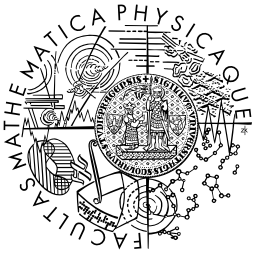

MATEMATICKO-FYZIKÁLNÍ
FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Václav Luňák

STP řešič pro OpenSMT

Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: doc. RNDr. Jan Kofroň, Ph.D.

Studijní program: Informatika

Studijní obor: Obecná informatika

Praha 2020

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Tato práce nebyla využita k získání jiného nebo stejného titulu.

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 má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ....

Podpis autora

Poděkování.

Název práce: STP řešič pro OpenSMT

Autor: Václav Luňák

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: doc. RNDr. Jan Kofroň, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Abstrakt.

Klíčová slova: klíčová slova

Title: STP solver for OpenSMT

Author: Václav Luňák

Department: Department of distributed and dependable systems

Supervisor: doc. RNDr. Jan Kofroň, Ph.D., Department of distributed and dependable systems

Abstract: Abstract.

Keywords: key words

Obsah

Úvod

1. Analýza

1.1 Fungování SMT řešičů

Problém splnitelnosti booleovské formule, označovaný též zkratkou SAT, patří k nejznámějším problémům z oboru matematické logiky. V roce 1971 se stal prvním dokazatelně NP-úplným problémem [2] a v důsledku toho i užitečným nástrojem pro teorii složitosti, pomocí něhož lze rozhodovat o NP-úplnosti dalších problémů.[1]

V praktickém použití však SAT naráží na své limitované vyjadřovací schopnosti. Práce s binárními proměnnými, omezenými pouze na dvě různé hodnoty, může komplikovat převod reálného problému do tvaru booleovské formule. Potřeba užití komplexnějších atomů tak vedla ke zobecnění SAT zvanému *Satisfiability modulo theories* (SMT).

Jak název napovídá, SMT rozšiřuje SAT o jazyk logických teorií. Máme-li nějakou teorii T, pak instancí SMT rozumíme formuli jazyka této teorie. Jiným pohledem můžeme na instanci SMT nahlížet jako na booleovskou formuli, ve které jsme nahradili některé binární proměnné za predikáty obsažené v T. Problému vztaženému k této konkrétní teorii pak říkáme *SMT s ohledem na T*.

Jako příklady teorií tradičně řešených v SMT lze uvézt např. teorii lineární aritmetiky (LA), teorii neinterpretovaných funkcí s rovností (EUF), či teorii diferenční logiky (DL), kterou se budeme zabývat ve zbytku práce.

Vzhledem k podobnostem mezi SAT a SMT není překvapením, že SMT řešiče využívají schopností SAT řešičů. Přechod mezi rozhraním termů teorie a kapacitami SAT řešiče zpravidla probíhá jedním ze dvou základních způsobů [**?**].

První přístup se nazývá *hladový*. Hladové SMT řešiče operují ve dvou krocích. V prvním převedou celou vstupní formuli na ekvisplnitelnou booleovskou formuli. Druhý krok pak již spočívá jen v předání této formule existujícímu SAT řešiči. Pokud bychom tedy pracovali například s aritmetikou nad osmibitovými čísly, mohli bychom reprezentovat každou proměnnou osmi binárními proměnnými a aritmetické operace převézt na odpovídající sekvence logických operací.

Nespornou výhodou hladových řešičů je možnost použití již existujících metod verifikace implementovaných na řešení SAT. Pro některé teorie se také hladové řešiče ukazují být rychlejší než jejich alternativy. Jejich největší problém pak obecně spočívá v překladu literálů teorie do booleovských formulí. Ten musí být zkonstruován samostatně pro každou teorii, a navíc může v závislosti na teorii produkovat formule znatelně delší, než byl původní vstup. Efektivní převody existují např. pro EUF s omezenou doménou [11], obecně však hladový přístup není příliš rozšířený.

Jeho protějškem je takzvaný *líný přístup*. Líný přístup se nesnaží měnit strukturu vstupní formule; namísto toho je každý predikát abstrahován pomocí nové binární proměnné. Když pak SAT řešič rozhodne o ohodnocení těchto proměnných, oznámí toto rozhodnutí *theory solveru* pro danou teorii. Theory solver je schopen určit, zda je dané ohodnocení správné, tzn. dokáže rozhodnout o splnitelnosti nějaké konjunkce literálů teorie. SAT řešič potom hledá platná ohodnocení, dokud nenajde takové, které theory solver prohlásí za konzistentní.

