi $C[t/c]$ a $t = c$ a teprve na nich se provede flattening.

Věta 5. *Nechť $C[t]$ je klauzule, v níž se vyskytuje term t neobsahující proměnné, a necht' c je konstanta. Potom množina klauzulí $\{C[t/c], t = c\}$ je splnitelná právě tehdy, když je splnitelná klauzule $C[t]$.*

Důkaz. Protože term t neobsahuje proměnné, závisí jeho hodnota pouze na interpretaci konstant a funkcí, ze kterých se skládá, a nikoliv na ohodnocení proměnných.

Pokud v nějaké struktuře \mathcal{S} platí formule $t = c$, je hodnota termu t a konstanty c stejná. Z platnosti klauzule $C[t/c]$ tedy vyplývá i platnost klauzule $C[t]$.

Naopak platí-li ve struktuře klauzule $C[t]$, můžeme nově zavedenou konstantu c interpretovat stejnou hodnotou, jakou má term t . Pak samozřejmě platí obě formule $C[t/c]$ i $t = c$ □

Po flatteningu může každá ze získaných klauzulí obsahovat nanejvýš stejný počet proměnných jako $C[t]$. Žádoucí je samozřejmě stav, kdy je tento počet menší. U klauzule $C[c]$ je jisté, že to nastane, neboť t byl složitější term než konstanta c . Ovšem vztah počtu proměnných klauzulí $t = c$ a $C[t]$ po flatteningu záleží na vnitřní stavbě $C[t]$, což ilustruje následující příklad.

Příklad. Vliv definice termu $f(a) := c$ na klauzule získané po flatteningu z klauzulí $R(f(a), a)$ a $R(f(a), b)$.

- $R(f(a), a)$

- standardní flattening

$$R(y, x) \vee a \neq x \vee f(x) \neq y$$

- definice $f(a) := c$ a následný flattening

$$\begin{array}{ccc} R(c, a) & \longrightarrow & R(x, y) \vee c \neq x \vee a \neq y \\ f(a) = c & & f(x) = y \vee a \neq x \vee c \neq y \end{array}$$

- $R(f(a), b)$

- standardní flattening

$$R(y, z) \vee a \neq x \vee f(x) \neq y \vee b \neq z$$

- definice $f(a) := c$ a následný flattening

$$\begin{array}{ccc} R(c, b) & \longrightarrow & R(x, y) \vee c \neq x \vee b \neq y \\ f(a) = c & & f(x) = y \vee a \neq x \vee c \neq y \end{array}$$

Pouze v případě $R(f(a), b)$ tedy došlo k úspoře počtu proměnných při definici termu před flatteningem. \square

Algoritmus, jakým systém Paradox vybírá termy vhodné k definování, Claessen a Sörensson v [3] neuvádí. Pouze zmiňují, že se v různých verzích Paradoxu liší.

2.3.3 Statická redukce symetrií

Myšlenku redukce symetrií pomocí LNH převzali autoři Paradoxu z metody SEM. V ní však lze použít LNH dynamicky, přímo v průběhu hledání modelu. To není pro metodu MACE prakticky možné, protože hledání probíhá až v SAT-solveru, který již o vztazích původní FOL úlohy nic neví. Redukci symetrií je tedy třeba provést staticky, před spuštěním SAT-solveru.

Statická redukce symetrií, jak ji navrhli autoři Paradoxu, spočívá v úpravě algoritmu převodu na SAT. Ke všem booleovským formulím vygenerovaným tak, aby nová SAT úloha zachovala vlastnost splnitelnosti původní úlohy, se navíc přidávají ještě booleovské formule, které omezí počty izomorfních modelů. Jinými slovy, pokud v původní úloze existovaly izomorfní struktury, které byly modelem zadané množiny axiomů, po převodu na SAT budou modelem SAT úlohy jen některé z nich, ideálně právě jedna. Tím se zrychlí hledání modelů, neboť některé volby interpretací budou okamžitě vyloučeny bez hlubšího zkoumání.

Konkrétní podoba statické redukce symetrií v Paradoxu je podrobněji popsána pouze pro problémy, které obsahují konstanty nebo alespoň jednu unární funkci. Mezi všemi izomorfními modely je vybrána struktura v určité *kanonické formě*, inspirované tím, jaké modely vznikají při použití LNH.

Definice. *Nechť a_1, \dots, a_k jsou všechny konstanty daného jazyka ve FOL a necht \mathcal{S} je struktura s univerzem $\{1, \dots, n\}$. Řekneme, že \mathcal{S} je v kanonické formě, pokud platí následující podmínky:*

- $a_1 = 1$
- pro každé i , $1 < i \leq k$ existuje $j < i$ splňující $a_j = a_i - 1$

Věta 6. *Každá FOL struktura \mathcal{S} velikosti n je izomorfní nějaké struktuře v kanonické formě.*

Důkaz. Necht a_1, \dots, a_k jsou konstanty daného jazyka.

Definujme permutaci π množiny $\{1, \dots, n\}$ tímto způsobem:

- $\pi(a_1^S) = 1$
- $\pi(a_2^S) = \begin{cases} 1, & a_2^S = a_1^S \\ 2, & \text{jinak} \end{cases}$
- pro každé $i, 2 < i \leq k$

$$\pi(a_i^S) = \begin{cases} a_j^S, & \text{existuje-li } j < i : a_i^S = a_j^S \\ \max\{a_j^S \mid j < i\} + 1, & \text{jinak} \end{cases}$$
- zbylým prvkům množiny $\{1, \dots, n\}$ přiřadí π zbývající nepoužité hodnoty $n, n-1, \dots$ libovolně.

Pomocí funkce π lze vytvořit novou strukturu \mathcal{S}' permutací prvků univerza struktury \mathcal{S} . \mathcal{S}' je pak izomorfní s \mathcal{S} a z konstrukce π okamžitě plyne, že je \mathcal{S}' v kanonické formě. \square

Význam kanonické formy spočívá v tom, že jsou-li již interpretovány konstanty a_1, \dots, a_{i-1} , pak za hodnotu a_i je možné zvolit jen takový prvek univerza, který se již v $\{a_1, \dots, a_{i-1}\}$ objevil, nebo nejmenší z dosud nepoužitých prvků. Speciálně tedy $a_i \leq i$.

Algoritmus, jak převést do výrokové logiky požadavky na kanonickou formu modelu, je přímočarý. Konstanty FOL úlohy a_1, \dots, a_k se libovolně uspořádají a do množiny axiomů se navíc přidají tyto booleovské formule:

- „ $a_1 = 1$ “ (tj. formule obsahující pouze výrokovou proměnnou „ $a_1 = 1$ “)
- „ $a_i \neq d$ “ \vee „ $a_1 = d - 1$ “ \vee „ $a_2 = d - 1$ “ \vee \dots \vee „ $a_{i-1} = d - 1$ “
pro každé $i, 1 < i \leq \min\{k, n\}$ a každé $d, 1 < d \leq n$
- „ $a_i = 1$ “ \vee „ $a_i = 2$ “ \vee \dots \vee „ $a_i = i$ “ pro každé $i, 1 < i \leq \min\{k, n\}$

Tyto klauzule jsou sice navíc, protože jsou důsledkem předchozích klauzulí a axiomů popisujících korektnost definice každé a_i jako (nulární) funkce, nicméně usnadňují a zrychlují práci SAT-solveru.

Je-li počet konstant k menší než velikost hledaného modelu, lze redukovat ještě více symetrií úlohy pomocí náhodně vybrané unární funkce f . Požadavek

na model v kanonické formě se rozšíří o podmínky, jakých hodnot může nabývat funkce f . Myšlenka je podobná jako u konstant, rovněž důkaz oprávněnosti těchto podmínek by byl podobný důkazu věty 6. Za hodnotu $f(i)$ bude možné zvolit pouze prvek, který se mohl objevit v $\{a_1, \dots, a_k, f(1), \dots, f(i-1)\}$, nebo nejmenší dosud nepoužitý prvek. Speciálně tedy $f(i) \leq k + i$.

Statická implementace těchto podmínek v Paradoxu spočívá pro $k > 0$ v tom, že pro každé $i > 1$ přidá k SAT úloze nové axiomy

$$\text{„}f(i) = 1\text{“} \vee \text{„}f(i) = 2\text{“} \vee \dots \vee \text{„}f(i) = k + i\text{“} \quad (2.1)$$

Případ $k = 0$ (tedy neobsahuje-li úloha žádné konstanty) je ošetřen zvlášť. Požadavek $f(i) \leq i$ by mohl změnit splnitelnost úlohy (například by v každém modelu vyžadoval existenci alespoň jednoho idempotentního prvku), proto se v tomto případě přidá do jazyka úlohy nová konstanta a dále se postupuje jako pro $k = 1$.