Ve prospěch líných SMT řešičů svědší fakt, že pro danou teorii často existují dobře známé postupy na ověření konjunkce literálů. Pro LA například můžeme využít metod

[1] Příklady převodů NP-úplných problémů na SAT můžeme nalézt např. v [7, kapitola 19]

lineárního programování, theory solver pro DL řeší STP (viz. 1.2) a podobně. Mohou však ztrácet efektivitu zejména v důsledku *slepého prohledávání* [8], kde hlavní řešič rozhoduje o hodnotě predikátů, aniž by a priori věděl o důsledcích těchto ohodnocení v rámci teorie, což může vést k nutnosti vyzkoušet velké množství ohodnocení, než je nalezeno nějaké, které je s teorií konzistentní.

V roce 2004 navrhli Gazinger a kol. nový přístup zvaný *DPLL(T)* [6]. DPLL(T) má koncepčně blíže k línému vyhodnocování, integruje však těsněji hlavní řešič s theory solverem. Místo toho, aby využíval theory solveru až po nalezení nějakého ohodnocení, průběžně mu oznamuje dosavadní rozhodnutí a periodicky se ho ptá na splnitelnost právě dosazené konjunkce. Theory solver pak kromě kontroly splnitelnosti také oznamuje hlavnímu řešiči důsledky této konjunkce. Tím jsme schopni dříve opustit větve rozhodovacího stromu nekonzistentní s teorií.

V jádru tohoto přístupu stojí všeobecný DPLL(X) engine, využívající DPLL [3] postupu pro SAT řešiče. Tento engine nemusí mít žádné znalosti o konkrétní teorii. Dosazením theory solveru $Solver_T$ pro danou teorii T za parametr X pak vytvoříme konkrétní instanci DPLL(T). Hlavní engine komunikuje se $Solver_T$ pomocí následujícího rozhranní [6]:

Initialize(L: množina literálů). Inicializuje $Solver_T$ s L jakožto množinou literálů, které se vyskytují v problému.

SetTrue(l: L-literál): množina L-literálů. Skončí výjimkou, pokud se l ukáže jako nekonzistentní s dosud zadanými literály teorie. V opačném případě přidá l do seznamu zadaných literálů a vrátí množinu L-literálů, které jsou důsledky přidání l do tohoto seznamu.

IsTrue?(l: L-literál): boolean. Vrátí *true* právě tehdy, když l je důsledkem seznamu přidaných literálů. *false* tedy vrací, pokud je důsledkem tohoto seznamu $\neg l$, nebo pokud z něj nevyplývá ani l, ani $\neg l$.

Backtrack(n: přir. číslo). Odstraní posledních n hodnot ze seznamu zadaných literálů. n nesmí být větší než velikost tohoto seznamu.

Explain(l: L-literál): množina L-literálů. Vrátí pokud možno co nejmenší podmnožinu zadaných literálů, z jejichž konjunkce plyne l. Pro l musí platit, že je důsledkem nějaké takové podmnožiny, tedy musí být obsažen v návratové hodnotě nějakého volání **SetTrue(l')** takového, že l' nebylo zahozeno žádným následným voláním **Backtrack**.

Při použití tohoto rozhranní přitom $Solver_T$ nemusí nic vědět o implementaci DPLL(X) enginu. Framework je tedy velice modulární a snadno rozšiřitelný o nové teorie. Vyžadujeme pouze, aby byl $Solver_T$ schopný inkrementálně přijímat a odebírat jednotlivé predikáty teorie. Tento postup se v praxi ukazuje jako efektivnější než dostupné alternativy. Většina dnes rozšířených SMT řešičů — včetně námi používaného OpenSMT2 — je tedy založena na metodě DPLL(T).

1.2 Rozbor STP

Jedním ze základních podproblémů vyskytujícím se v takřka všech plánovacích problémech je takzvaný Simple Temporal Problem (STP). STP poprvé postulovali v roce

1991 Dechter, Meiri a Pearl [4] a od té doby našel široká využití jak v informatických oblastech, tak v oborech od medicíny [1] po vesmírný let [5].

Vstupem STP je množina rozdílových omezení, to jest nerovnic tvaru

$$x - y \leq c,$$

kde x a y jsou proměnné a c je konstanta. V závislosti na tom, jakou verzi problému řešíme, přitom pracujeme buď s celočíselnými, nebo s reálnými hodnotami. Výstupem tohoto problému je pak rozhodnutí, zda existuje ohodnocení proměnných tak, aby byla splněna všechna zadaná omezení. V rozšíření problému pak můžeme požadovat na výstupu i nějaké takovéto splnitelné ohodnocení, pokud existuje, případně nalezení pokud možno co nejmenší podmnožiny omezení, která zajišťuje nesplnitelnost problému.

Na první pohled se může zdát pevně daný tvar nerovnic příliš omezující, uvědomme si však, že do této formy můžeme převést několik dalších druhů nerovnic. Nejsnáze zahrneme do problému omezení tvaru $x - y = c$; ty stačí jednoduše nahradit nerovnicemi $x - y \leq c$ a $x - y \geq c$.

Problematické nejsou ani nerovnice typu $\pm x \leq c$. Pro účely takovýchto omezení si zavedeme novou globální proměnnou $zero$, s jejíž pomocí převedeme předchozí do tvaru $x - zero \leq c$, respektive $zero - x \leq c$. Pokud pak hledáme splňující ohodnocení proměnných, najdeme takové, kde $zero$ je ohodnoceno nulou. Korektnost tohoto postupu zaručuje následující zjevnétvrzení.

Tvrzení 1. *Je-li σ splňující ohodnocení nějakého STP a ε libovolná konstanta, pak ohodnocení π definované pro všechny proměnné x jako $\pi(x) = \sigma(x) + \varepsilon$ je také splňující ohodnocení tohoto STP.*

Důkaz. Plyne okamžitě z tvaru rozdílových omezení. □

Můžeme do problému zahrnout taktéž omezení tvaru $x - y < c$. Pro celočíselné proměnné lze tuto nerovnici ekvivalentně zapsat jako $x - y \leq c - 1$. V reálné variantě pak nahradíme nerovnici výrazem $x - y \leq c - \delta$, přičemž nenastavujeme okamžitě konkrétní hodnotu δ, ale udržujeme si ji pouze symbolicky a určujeme její vhodné dosazení až při výpočtu splňujícího ohodnocení. Tento postup je detailněji popsán v sekci 2.5. Uvědomme si, že pokud jsme schopni vyjádřit ostré nerovnosti, umíme vyjádřit i negace neostrých nerovností a naopak.

V jazyce výrokové logiky pak teorii obsahující výše popsané nerovnice nazveme *teorie diferenční logiky* a budeme ji značit *DL*. Celočíselnou variantu této teorie pak budeme značit jako *IDL* a reálnou variantu jako *RDL*. Nahradíme-li pak v booleovské formuli některé termy těmito nerovnicemi, ověření splnitelnosti takto vzniklé formule je instancí SMT problému s ohledem na DL.

1.3 Převod na grafový problém

Velkou rozšířenost STP můžeme mimo jiné přisoudit tomu, že jsme schopni ho efektivně řešit. Jelikož se problém skládá výlučně z lineárních omezení, mohli bychom na první pohled využít metod lineárního programování, jako je například simplexový algoritmus. Tyto metody jsou schopny řešit i mnohem komplexnější problémy, avšak s jejich výpočetní silou se pojí znatelně vyšší časová náročnost. Algoritmy specializované na STP se proto už od svého počátku [4, Kapitola 2] obracejí jiným směrem,

a to k formalizmu teorie grafů. Přestože v průběhu let vzikly různé metody řešení tohoto problému, všechny fungují na základě převedení množiny omezení na takzvaný *omezující graf*.

Definice 1 (Omezující graf)**.** *Nechť* Π *je množina rozdílových omezení. Omezujícím grafem této množiny rozumíme hranově ohodnocený orientovaný graf* G *takový, že vrcholy* G *tvoří proměnné vyskytující se v* Π *a každému omezení* $(x - y \leq c) \in \Pi$ *odpovídá v* G *hrana* $\langle x,y \rangle$ *s ohodnocením* c*.*

Poznámka. Hranu $\langle x,y \rangle$ s ohodnocením c budeme značit $x \xrightarrow{c} y$. Orientovanou cestu z x do y se součtem ohodnocení k pak budeme značit $x \xrightarrow{k*} y$.

Pro úplnost dodejme, že dvojice proměnných se může vyskytovat v libovolně mnoha omezeních. Omezující graf je tedy formálně orientovaným multigrafem. Vzhledem k vzájemné bijekci mezi hranami grafu a nerovnicemi problému budeme v průběhu práce volně přecházet mezi oběma reprezentacemi.