Důsledkem předchozích odstavců a věty 6 je fakt, že pokud nějaký model množiny FOL axiomů existuje, lze nalézt také model v kanonické formě, a proto i vytvořená SAT úloha je splnitelná.

2.3.4 Postupné prohledávání

Technika postupného prohledávání souvisí se schopností solveru využívat již dříve (tj. pro menší velikosti n) odvozené vlastnosti řešeného problému. Z neúspěchů při hledání menších modelů se může solver nějakým způsobem poučit a při hledání větších modelů se chybám vyhnout.

V Paradoxu je tato technika implementována uvnitř zabudovaného SAT-solveru. Pro metodu konstrukce modelů pomocí CSP je tedy prakticky nepoužitelná, pokud ji CSP-solver nepodporuje.

2.3.5 Odvození sortů

Některé FOL úlohy vedou při řešení k modelům, v nichž se objevují prvky rozdílných typů, tzv. *sortů*. Příkladem jsou geometrické úlohy, které se týkají bodů, přímek či rovin a jejich vzájemných vztahů. Každý tento objekt má jiný sort,

nicméně ve FOL (na rozdíl od logik vyšších řádů) musí být interpretovány s použitím jednoho společného univerza, což je zbytečně složité. Kdyby bylo u takové úlohy předem známo, na kterých sortech které funkce a relace operují, daly by se tyto informace využít k efektivnějšímu hledání modelů. Například metoda statické redukce symetrií by se mohla aplikovat pro každý sort zvlášť.

Paradox používá speciální algoritmus, který se pokouší v zadaném problému informace o rozdílných sortech najít. Claessen a Sörensson v [3] uvádí, že z testovaných úloh z knihovny TPTP (verze 2.5.0 z roku 2003) bylo přibližně 30 % takových, v nichž skutečně algoritmus našel více sortů. Nicméně se dá předpokládat, že šlo např. o geometrické úlohy, na něž se tato práce nezaměřuje.

3. Metoda převodu na CSP

Tato kapitola popisuje návrh a implementaci metody konstrukce konečných modelů FOL axiomů pomocí CSP. Schéma řešení je podobné jako u metody MACE – úloha se převede na CSP a poté se pomocí CSP-solveru hledá její řešení, postupně pro vzrůstající velikost hledaného modelu.

Jako CSP lze formulovat problémy z mnoha oblastí a i různé CSP-solvery se často specializují jen na nějaký konkrétní typ úloh. Mohou se proto výrazně lišit jak používaným modelovacím jazykem, tak vůbec úrovní abstrakce, na které s CSP úlohami pracují. To znesnadňuje objektivní porovnávání jejich účinnosti, neboť každou testovací úlohu by bylo nutné převádět do všech modelovacích jazyků a zároveň tím žádný z nich nezvýhodnit či neznevýhodnit. Snahou o řešení těchto potíží je zavedení nezávislého modelovacího jazyka MiniZinc [10], jemuž se věnuje první část této kapitoly.

Metoda převodu na CSP je realizována pomocí jazyka MiniZinc. Sekce 3.2 popisuje způsob užití pokročilých technik Paradoxu v navržené metodě a začlenění specifických vlastností MiniZincu. Implementace těchto technik a postup překladač FOL úlohy do MiniZincu jsou shrnuty v sekci 3.3.

Sekce 3.4 se týká CSP-solveru Gecode, který byl vybrán pro řešení úlohy vymodelované v MiniZincu.

Poslední část této kapitoly shrnuje celý postup vytvořeného nástroje při hledání modelu FOL úlohy, od zpracování jejího zadání až po nalezení modelu resp. zjištění jeho neexistence.

3.1 MiniZinc

Nezávislý modelovací jazyk MiniZinc vyvíjejí N. Nethercote a kol. od roku 2007 [10]. Kládou si za cíl vytvořit jazyk, který by se mohl stát standardem pro zápis CSP úloh. Úroveň abstrakce MiniZincu popisují jeho autoři v [10] takto:

„MiniZinc je na dostatečně vysoké úrovni, aby byl schopný snadno vyjádřit většinu CSP úloh nezávisle na konkrétním solveru; například podporuje množiny, pole, uživatelem definované predikáty a někte-

ré automatické převody mezi různými typy. Zároveň však je MiniZinc dostatečně nízkoúrovňový, aby ho šlo snadno propojit s velkým počtem solverů. Pracuje ve FOL a používá pouze takové typy proměnných, které jsou podporovány většinou existujících CSP-solverů: celá čísla, reálná čísla, booleovské hodnoty a množiny celých čísel.“

Teoretický popis CSP úlohy pomocí trojice (V, D, C) se do MiniZinc promítá tak, že množiny V a D jsou určeny deklarací rozhodovacích proměnných a relace v podmínkách z C jsou konstruovány pomocí rovnosti či zabudovaných logických a aritmetických operací. Jako nadstavbu nad základním CSP jazykem MiniZinc nabízí parametry, struktury proměnných, definice predikátů a další nástroje určené k snazšímu modelování úloh (viz oddíl 3.1.1). Význam tzv. prohledávacích anotací pro komunikaci s cílovým CSP-solverem je vysvětlen v oddílu 3.1.2.

Pro popis základní syntaxe MiniZinc bylo čerpáno z dokumentu [7], který slouží jako učebnice modelování v MiniZinc.

3.1.1 Základy syntaxe

CSP model se v MiniZinc skládá ze čtyř hlavních částí: deklarace proměnných, seznam podmínek, typ řešení a volitelně formát výstupu.

Proměnná může být dvojího druhu. Buď se jedná o *parametr* (druh `par`), jehož hodnota je pevně určena (inicializována) při modelování problému, nebo je to rozhodovací proměnná (druh `var`). Nemá-li proměnná výslovně uvedený druh, považuje se automaticky za parametr.

Kromě druhu musí mít proměnná určený některý z těchto typů:

<i>Skaláry:</i>	<code>int</code>	celá čísla
	<code>float</code>	reálná čísla
	<code>bool</code>	booleovské hodnoty
	<code>m..n</code>	(<code>m,n</code> parametry) podmnožina <code>int</code> či <code>float</code>
<i>Struktury:</i>	<code>set of skalár</code>	množina skalárních prvků
	<code>array[M₁,...,M_d] of skalár</code>	d -rozměrné pole skalárů
	(M_1, \dots, M_d indexové množiny)	

Přístup k hodnotám v poli je stejný jako ve většině programovacích jazyků, tedy např. prvek pole `Pole` s indexem 3 je `Pole[3]`. V deklaraci pole musí být

jeho indexová množina pevná, tj. druhu `par`. Důležitá vlastnost polí je ta, že indexem prvku pole může být i rozhodovací proměnná.

Podmínka (uvozená slovem `constraint`) je pravdivostní výraz obsahující různé aritmetické, logické či množinové operace s proměnnými. Dají se použít předdefinované funkce `forall` resp. `exists`, které rozvinou seznam pravdivostních výrazů v jeden velký výraz, a to spojením prvků konjunkcí (`/\`) resp. disjunkcí (`\|`). Nejčastěji se tyto funkce při modelování používají pro zastoupení univerzálního a existenčního kvantifikátoru.

Typ řešení může být optimalizačního rázu, kdy uživatel určí proměnnou, která má mít největší či nejmenší hodnotu při splnění všech podmínek. Druhým typem je hledání libovolného vyhovujícího řešení (`solve satisfy`).

Mezi další vlastnosti modelování v MiniZincu patří možnost oddělit model od jeho dat. Každá proměnná typu `par` musí mít přiřazenou konkrétní hodnotu, nicméně inicializaci je možné provést v odděleném *datovém souboru*. To usnadňuje modelování různých úloh, které se liší právě jen v číselných parametrech.

Uživatel si v MiniZincu také může definovat vlastní *predikáty*, což se dá využít v případě, kdy se často používá nějaký stejný vztah mezi proměnnými. Nejpoužívanější predikáty (např. `all_different`, `maximum`, `minimum`) jsou předdefinovány v knihovně *globálních* predikátů. Každý CSP-solver může poskytovat vlastní definici globálních predikátů, která je pro jeho řešící algoritmus nejvýhodnější.