Převod do formy grafu je pro řešení problému zásadní. Umožňuje nám totiž formulovat následující klíčové tvrzení.

Tvrzení 2 (Dechter a kol. [4])**.** *Nechť* Π *je množina rozdílových omezení. Instance STP tvořená touto množinou je splnitelná právě tehdy, když omezující graf* Π *neobsahuje záporné cykly.*

Důkaz. Najdeme-li v omezujícím grafu záporný cyklus obsahující vrchol x, sečtením všech nerovnic vyskytujících se v tomto cyklu dostaneme $x - x \leq c < 0$, z čehož je jasně problém nesplnitelný. Je-li na druhou stranu problém nesplnitelný, obsahuje Π nějakou nerovnici $x - y \leq c$ takovou, že z Π vyplývá $y - x < -c$. Tato implikace znamená, že v omezujícím grafu existuje cesta $y \xrightarrow{k*} x$ taková, že $k < -c$. Hrana $x \xrightarrow{c} y$ pak společně s touto cestou tvoří záporný cyklus. □

Hledání splnitelnosti STP jsme tedy schopni převést na hledání záporného cyklu v grafu. To je problém, který dokážeme efektivně řešit. Využít můžeme např. některý algoritmus na hledání nejkratší cesty, kupříkladu Floydův-Warshallův algoritmus operující v čase $\Theta(|V|^3)$ nebo Bellmanův-Fordův algoritmus, který má časovou složitost $\Theta(|V| \cdot |E|)$.

Tyto algoritmy však trpí pro náš účel zásadním nedostatkem. Jejich použití znamená, že po každém přidání nové hrany do grafu musí znovu proběhnout celé prohledávání. Tento postup není vhodný pro použití v SMT řešičích, ve kterých je kladen velký důraz na inkrementalitu. V následující sekci tedy podrobně rozebereme několik postupů pro řešení problémů SMT s ohledem na DL a motivujeme výběr námi použitého algoritmu.

1.4 Volba algoritmu

2. Popis řešení

2.1 Prostředí

Požadavky na použité prostředí jsou určeny převážně požadavky frameworku OpenSMT, pod nějž tato práce spadá. OpenSMT — a tudíž i tento projekt — je programován v jazyce C++, konkrétně ve verzi C++11. Práce byla vyvíjena a testována na operačním systému s linuxovým jádrem nad architekturou x64. Jelikož si nejsme vědomi toho, že bychom použili vlastnosti jazyka specifické pro tuto konfiguraci, věříme, že náš kód bude možné bez větších potíží zprovoznit i na jiných platformách a operačních systémech.

2.2 Úpravy referenčního algoritmu

Naše implementace je převážně založená na algoritmu tak, jak ho postulovali Nieuwenhuis a Oliveras [9]. Přesto se liší v některých implementačních detailech, daných zejména odlišnostmi OpenSMT od referenčního DPLL(T) frameworku.

Prvním důležitým rozdílem je způsob abstrakce literálů teorie. V OpenSMT neexistuje způsob, kterým bychom mohli informovat hlavní engine o vztazích mezi různými nerovnicemi. Obsahuje-li například vstupní formule nerovnice $(x - y \leq 3)$ a $(y - x < -3)$, algoritmus popsaný v [9] je pro DPLL(X) abstrahuje do booleovských symbolů p a $\neg p$. Této abstrakce nejsme pomocí rozhranní v OpenSMT schopni. Náš řešič si tedy musí tyto vztahy udržovat interně.

S tímto omezením úzce souvisí druhý zásadní rozdíl. Můžeme si všimnout, že schéma uvedené v sekci 1.1 umožňuje frameworku ohodnotit literál pouze jako *true*. Uvědomme si, že to obecně není nijak omezující. Ohodnocení termu p jako *false* totiž můžeme snadno zařídit voláním **SetTrue(**$\neg p$**)**. Protože však OpenSMT nezná tato mapování mezi termy a jejich negacemi, není pro nás tento přístup validní. Namísto toho využíváme přímočařejší implementace, kde termům můžeme přiřadit ohodnocení *true* i *false*. Tato odlišnost vyžaduje několik modifikací našeho řešiče.