Pro ilustraci, jak typický CSP model v MiniZincu vypadá, je uveden model latinského čtverce rozměru N . Latinský čtverec je tabulka rozměru $N \times N$, kde se v každém řádku a každém sloupci vyskytuje každé číslo mezi 1 a N .

```
% Latinský čtverec velikosti N
% globální predikáty
include "globals.mzn";

% parametr: rozměr pole
int: N;

% 2D-pole pro rozhodovací proměnné - políčka latinského čtverce
array[1..N, 1..N] of var 1..N: Ctverec;

% podmínka na řádky a sloupce
constraint forall(i in 1..N)(all_different([Ctverec[i,j] | j in 1..N]));
constraint forall(j in 1..N)(all_different([Ctverec[i,j] | i in 1..N]));

solve satisfy;
```

(znak % uvozuje komentáře, konstrukce `[Ctverec [i, j] | j in 1..N]` převádí dvou-
rozměrné pole na jednorozměrné)

K souboru obsahujícímu tento kód je nutné vytvořit ještě datový soubor ob-
sahující například `N=10;`. Tím je inicializován parametr N hodnotou 10.

3.1.2 Prohledávací anotace

Významnou vlastností MiniZincu je možnost použití tzv. *anotací*, které slouží ke komunikaci s cílovým CSP-solverem. Každá podmínka či každá proměnná může mít připojenou anotaci (pomocí spojky `:`). Anotace u proměnné může solveru například říci, že jde o proměnnou nově přidanou při převodu do FlatZincu (viz sekce 3.1.3) a že ji definuje podmínka s příslušnou anotací.

Důležitá je především tzv. *prohledávací anotace* (v orig. search annotation), která představuje jedinou možnost, jak může autor modelu ovlivnit postup cílového CSP-solveru při konstrukci řešení $\varphi : V \rightarrow D$. Nemí-li specifikována žádná prohledávací anotace, nebo je-li vybrána nevhodně, nemusí solver vyřešit složitější úlohu vůbec. Prohledávací anotace je vždy spojena s nějakou podmnožinou rozhodovacích proměnných V , jejímž prvkům je pak při hledání řešení přiřazována hodnota jako prvním.

Prohledávací anotace se připojuje k položce `solve` ve tvaru

```
solve :: int_search( pole proměnných, výběr proměnné,  
                   přiřazení hodnoty, strategie) satisfy;
```

pokud pole proměnných obsahuje skaláry typu `int`. Analogicky se používají ano-
tace `bool_search` a `set_search`.

Význam parametrů prohledávací anotace a některé jejich typické hodnoty jsou tyto (úplný seznam je v [1]):

- *výběr proměnné* určuje v rámci *pole proměnných* pořadí, ve kterém se proměnným hodnoty přiřazují
 - `input_order`: podle pořadí v poli
 - `first_fail`: vybere se vždy proměnná s momentálně nejmenším ob-
rem možných hodnot

- *přiřazení hodnoty* určuje hodnotu přiřazenou vybrané proměnné
 - `indomain_min`: nejmenší prvek z jejího oboru možných hodnot
 - `indomain_max`: největší prvek z jejího oboru možných hodnot
- *strategie* určuje hledací strategii
 - `complete`: prochází prostor všech řešení (tzv. řešení hrubou silou)

Výše uvedeným způsobem je možné definovat více anotací A_1, \dots, A_k a připojit je jedinou anotací `seq_search([A1, ..., Ak])`. Příliš mnoho zahrnutých anotací však může vést ke zpomalení CSP-solveru.

3.1.3 FlatZinc

Překlad úlohy vymodelované v MiniZincu do syntaxe vybraného CSP-solveru usnadňuje jazyk FlatZinc [1]. FlatZinc je podmnožinou MiniZincu v tom smyslu, že používá stejné datové typy (kromě vícerozměrných polí), stejné typy řešení a anotace. Veškeré aritmetické a logické operace však jsou určitým způsobem zjednodušeny, normalizovány a zakódovány v zabudovaných predikátech. Autoři MiniZincu rovněž poskytují skript *mzn2fzn* pro překlad z MiniZincu do FlatZincu. Načíst model ve FlatZincu je pro CSP-solver výrazně jednodušší.

Postup *mzn2fzn* při překladu z MiniZincu do FlatZincu je popsán v [9]. Mimo jiné obsahuje tyto kroky:

- Rozvnutí funkcí `forall`, `exists` (sekce 2.1.4 a 2.1.5 v [9])

Každý výraz obsahující tyto funkce se nahradí alternativním výrazem, ve kterém se objevují pouze konjunkce konstantních literálů. Např. pro $n = 3$ se provede úprava

`forall(i in 1..n)(R[i])` nahradí za `R[1] /\ R[2] /\ R[3]`

V důsledku se tedy z formule vytvoří všechny její instance, které se spojí konjunkcí. Podobný proces probíhá v metodě MACE ve fázi vytváření instancí.

- Rozklad numerických výrazů (sekce 2.2.3 v [9])

Kdykoli nějaká rovnost či nerovnost obsahuje složený aritmetický výraz, nebo když index nějakého prvku pole není přímo číslo či proměnná, příslušný nevyhovující výraz se nahradí novou proměnnou. Její hodnota je pak určena v nové odpovídající podmínce. Tento krok v podstatě provádí techniku definice termů.

- Zjednodušení polí (sekce 2.2.6 v [9])

Každé vícerozměrné pole je nahrazeno jednorozměrným polem. Přepočít vektoru indexů původního pole na index v novém poli je zařazen jako nová podmínka CSP modelu.

Na závěr skript *mzn2fzn* provádí optimalizaci výsledného kódu, která však není v oficiální dokumentaci popsána. Z pohledu uživatele tato optimalizace spočívá v odstranění některých proměnných, které byly převodem do FlatZinc nově vygenerovány a jejichž hodnota je už na začátku hledání jednoznačně určena. Ovšem pro velké úlohy probíhá tato optimalizace tak pomalu, že se vyplatí ji parametrem `--no-optimize` skriptu *mzn2fzn* vypnout.

3.2 Užítí pokročilých technik

Tato sekce popisuje způsob, jakým byly do navržené metody hledání modelu velikosti n začleněny pokročilé techniky standardních metod. Dále je popsán výběr prohledávací anotace.

Instanciací formulí a flattening pomocí definice termů provádí automaticky skript *mzn2fzn* (viz sekce 3.1.3). Při převodu do MiniZinc se jimi tedy není třeba zabývat.

Z ostatních pokročilých technik Paradoxu navržená metoda používá dělení klauzulí (beze změny) a statickou redukci symetrií.

Redukce symetrií vychází z LNH aplikované na konstanty, unární nebo binární funkce a od podoby implementované v Paradoxu se liší. Paradox používá LNH na všechny konstanty a pokud jich je méně než velikost hledaného modelu, omezí ještě u náhodně vybrané unární funkce hodnotu na některých prvcích. Navržená metoda naopak omezí pouze jednu (nejčtenější) konstantu, kterou v sou-

ladu s LNH interpretuje jako prvek 1, a dále aplikuje LNH na funkci s nejmenší aritou. Je-li v úloze takových funkcí více, vybere se ta nejčtetnější. Pouze pokud se v úloze nevyskytuje žádná unární ani binární funkce, použije se LNH na množinu všech konstant. Vybraný postup redukce symetrií je podle výsledků předběžného testování účinnější než postup Paradoxu.

Aplikace LNH na konstanty je stejná jako ve statické redukci symetrií Paradoxu. Konstanty jsou seřazeny podle počtu výskytů ve vstupních FOL formulích od nejčtetnější.

Aplikace LNH na unární funkce se provádí po vzoru sekce 2.3.3, kde se ve formuli (2.1) dosadí $k = 1$, protože je omezena hodnota pouze jedné konstanty.