Předně musíme být schopni detekovat, zda jedna nerovnice neodpovídá negaci druhé, jak už jsme uvedli výše. Jakmile jsme toho schopni, můžeme záporné ohodnocení hrany implementovat jako přidání negace této hrany do omezujícího grafu. Jak ale postupovat, pokud jsme u některé hrany tuto negaci nenašli? Naivní přístup by byl vytvořit negaci takové hrany ve chvíli, kdy se objeví její záporné ohodnocení. Takové řešení však není možné. Uvažujme neohodnocenou hranu h, která ve vstupní formuli nemá svou negaci. Mějme v omezujícím grafu nějakou množinu hran M takovou, že $M \cup \{h\}$ tvoří záporný cyklus. Pokud M existuje, očividně je $\neg h$ důsledkem tohoto grafu. Jelikož se ale $\neg h$ nevyskytuje ve vstupní formuli a h nebyla nikdy záporně ohodnocená, nevyskytuje se $\neg h$ v seznamu možných hran našeho řešiče. Algoritmus hledání důsledků ji tudíž nemůže nalézt. Kladným ohodnocením h potom vytvoříme záporný cyklus v omezujícím grafu, čímž porušíme invariant našeho algoritmu a nekonečně zacyklíme příští hledání důsledků.

Jak vidíme, negace všech hran musí být známy předtím, než proběhne prohledávání grafu. Tento problém jsme se tedy rozhodli vyřešit už při oznamování možných literálů. Když je řešiči předán literál vyskytující se ve formuli, vytvoříme nejen hranu odpovídající tomuto literálu, ale okamžitě i hranu odpovídající její negaci. Při předávání

dalších literálů je pak jen třeba ověřit, zda neodpovídají některé již vytvořené hraně. Podrobněji tento postup popisujeme v sekci 3.1.

Posledním větším rozdílem je způsob zpracování konfliktů. Jedním z důsledků vyčerpávající propagace je fakt, že řešič nemusí kontrolovat nesplnitelnost předaného ohodnocení. Zapříčinilo-li by přidání nějaké hrany spor, negace této hrany je důsledkem omezujícího grafu. Můžeme přitom předpokládat, že DPLL(X) negaci objeveného důsledku nikdy řešiči nepředá. OpenSMT se v tomto ohledu liší ve způsobu, jakým analyzuje sporný stav. Jeho řešiče obecně nepodporují operaci **Explain**, vracející pro nějaký důsledek množinu jeho příčin. Místo toho implementují funkci **getConflict**, která hledá nesplnitelnou množinu literálů. Pro použití této funkce, nutné k určení úrovně backtrackingu, se ale nejdřív musí řešič dostat do nekonzistentního stavu.

Tyto dva přístupy jsou naštěstí ekvivalentní. Když OpenSMT objeví spor, předá řešiči nějaké zaručeně sporné ohodnocení, čímž ho dostane do nekonzistentního stavu. Z tohoto důvodu musíme před každým přidáním hrany kontrolovat, zda není sporná se stávajícím ohodnocením (více viz. 3.1). Náš řešič si zapamatuje hranu h odpovídající tomuto ohodnocení a funkce **getConflict** pak odpovídá volání **Explain(**$\neg h$**)**.

2.3 Datové struktury

Základní datovou strukturou v OpenSMT je `Pterm`, reprezentující jeden term vyskytující se ve vstupní formuli. Odkazy na tyto termy jsou pak předávány pomocí referenční struktury `PTRef`. Ta obsahuje pouze numerický identifikátor použitý k jejímu rozlišení. Mapování jednotlivých referencí na odpovídající termy přitom zařizuje třída `Logic`, respektive její potomci. `Logic` si ukládá převodní tabulku párující `PTRef` a jejich odpovídající `Pterm` a poskytuje také mechanismy pro vytváření nových termů. Její potomci pak rozšiřují tyto mechanismy o možnosti odpovídající dané teorii, např. s `LALogic` jsme schopni vytvářet termy odpovídající nerovnicím z lineární aritmetiky. Jelikož rozdílová omezení jsou speficickým tvarem lineárních nerovnic, využívá náš řešič právě schopností `LALogic`, konkrétně `LIALogic` pro implementovanou celočíselnou verzi.

Ohodnocení proměnných je uloženo ve struktuře `PtAsgn`, která obsahuje `PTRef` odpovídající ohodnocené proměnné a `lbool` označující její ohodnocení (`lbool` je běžný optional boolean).

Samotné termy mají stromovou strukturu. Pokud se nejedná o atomickou proměnnou, reprezentuje term nějaký n-ární funkční či relační symbol společně s jeho argumenty. K rozlišení typu symbolu nám opět poslouží API třídy `Logic`, pro přístup k argumentům je pak použit operátor `[]`. Reprezentuje-li například `Pterm p` term $(x \vee y)$, budeme mít přístup k proměnným `p[0]` a `p[1]`, což jsou `PTRef` reprezentující x, respektive y.