Algoritmus užití LNH na binární funkci byl nově navržen v této práci. Pracuje s představou, že definování hodnot funkce se provádí vyplňováním buněk její multiplikační tabulky (viz tabulka (3.1)), podobně jako u metody SEM [14].

f	1	2	...	n
1	$f(1, 1)$	$f(1, 2)$...	$f(1, n)$
\vdots	\vdots	\vdots	\ddots	\vdots
n	$f(n, 1)$	$f(n, 2)$...	$f(n, n)$

Tabulka 3.1: Multiplikační tabulka binární funkce f

Tabulka se vyplňuje od levého horního rohu a na začátku je prázdná. V i -tém kroku vyplňování se určí hodnoty prázdných buněk levého horního čtverce $i \times i$. Prvky univerza jsou při vyplňování každé buňky rozděleny do dvou disjunktních množin:

- *Použité prvky* obsahují ty hodnoty, které už se v nějaké buňce objevily, a množinu $\{1, \dots, i\}$
- *Nepoužité prvky*

Seznam možných hodnot buňky se skládá ze všech použitých prvků a jen z jednoho, nejmenšího, prvku nepoužitého – podobně jako při užití LNH na konstanty. Množina $\{1, \dots, i\}$ se řadí mezi použité prvky proto, aby nedošlo ke změně splnitelnosti úlohy.

Uvedený postup vede ke zřejmému pozorování, že se v levém horním čtverci $i \times i$ bude vyskytovat nanejvýš hodnota $i^2 + 1$, a k následujícímu algoritmu.

Algoritmus 4 (Užití LNH na binární funkci).

vstup: nedefinovaná binární funkce f

výstup: definovaná binární funkce f

1. polož $f(1, 1) \in \{1, 2\}$

2. **for** $i = 2, \dots, n$ **do**

while $\exists j, k \leq i$ že $f(j, k)$ není definováno **do**

$N := \{1, \dots, n\} \setminus \{f(l, m) \mid l, m \in \{1, \dots, i\}, f(l, m) \text{ je definováno}\}$

polož $f(j, k) \in \{1, \dots, \max\{i + 1, \min\{N\}\}$

3. **return** f

Otázku, jaká **prohledávací anotace** je pro úlohy FOL převedené na CSP nejefektivnější, je obtížné zodpovědět obecně. Pokud FOL úloha obsahuje jen jedinou funkci, vytvoří se prohledávací anotace spojená s proměnnou, která tuto funkci v MiniZincu reprezentuje. Jakmile však úloha obsahuje funkcí, relací či konstant více, je třeba některé vybrat a navíc zvolit pořadí, v jakém se odpovídající anotace zapíše do `seq_search`.

Předběžné testování ukázalo, že v průměru se nejlepších výsledků dosahuje s prohledávací anotací, v níž je na prvním místě nejčtenější funkce (unární či binární), na druhém místě množina konstant a na třetím místě nejčtenější relace s nejmenší aritou. Toto pořadí také koresponduje s tím, jakým způsobem se provádí redukce symetrií.

Za parametr prohledávací anotace *výběr proměnné* bylo zvoleno `first_fail`, protože zastupuje výkonnou heuristiku (tzv. *search-rearrangement method*), která u algoritmu prohledávání do hloubky vede obvykle ke zlepšení o několik řádů [6].

Pro parametr *přiřazení hodnoty* bylo vybráno `indomain_min` v souladu s použitím LNH pro redukci symetrií.

3.3 Převod do MiniZincu

Tato sekce popisuje postup překladu množiny FOL axiomů (klauzulí) na model v MiniZincu, jehož parametrem je velikost FOL modelu (`int: n`). Zdrojový kód vytvořeného nástroje je k dispozici na přiloženém CD, proto jsou zde zřejmé

kroky překlada vynechány.

Volba datových struktur v MiniZincu vychází z toho, že hledání interpretace jednotlivých funkcí a relací ve FOL je ekvivalentní hledání hodnot rozhodovacích proměnných v CSP. Každá konstanta je tedy reprezentována celočíselnou rozhodovací proměnnou `var 1..n` a každá nulární relace booleovskou proměnnou `var bool`. Pro reprezentaci funkcí a relací kladné arity jsou použita n -rozměrná pole celočíselných a booleovských rozhodovacích proměnných. Např. unární funkci odpovídá v MiniZincu proměnná `array[1..n] of var 1..n` a binární relaci odpovídá proměnná `array[1..n,1..n] of var bool`.

Před samotným překladem do MiniZincu se na každý axiom použije technika dělení klauzulí. Získaná množina klauzulí se do MiniZincu zakóduje přímočaře díky tomu, že indexem prvku pole může být i rozhodovací proměnná. Na přeložené klauzule se aplikuje funkce `forall`, protože MiniZinc nepodporuje otevřené formule. Například axiom $R(f(x), y, z)$ by se převedl na podmínku

```
constraint forall(x,y,z in 1..n)(s_R[s_f[x],y,z]);
```

(předpona `s_` předchází nechtěnému použití nějakého klíčového slova MiniZincu).

Technika redukce symetrií spočívá v přidání nových podmínek do kódu MiniZincu. Konkrétně algoritmus LNH pro binární funkci f je implementován přidáním této podmínky:

```
constraint forall (i in 1..n) (  
  let { var 1..n: max } in  
    max <= i*i+1;  
    maximum(max, [s_f[j,k] | j,k in 1..i]);  
    forall(j in i+1..n)(j >= max \ / (exists(k,l in 1..i)(s_f[k,l] = j)));
```

Příkaz `let { var 1..n: max } in` zavede (pro každé i) novou proměnnou `max`, která s pomocí globálního predikátu `maximum` bude obsahovat největší použitou hodnotu v levém horním čtverci $i \times i$ tabulky f . V souladu s algoritmem 4 se musí v tomto čtverci vyskytovat každá hodnota mezi $i + 1$ a `max`, což zajišťuje formule na posledním řádku.

3.4 Gecode

Gecode (Generic constraint development environment) je prostředí pro vývoj systémů a aplikací zabývajících se CSP. Zároveň poskytuje CSP-solver implementovaný v jazyce C++, který od verze 3.2.0 nabízí možnost vstupu v jazyce FlatZinc.

Pro CSP-solvery podporující FlatZinc pořádají autoři MiniZinc od roku 2008 každoročně soutěž MiniZinc Challenge [11], jejímž vítězem byl zatím pokaždé Gecode. Rovněž podle R. Cipriana a kol. v [2] jde o nástroj s vynikající výkonností, jehož nevýhodou je pouze menší uživatelská přívětivost při modelování úloh, než nabízejí jiné solvery. A zde je právě příležitost pro MiniZinc.

Ve zvoleném postupu MiniZinc \rightarrow FlatZinc \rightarrow Gecode se kombinuje efektivní kódování úloh v MiniZincu s výpočetní silou Gecode.

3.5 Vytvořený nástroj

Navržená metoda konstrukce modelů pomocí CSP byla implementována v programovacím jazyce Perl. Zdrojový kód programu obsahuje přibližně tisíc řádek a je k dispozici na přiloženém CD. Spustit ho lze příkazem

```
perl preklad.pl Nazev [-tiLimit]
```

(*Nazev* je název úlohy podle TPTP (např. BOO059-1.p) a *Limit* je časový limit pro hledání modelu ve vteřinách (implicitně 300 s)).

Algoritmus vytvořeného nástroje

Vstup: Množina FOL axiomů ve formátu TPTP

Výstup: Model množiny axiomů, nebo „Unknown“

0. Formát zadávaných úloh

Program byl vytvářen tak, aby se jeho účinnost dala testovat na CNF úlohách z TPTP [12]. Je-li vstup v obecném FOL formátu, lze pro převod do CNF použít systém FLOTTER (viz sekce 2.1) nebo skripty, které jsou součástí TPTP.

1. Načtení vstupu a překlad do MiniZincu

Vstupní soubor s CNF úlohou se čte jednou, použité funkce a relace se zaznamenávají během překladu jednotlivých klauzulí. Výsledek překladu se uloží do

dočasného souboru `temp.mzn`.

2. Vytvoření datového souboru

Parametr `int: n`, zastupující velikost hledaného modelu, se inicializuje v dočasném datovém souboru `temp.dzn` kódem `n=2;`. Tím je CSP model kompletní.

3. Překlad do FlatZincu

Příkaz

```
mzn2fzn -G gecode --no-optimise -S -o temp.fzn temp.mzn temp.dzn
```

vytvoří soubor `temp.fzn`, do nějž se uloží překlad modelu složeného ze souborů `temp.mzn temp.dzn` do jazyka FlatZinc. Volba `-G gecode` způsobí, že se při překladu použijí vlastní globální predikáty CSP-solveru Gecode. Volba `-S` generuje statistiku průběhu překladu.

4. Spuštění Gecode

Příkazem

```
fz -mode stat -time N temp.fzn
```

se spustí CSP-solver Gecode ve statistickém režimu a s horním limitem N vteřin na dobu běhu. Výstupem Gecode je buď nalezený model, nebo zpráva „Unknown“ signalizující vypršení časového limitu, anebo zpráva „Unsatisfiable“.

V případě vypršení limitu se v dalším hledání modelu nepokračuje, protože je nízká šance, že by pro větší velikost byl model nalezen se stejným časovým limitem.

Po výsledku „Unsatisfiable“ se hodnota n v datovém souboru `temp.dzn` přepíše na hodnotu o jedna větší a proces konstrukce se vrátí zpět do bodu 3.

Má-li úloha netriviální konečný model, popsaná metoda k němu musí po několika cyklech kroků 3, 4 dojít. Jedinými překážkami se mohou stát časový limit a nedostatek paměti použitého hardware.

4. Výsledky

Tato kapitola obsahuje výsledky testování vytvořeného nástroje na konstrukci konečných modelů pomocí CSP a srovnání se zástupci standardní metody. Systém Paradox reprezentuje metodu MACE, systém Mace4 metodu SEM.

4.1 Testování

Testovací úlohy byly vybrány z knihovny TPTP [12], která je určena pro všechny typy automatických dokazovačů vět. Knihovna TPTP obsahuje velké množství problémů, které jsou rozděleny do tříd podle toho, jaké oblasti se týkají. K testování byly použity všechny úlohy v CNF syntaxi s označením „Satisfiable“ (tj. úlohy, pro něž existuje model) z oblasti algebry, teorie čísel a topologie.

Konkrétně byly zařazeny úlohy z těchto tříd v uvedeném počtu:

- ALG (obecné algebraické úlohy) – 33 úloh
- BOO (Booleova algebra) – 15 úloh
- GRP (teorie grup) – 85 úloh
- LAT (teorie svazů) – 62 úloh
- LDA (levodistributivní algebry) – 24 úloh
- NUM (teorie čísel) – 5 úloh
- RNG (teorie okruhů) – 8 úloh
- ROB (Robbinsova algebra) – 5 úloh
- TOP (topologie) – 18 úloh

Testování proběhlo na clusteru Sněhurka s procesory Intel Core i7, 2266.745 MHz, 8192 KB Cache, 24 GB RAM. Dále byl použit software v těchto verzích: Paradox 4.0, Mace4 LADR-2009-11A, MiniZinc 1.3.2, Gecode 3.5.0 a Perl 5.10.0.

4.2 Přehled výsledků

V následujících tabulkách jsou zpracovány výsledky testování jednotlivých nástrojů. Název „Metoda CSP“ označuje nástroj vytvořený v této práci.

Kompletní výsledky testů jsou připojeny v příloze A. Časový limit pro vyřešení jedné úlohy byl nastaven na 5 minut. Údaj „fail“ znamená, že příslušný proces nebyl v časovém limitu úspěšně dokončen.

Tabulka (4.1) obsahuje pro každou třídu testovaných úloh a každý systém údaj, kolik úloh dané třídy systém vyřešil v časovém limitu.

Třída	Počet úloh	Metoda CSP	Paradox	Mace4
ALG	33	1	28	1
BOO	15	14	14	12
GRP	85	63	76	70
LAT	62	17	59	62
LDA	24	0	21	24
NUM	5	5	5	5
RNG	8	8	8	8
ROB	5	5	5	5
TOP	18	12	18	7
Celkem	255	125	234	194

Tabulka 4.1: Celkový počet vyřešených úloh z příslušné třídy

Tabulka (4.2) obsahuje srovnání rychlosti „Metody CSP“ s rychlostí obou existujících systémů. Nejprve se pro každou úlohu spočítalo, jaké procento z doby výpočtu „Metody CSP“ potřeboval Paradox resp. Mace4 na vyřešení dané úlohy. Poté se z těchto hodnot vypočítal průměr pro každou třídu úloh. Zahrnuty byly pouze úlohy, které oba porovnávané nástroje vyřešily v limitu; proto jsou třídy ALG a LDA vynechány.

Třída	Paradox	Mace4
BOO	23 %	371 %
GRP	7 %	18 %
LAT	15 %	0 %
NUM	12 %	0 %
RNG	18 %	0 %
ROB	11 %	57 %
TOP	27 %	64 %
Průměr	19 %	125 %

Tabulka 4.2: Srovnání rychlosti: průměrné procento z doby potřebné „Metodou CSP“

Například údaj „100 %“ by znamenal, že příslušný systém potřeboval na

vyřešení úloh průměrně stejnou dobu jako „Metoda CSP“. Hodnota menší než „100 %“ znamená, že existující systém je v průměru rychlejší. Ve speciálním případě vychází dokonce hodnota „0 %“, která je důsledkem zaokrouhlování na setiny při měření časů u testování, neboť pak se u velmi jednoduchých úloh uvádí doba výpočtu programu 0.00 s.

Tabulka (4.3) uvádí pro každou třídu úloh, jak velkou část doby výpočtu průměrně stráví „Metoda CSP“ v externích procesech – převáděním z MiniZincu do FlatZincu (údaj *mzn2fzn*) a hledáním CSP-řešení v Gecode. Zbylou část doby zabírají nutná volání systému a převod z TPTP do MiniZincu.

Do výpočtu uvedeného údaje byly zahrnuty všechny úlohy příslušné třídy, které „Metoda CSP“ vyřešila v časovém limitu. Proto jsou třídy ALG a LDA vynechány.

Třída	mzn2fzn	Gecode
BOO	34 %	39 %
GRP	65 %	16 %
LAT	51 %	15 %
NUM	40 %	0 %
RNG	57 %	7 %
ROB	22 %	30 %
TOP	74 %	13 %
Průměr	49 %	17 %

Tabulka 4.3: Podíl trvání procesů *mzn2fzn* a Gecode na celkovém trvání „Metody CSP“

4.3 Komentář výsledků

„Metoda CSP“ uspěla s daným časovým limitem téměř v polovině všech úloh.

Nejllepších výsledků dosáhla ve třídě BOO, kde vyřešila všechny úlohy až na jednu, stejně jako Paradox. Na složitých úlohách BOO058-1 až BOO061-1, kde se doba výpočtu pohybovala v desítkách vteřin, jednoznačně překonala systém Mace4, který dvě z těchto úloh ani nevyřešil.

Rovněž ve třídě GRP vyřešila „Metoda CSP“ většinu úloh a znovu i dvě těžší úlohy, na kterých Mace4 neuspěl. Na druhou stranu Mace4 uspěl na pěti jiných, u nichž „Metodě CSP“ vypršel limit, a v ostatních úlohách byl v průměru o dost rychlejší. Nejvýkonnější však byl jednoznačně Paradox, který většinu problémů

vyřešil v řádu setin vteřiny.

Ve třídě TOP byla navržená metoda poměrně úspěšná, dokonce vyřešila o několik úloh více než Mace4. Ale jiné problémy zase vůbec nezvládla, na rozdíl od Paradoxu, jehož doba výpočtu byla stabilně kratší než 0,3 s.

Zdánlivě nízká rychlost „Metody CSP“ ve třídách NUM, RNG a ROB je způsobena spíše nenáročností testovaných úloh z těchto tříd. Doba jejich řešení se u všech systémů pohybovala v řádu setin sekundy. „Metoda CSP“ ovšem volá během svého postupu několikrát externí programy (*mzn2fzn*, Gecode), což ji může sice nepatrně, ale u jednoduchých úloh zřetelně, zpomalovat.

Špatných výsledků dosáhl vytvořený nástroj ve třídě LAT. Jediné problémy, které vyřešil, byly vesměs triviální. Na těžších úlohách, stejně jako na všech úlohách ze třídy LDA, bylo dosaženo časového limitu bez nalezení modelu. V obou těchto třídách byl naopak nejúspěšnější systém Mace4, který našel řešení všech úloh a ve většině případů i rychleji než systém Paradox.

Na závěr zmiňme problematiku úloh ze třídy ALG. U „Metody CSP“ zde nedošlo k vypršení časového limitu za běhu programu Gecode, nýbrž už při překladu do FlatZincu. Testované úlohy totiž obsahují funkční symboly vysoké arity, v klauzulích se vyskytuje mnoho proměnných a nelze provést dělení klauzulí. Dochází k problémům popsaným v sekci 2.2 a k selhání procesu *mzn2fzn*. Je zajímavé, že Paradox je jediný systém, který tyto úlohy dokáže vyřešit, když by na nich naopak „měl“ selhat jako každý představitel metody MACE.

Motivace k sestavení tabulky (4.3) byly potíže při překladu z MiniZincu do FlatZincu u třídy ALG. Ukázalo se však, že to je obecný problém navržené metody, neboť téměř u všech úloh (u 111 ze 125 vyřešených) trval překlad do FlatZincu déle než samotný výpočet Gecode.

Závěr

Cíle definované v úvodu této bakalářské práce byly dosaženy. MiniZinc se ukázal jako jednoduchý jazyk vhodný pro formulaci FOL úloh, s jedinou vadou – nutností předem definovat velikost hledaného modelu. Podařilo se nalézt účinnější způsob redukce symetrií, než používá Paradox. Vytvořený nástroj je plně funkční a na vybraných úlohách z knihovny TPTP dosahuje lepších výsledků než většina existujících hledačů konečných modelů (zjištěno pomocí on-line aplikace TPTP2T).

Přesto výsledky metody konstrukce modelů pomocí CSP, inspirované metodou převodu na SAT a konkrétně systémem Paradox, nejsou zatím s výkonem Paradoxu srovnatelné. Důvodů je jistě více, ale podle mého názoru spočívá hlavní nedostatek v nutnosti překladač modelu z MiniZincu do FlatZincu. V části 3.1.3 jsem zmínila některé z úprav, které se provádějí automaticky. Například rozvinutí funkce `forall` je nadbytečné, umí-li cílový CSP-solver pracovat s cykly (např. cyklus `for` v C++). Rovněž při převodu polí obecných na jednodimenzionální se ztratí potenciálně užitečná informace a získá se jen mnoho nových podmínek na převod indexů pole.

Bude-li se v budoucnu rozvíjet metoda převodu na CSP, je třeba z ní odstranit překlad do dvou mezijazyků. Možnosti propojení efektivního modelovacího jazyka MiniZinc a výkonného CSP-solveru Gecode se již věnovali Cipriano, Davier, Mauro v [2], kteří vyvinuli nástroj na přímý převod CSP modelů z MiniZincu do Gecode. Možnost převádět FOL úlohy přímo do FlatZincu je zatím neprozkoumaná, ale dá se předpokládat, že nástroj vytvořený pro takto konkrétní účely by byl rychlejší než *mzn2fzn*, který má překládat CSP úlohy nejrůznějšího typu.

Další možnou oblastí budoucího zájmu je podrobnější prozkoumání vlivu prohledávacích anotací a techniky postupného prohledávání. Ta nebyla použita, protože vyžaduje bližší propojení s CSP-solverem, což Gecode (resp. jeho program se vstupem ve FlatZincu) nenabízí. Univerzálnost MiniZincu a FlatZincu by se však dala využít zapojením různých CSP-solverů do procesu hledání. Pokud by se podařilo některý z nich integrovat podobně jako MiniSat v Paradoxu, mohla by se účinnost metody převodu na CSP dostat na úroveň systémů Mace4 a Paradox.

Seznam použité literatury

- [1] BECKET, R. *Specification of FlatZinc: Version 1.3* [online]. 2011 [cit. 2011-05-09]. Dostupné z: <<http://www.g12.cs.mu.oz.au/minizinc/downloads/doc-1.3/flatzinc-spec.pdf>>.
- [2] CIPRIANO, R. – DOVIER, A. – MAURO, J. Compiling and Executing Declarative Modeling Languages to Gecode. In *ICLP '08: Proceedings of the 24th International Conference on Logic Programming*, Udine, Italy. Garcia de la Banda, M. – Pontelli, E. Berlin, Heidelberg: Springer-Verlag, 2008. s. 744–748. ISBN 978-3-540-89981-5.
- [3] CLAESSEN, K. – SÖRENSSON, N. New Techniques that Improve MACE-style Finite Model Finding. In *Proceedings of the CADE-19 Workshop: Model Computation – Principles, Algorithms, Applications on July 29, 2003*, Miami, Florida, USA. Baumgartner, P. – Fermüller, C. 2003. s. 11–27. Dostupné z: <<http://www.cs.miami.edu/~geoff/Conferences/CADE/Archive/CADE-19/WS4/04.pdf>>.
- [4] KAZDA, A. – MARÓTI, M. *Constraint Satisfaction Problem – Lecture Notes* [online]. 2008 [cit. 2011-05-25]. Dostupné z: <<http://atrey.karlin.mff.cuni.cz/~alexak/csp-notes/csp-notes.pdf>>.
- [5] KOROVIN, K. iProver: An Instantiation-Based Theorem Prover for First-Order Logic (System Description). In *IJCAR '08: Proceedings of the 4th international joint conference on Automated Reasoning*, Sydney, Australia. Armando, A. – Baumgartner, P. – Dowek, G. Berlin, Heidelberg: Springer-Verlag, 2008. s. 292–298. ISBN 978-3-540-74969-1.
- [6] KUMAR, Vipin. Algorithms for Constraint-Satisfaction Problems: A Survey. *AI Magazine*. Spring 1992, vol. 13, 1, s. 32–44. ISSN 0738-4602. Dostupné z: <<http://www.aaai.org/ojs/index.php/aimagazine/article/view/976/894>>.

- [7] MARRIOTT, K., et al. *An Introduction to MiniZinc: Version 1.3* [online]. 2011 [cit. 2011-05-05]. Dostupné z: <<http://www.g12.cs.mu.oz.au/minizinc/downloads/doc-1.3/minizinc-tute.pdf>>.
- [8] MCCUNE, W. *Mace4 Reference Manual and Guide*. [online] Tech. Memo ANL/MCS-TM-264, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, August 2003. Dostupné z: <<http://www.cs.unm.edu/~mccune/prover9/mace4.pdf>>.
- [9] NETHERCOTE, N. *Converting MiniZinc to FlatZinc: Version 1.3* [online]. 2011 [cit. 2011-05-09]. Dostupné z: <<http://www.g12.cs.mu.oz.au/minizinc/downloads/doc-1.3/mzn2fzn.pdf>>.
- [10] NETHERCOTE, N., et al. MiniZinc: Towards a standard CP modelling language. In *CP'07: Proceedings of the 13th international conference on Principles and practice of constraint programming*, Providence, RI, USA. Bessiere, Christian. Berlin, Heidelberg: Springer-Verlag, 2007. s. 529–543. ISBN 978-3-540-74969-1, ISSN 0302-9743.
- [11] STUCKEY, P. J. – BECKET, R. – FISHER, J. Philosophy of the MiniZinc challenge. In *Constraints*. July 2010, vol. 15, 3, s. 307–316. ISSN 1383-7133. Dostupné z: <<http://ww2.cs.mu.oz.au/~pjs/papers/challenge.pdf>>.
- [12] SUTCLIFFE, G. *The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0*. Journal of Automated Reasoning. 2009, vol. 43, n. 4, s. 337–362. ISSN 0168-7433.
- [13] WEIDENBACH, Ch. – Gaede, B. – Rock, G. SPASS & FLOTTER Version 0.42. In *Proceedings of International Conference on Automated Deduction (CADE-13), New Brunswick, NJ, USA, July 30 – August 3, 1996*. McRobbie, M. A. – Slaney, J. K. vol. 1104. London, UK: Springer, 1996. s. 141-145. ISBN 3-540-61511-3.
- [14] ZHANG, J. – ZHANG, H. SEM: a system for enumerating models. In *IJCAI'95: Proceedings of the 14th international joint conference on Artificial intelligence – Volume 1*, Montreal, Quebec, Canada. Mellish, C. S. San

Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995. s. 298–303.
ISBN 1-55860-363-8.

Seznam www odkazů

CASC	http://www.cs.miami.edu/~tptp/CASC/
Cluster Sněhurka	http://cluster.karlin.mff.cuni.cz/
Gecode	http://www.gecode.org
MiniSat	http://minisat.se/Papers.html/
MiniZinc and FlatZinc	http://www.g12.cs.mu.oz.au/minizinc/
Otter and Mace2	http://www.cs.unm.edu/~mccune/otter/
Prover9 and Mace4	http://www.cs.unm.edu/~mccune/prover9/
TPTP	http://www.cs.miami.edu/~tptp/

Seznam použitých zkratek

- CADE - The Conference on Automated Deduction
- CASC - The CADE ATP System Competition
- CNF - conjunctive normal form, konjunktivní normální forma
- CSP - constraint satisfaction problem, problém splňování podmínek
- EPR - effectively propositional
- FOL - first-order logic, predikátová logika 1. řádu
- LNH - least-number heuristic, heuristika nejmenšího čísla
- SAT - Boolean satisfiability problem, splnitelnost booleovských formulí
- SEM - System for Enumerating Models
- TPTP - Thousands of Problems for Theorem Provers

Příloha A

Kompletní výsledky testování

Následující tabulka obsahuje všechny hodnoty získané při testování systémů „Metoda CSP“, Paradox a Mace4. Údaje jsou ve vteřinách.

Úloha	Metoda CSP			Paradox	Mace4
	mzn2fzn	Gecode	Celkem		
ALG008-1	0.17	0.02	0.22	0.01	0.07
ALG303-1	fail	-	fail	111.06	fail
ALG304-1	fail	-	fail	67.71	fail
ALG305-1	fail	-	fail	8.20	fail
ALG306-1	fail	-	fail	5.48	fail
ALG307-1	fail	-	fail	9.06	fail
ALG308-1	fail	-	fail	175.56	fail
ALG309-1	fail	-	fail	59.52	fail
ALG310-1	fail	-	fail	74.09	fail
ALG311-1	fail	-	fail	129.68	fail
ALG312-1	fail	-	fail	119.57	fail
ALG313-1	fail	-	fail	84.47	fail
ALG314-1	fail	-	fail	75.40	fail
ALG315-1	fail	-	fail	288.09	fail
ALG316-1	fail	-	fail	24.62	fail
ALG317-1	fail	-	fail	239.00	fail
ALG318-1	fail	-	fail	35.87	fail
ALG319-1	fail	-	fail	40.61	fail
ALG320-1	fail	-	fail	41.06	fail
ALG321-1	fail	-	fail	240.23	fail
ALG322-1	fail	-	fail	fail	fail
ALG323-1	fail	-	fail	38.89	fail
ALG324-1	fail	-	fail	fail	fail
ALG325-1	fail	-	fail	72.38	fail
ALG326-1	fail	-	fail	85.11	fail
ALG327-1	fail	-	fail	38.44	fail
ALG328-1	fail	-	fail	fail	fail
ALG329-1	fail	-	fail	86.56	fail
ALG330-1	fail	-	fail	fail	fail
ALG331-1	fail	-	fail	65.35	fail
ALG332-1	fail	-	fail	59.16	fail
ALG333-1	fail	-	fail	fail	fail
ALG334-1	fail	-	fail	75.87	fail
BOO008-3	0.06	0.01	0.10	0.08	0.00
BOO019-1	0.11	0.02	0.16	0.00	0.00
BOO027-1	0.01	0.00	0.04	0.01	0.00
BOO030-1	0.01	0.00	0.04	0.02	0.00
BOO032-1	0.42	0.11	0.60	0.05	0.00
BOO033-1	0.03	0.00	0.06	0.01	0.00
BOO036-1	0.02	0.00	0.06	0.00	0.00
BOO037-1	0.10	0.01	0.14	0.02	0.00
BOO056-1	0.18	0.35	0.60	0.09	0.10
BOO057-1	0.17	0.52	0.75	0.22	0.12
BOO058-1	0.24	175.66	175.61	0.36	83.33
BOO059-1	0.25	1.91	2.25	1.10	98.50
BOO060-1	0.24	60.24	60.43	4.59	fail
BOO061-1	0.24	3.03	3.38	0.96	fail
BOO066-1	0.26	fail	fail	fail	fail
GRP024-4	0.23	0.13	0.46	0.06	0.66
GRP025-2	2.61	0.38	3.04	0.07	0.13
GRP025-3	0.70	1.28	2.01	0.18	0.07
GRP025-4	2.82	0.40	3.26	0.07	0.12
GRP026-2	0.42	0.06	0.53	0.17	0.01
GRP026-3	0.63	1.68	2.35	0.40	0.18
GRP026-4	0.52	0.06	0.63	0.18	0.02
GRP027-1	0.43	0.06	0.54	0.04	0.02

GRP027-2	0.68	0.09	0.82	0.07	0.02
GRP081-1	0.02	0.02	0.08	0.01	0.00
GRP112-1	0.02	0.00	0.05	0.01	0.00
GRP123-1.005	6.49	0.74	7.30	0.02	0.15
GRP123-2.005	6.54	fail	fail	0.03	0.14
GRP123-3.004	6.70	0.66	7.41	0.04	0.20
GRP123-4.004	1.33	0.14	1.53	0.02	0.03
GRP123-6.005	0.76	0.29	1.12	0.02	0.02
GRP123-7.005	0.81	0.34	1.23	0.02	fail
GRP123-8.004	1.07	0.16	1.30	0.04	0.07
GRP123-9.004	0.33	0.04	0.43	0.02	0.01
GRP124-1.005	6.42	0.74	7.24	0.01	0.16
GRP124-2.005	6.62	0.62	7.31	0.03	0.15
GRP124-3.005	24.90	20.43	45.30	0.04	0.66
GRP124-4.005	6.64	0.70	7.44	0.02	0.16
GRP124-6.005	0.79	0.20	1.06	0.02	0.02
GRP124-7.005	0.77	0.22	1.09	0.03	fail
GRP124-8.005	2.31	fail	fail	0.04	fail
GRP124-9.005	0.91	fail	fail	0.02	fail
GRP125-1.004	0.18	0.02	0.27	0.01	0.00
GRP125-2.004	0.23	0.03	0.31	0.01	0.00
GRP125-3.004	0.72	0.09	0.92	0.03	0.04
GRP125-4.004	0.28	0.04	0.37	0.02	0.00
GRP126-1.005	0.44	0.06	0.59	0.02	0.01
GRP126-2.005	0.52	0.06	0.66	0.02	0.01
GRP126-3.005	1.50	fail	fail	0.02	0.15
GRP126-4.005	0.73	0.10	0.91	0.02	0.01
GRP127-1.005	0.44	0.06	0.57	0.01	0.00
GRP127-2.005	0.52	0.07	0.70	0.03	0.00
GRP127-3.005	1.51	fail	fail	0.04	0.15
GRP127-4.005	0.80	0.10	0.99	0.02	0.02
GRP128-1.004	0.16	0.03	0.27	0.02	0.00
GRP128-2.004	0.20	0.03	0.28	0.02	0.00
GRP128-3.004	0.73	0.35	1.15	0.03	0.35
GRP128-4.004	0.27	0.04	0.35	0.02	0.00
GRP129-1.005	0.42	3.50	4.01	0.02	0.10
GRP129-2.005	0.52	0.08	0.68	0.02	0.22
GRP129-3.005	1.59	fail	fail	0.04	fail
GRP129-4.005	0.70	0.11	0.90	0.02	0.02
GRP130-1.005	0.45	0.21	0.76	0.02	0.04
GRP130-2.005	0.48	0.07	0.65	0.02	0.02
GRP130-3.004	0.72	0.11	0.91	0.03	0.48
GRP130-4.004	0.24	0.04	0.35	0.01	0.01
GRP131-1.005	7.03	fail	fail	0.02	1.30
GRP131-2.005	7.21	fail	fail	0.03	5.56
GRP132-1.005	6.78	fail	fail	0.03	1.03
GRP132-2.005	6.78	fail	fail	0.03	1.40
GRP133-1.004	0.18	0.02	0.26	0.01	0.00
GRP133-2.004	0.22	0.03	0.32	0.02	0.00
GRP134-1.005	0.48	0.16	0.72	0.02	0.05
GRP134-2.005	0.54	0.08	0.69	0.02	0.01
GRP135-1.005	0.47	3.93	4.47	0.02	0.19
GRP135-2.005	0.50	0.08	0.67	0.02	5.04
GRP204-1	0.04	0.01	0.08	0.01	0.00
GRP207-1	0.02	0.00	0.05	0.01	0.00
GRP392-1	0.03	0.00	0.05	0.01	0.00
GRP393-1	0.02	0.00	0.06	0.00	0.00
GRP394-1	0.03	0.00	0.06	0.02	0.00
GRP394-2	0.03	0.00	0.07	0.01	0.00
GRP394-3	0.02	0.00	0.06	0.01	0.00
GRP395-1	0.04	0.01	0.07	0.01	0.00
GRP397-1	0.02	0.00	0.04	0.01	0.00
GRP398-1	0.02	0.00	0.05	0.00	0.00
GRP398-2	0.03	0.00	0.06	0.01	0.00
GRP398-3	0.02	0.00	0.06	0.00	0.00
GRP399-1	4.92	fail	fail	0.02	fail
GRP735-1	1.58	fail	fail	fail	fail
GRP736-1	2.11	fail	fail	fail	fail
GRP737-1	2.12	fail	fail	fail	3.62
GRP738-1	2.49	fail	fail	fail	fail
GRP739-1	15.36	fail	fail	fail	fail
GRP740-1	12.95	fail	fail	fail	fail
GRP741-1	12.97	fail	fail	141.82	fail

GRP742-1	3.01	fail	fail	fail	fail
GRP743-1	8.26	fail	fail	174.36	0.14
GRP744-1	8.24	fail	fail	fail	fail
GRP773-1	2.53	fail	fail	fail	fail
LAT016-1	1.62	fail	fail	1.18	0.06
LAT024-1	0.02	0.00	0.06	0.01	0.00
LAT025-1	0.28	0.15	0.52	0.03	0.00
LAT046-1	0.54	0.14	0.78	0.12	0.00
LAT047-1	0.21	0.08	0.35	0.03	0.00
LAT048-1	0.43	0.11	0.62	0.07	0.00
LAT049-1	1.04	0.65	1.81	0.26	0.01
LAT050-1	2.61	3.10	5.86	2.48	0.08
LAT051-1	0.31	0.08	0.50	0.05	0.00
LAT052-1	3.72	5.64	9.45	0.64	0.04
LAT053-1	5.53	fail	fail	fail	1.40
LAT054-1	5.14	fail	fail	fail	0.54
LAT055-1	0.05	0.01	0.10	0.01	0.00
LAT055-2	0.02	0.00	0.05	0.01	0.00
LAT056-1	0.02	0.00	0.05	0.01	0.00
LAT057-1	0.02	0.00	0.04	0.01	0.00
LAT058-1	0.03	0.00	0.05	0.01	0.00
LAT059-1	0.02	0.00	0.05	0.01	0.00
LAT060-1	0.03	0.00	0.06	0.01	0.00
LAT061-1	0.02	0.00	0.04	0.00	0.00
LAT062-1	4.89	fail	fail	4.93	0.14
LAT063-1	4.89	fail	fail	2.46	0.06
LAT098-1	1.34	fail	fail	3.18	0.11
LAT099-1	1.27	fail	fail	fail	1.84
LAT100-1	1.14	fail	fail	5.60	1.08
LAT101-1	1.16	fail	fail	1.16	0.06
LAT102-1	1.35	fail	fail	21.94	1.11
LAT103-1	1.13	fail	fail	1.45	0.07
LAT104-1	1.27	fail	fail	79.87	0.39
LAT105-1	1.29	fail	fail	16.67	0.35
LAT106-1	1.26	fail	fail	80.56	0.40
LAT107-1	1.23	fail	fail	83.66	0.32
LAT108-1	5.85	fail	fail	206.18	11.46
LAT109-1	5.55	fail	fail	12.47	1.54
LAT110-1	5.56	fail	fail	242.65	12.74
LAT111-1	5.08	fail	fail	20.42	1.80
LAT112-1	5.08	fail	fail	216.83	13.47
LAT113-1	5.07	fail	fail	18.94	1.65
LAT114-1	1.07	fail	fail	1.72	0.08
LAT115-1	1.06	fail	fail	5.12	0.89
LAT116-1	1.04	fail	fail	7.66	0.84
LAT117-1	4.98	fail	fail	37.45	6.74
LAT118-1	4.72	fail	fail	256.00	7.38
LAT119-1	1.09	fail	fail	34.08	0.40
LAT120-1	1.20	fail	fail	1.41	0.08
LAT121-1	1.21	fail	fail	2.93	0.07
LAT122-1	1.30	fail	fail	18.70	0.20
LAT123-1	1.27	fail	fail	104.17	0.22
LAT124-1	5.28	fail	fail	49.15	8.03
LAT125-1	3.27	fail	fail	36.87	9.58
LAT126-1	4.46	fail	fail	102.25	8.43
LAT127-1	1.10	fail	fail	1.56	0.07
LAT128-1	1.14	fail	fail	3.93	0.10
LAT129-1	1.17	fail	fail	2.72	0.08
LAT130-1	1.86	fail	fail	7.22	0.37
LAT131-1	1.10	fail	fail	90.32	2.68
LAT132-1	1.01	fail	fail	297.28	3.11
LAT133-1	1.07	fail	fail	1.71	0.07
LAT134-1	1.13	fail	fail	4.30	0.07
LAT135-1	1.10	fail	fail	7.15	0.51
LAT136-1	1.85	fail	fail	9.64	0.62
LAT137-1	1.07	fail	fail	18.48	0.65
LDA015-1	1.07	fail	fail	17.35	91.19
LDA016-1	1.21	fail	fail	56.46	111.55
LDA017-1	1.09	fail	fail	238.51	97.98
LDA018-1	1.09	fail	fail	153.26	104.02
LDA019-1	1.11	fail	fail	87.27	97.90
LDA020-1	1.04	fail	fail	122.99	91.09
LDA021-1	1.09	fail	fail	85.96	111.05

LDA022-1	1.08	fail	fail	221.77	114.78
LDA023-1	1.06	fail	fail	fail	104.83
LDA024-1	1.05	fail	fail	21.08	95.05
LDA025-1	1.07	fail	fail	53.74	106.75
LDA026-1	1.06	fail	fail	226.62	112.03
LDA027-1	1.08	fail	fail	fail	107.49
LDA028-1	1.06	fail	fail	20.56	97.11
LDA029-1	1.07	fail	fail	52.39	102.47
LDA030-1	1.10	fail	fail	150.08	99.77
LDA031-1	1.07	fail	fail	37.10	100.29
LDA032-1	1.09	fail	fail	223.37	111.87
LDA033-1	1.11	fail	fail	fail	109.07
LDA034-1	1.21	fail	fail	56.12	109.85
LDA035-1	1.12	fail	fail	235.56	97.95
LDA036-1	1.16	fail	fail	177.23	111.78
LDA037-1	1.11	fail	fail	86.60	97.58
LDA038-1	1.08	fail	fail	121.80	89.54
NUM285-1	0.02	0.00	0.05	0.01	0.00
NUM286-1	0.01	0.00	0.04	0.00	0.00
NUM286-2	0.03	0.00	0.06	0.00	0.00
NUM287-1	0.02	0.00	0.04	0.01	0.00
NUM288-1	0.02	0.00	0.06	0.01	0.00
RNG007-5	0.09	0.01	0.12	0.02	0.00
RNG025-8	0.13	0.04	0.21	0.03	0.00
RNG025-9	0.19	0.05	0.28	0.04	0.00
RNG042-1	0.09	0.01	0.14	0.02	0.00
RNG042-2	0.02	0.00	0.06	0.01	0.00
RNG042-3	0.03	0.00	0.05	0.01	0.00
RNG043-1	0.03	0.00	0.06	0.02	0.00
RNG043-2	0.03	0.00	0.07	0.01	0.00
ROB012-1	0.11	5.38	5.53	0.06	11.50
ROB012-2	0.14	10.02	10.20	0.03	7.94
ROB015-1	0.02	0.00	0.06	0.02	0.00
ROB028-1	0.02	0.00	0.06	0.00	0.00
ROB029-1	0.02	0.00	0.05	0.01	0.00
TOP001-1	0.29	0.05	0.39	0.13	0.01
TOP002-1	0.29	fail	fail	0.13	0.01
TOP003-2	0.03	0.00	0.05	0.01	0.00
TOP005-1	0.29	fail	fail	0.29	fail
TOP006-1	0.30	fail	fail	0.13	0.32
TOP007-1	1.89	0.48	2.38	0.23	0.06
TOP008-1	0.30	0.05	0.41	0.12	fail
TOP009-1	1.84	0.46	2.34	0.28	fail
TOP010-1	0.29	0.05	0.38	0.12	fail
TOP011-1	0.29	0.06	0.40	0.10	0.02
TOP012-1	0.30	fail	fail	0.14	0.01
TOP013-1	0.30	0.05	0.41	0.12	fail
TOP014-1	0.28	fail	fail	0.13	fail
TOP015-1	0.28	0.05	0.38	0.13	fail
TOP016-1	0.28	0.05	0.38	0.13	fail
TOP017-1	0.28	0.05	0.38	0.13	fail
TOP018-1	0.29	0.05	0.39	0.12	fail
TOP019-1	0.29	fail	fail	0.14	fail