Nejdůležitější datovou strukturou našeho řešiče je `Edge`. V té jsou uloženy informace o jedné hraně omezujícího grafu. Konkrétně tedy obsahuje reference na její vstupní a výstupní vrchol, ohodnocení, odkaz na svou negaci a informaci o tom, kdy byla přidána do omezujícího grafu. Po vzoru frameworku přitom zbytek řešiče nepracuje přímo s těmito strukturami, ale s jejich referencemi `EdgeRef`, které opět obsahují pouze jednoznačný numerický identifikátor. Samotné hrany se pak nacházejí jen v centrálním úložišti, které tvoří třída `STPStore`. Ta zařizuje zejména tvorbu nových hran a převod z `EdgeRef` na `Edge&`. Použit je i protějšek k `EdgeRef` pro vrcholy, struktura `VertexRef`. Jelikož se však s vrcholy nepojí žádná informace, nejedná se o odkaz

na další strukturu, ale pouze o symbolické reference, sloužící pro vzájemné rozlišení jednotlivých vrcholů.

Jelikož náš řešič dostává od frameworku informace o proměnných zásadně jako `PTRef`, potřebujeme způsob, jak přecházet mezi reprezentací frameworku a interní reprezentací našeho řešiče. K tomuto účelu slouží třída `STPMapper`. V této třídě se vyskytuje hned několik druhů převodních tabulek. Pamatuje si převod z `PTRef` na `VertexRef`, přiřazující proměnné k vrcholům grafu, a převod z `PTRef` na `EdgeRef`, přiřazující nerovnice k hranám. Tyto převody jsou zásadní pro interpretaci příkazů frameworku. Pro hrany si pamatuje i opačný převod, mapující `EdgeRef` zpět na odpovídající `PTRef`. Ten je důležitý pro oznámení nalezených dedukcí (viz. 3.2). Pro účely oznámení nesplnitelné množiny literálů si pro hrany právě v grafu pamatujeme i mapu z `EdgeRef` na `PtAsgn`, které způsobily jejich přidání do grafu. Uvědomme si, že tento převod není zaměnitelný s předchozím převodem na `PTRef`, jelikož hrana se může vyskytnout v grafu z důvodu záporného ohodnocení její negace. `STPMapper` si navíc pamatuje pro každý vrchol seznam všech hran, ve kterých se daný vrchol vyskytuje, jak je popsáno v 1.4.

Samotný omezující graf ukládáme do struktury `STPEdgeGraph`. Ta obsahuje seznam přidaných hran a oboustranný seznam sousedů pro všechny vrcholy grafu. S grafem přímo manipuluje třída `STPGraphManager`, která působí jako hlavní výpočetní třída řešiče. Provádí přidávání hran do grafu a jejich případné odebírání z grafu, ale i hledání důsledků přidané hrany a hledání vysvětlení nalezeného důsledku.

Třídu `STPModel` využijeme, pokud chceme pro splnitelnou množinu nerovnic najít nějaké ohodnocení proměnných. `STPModel` dostane kopii grafu, ze které vytvoří mapu ohodnocení obsažených vrcholů.

Všechny struktury řešiče spojuje dohromady hlavní třída `STPSolver`. Jakožto potomek `TSolver` implementuje tato třída rozhranní mezi řešičem a zbytkem frameworku.

Je vhodné zmínit, že za dobu vývoje OpenSMT v něm vzniklo několik implementací základních datových struktur. Hojně užívaným příkladem je třída `vec`, reprezentující běžný vektor. Vyjma malých rozdílů API a údajné vyšší efektivity na primitivních typech se tyto třídy výrazně neliší od implementací ze standardní knihovny. Konkrétně třída `vec` je navíc omezena skutečností, že jejími prvky nemohou být typy obsahující odkazy (vyjímkou z tohoto pravidla je zvlášť implementovaný `vec⟨vec⟨T⟩⟩`). Za účelem konzistence se zbytkem frameworku jsme se přesto rozhodli využívat tyto lokální struktury všude, kde je to možné.

2.4 Popis běhu programu

V první fázi algoritmu jsou řešiči předány všechny literály teorie, které se vyskytují ve vstupní formuli. Řešič z nich nejprve extrahuje relevantní hodnoty. Následně zkontroluje, zda se nejedná o negaci některého z již zapamatovaných literálů. Pokud ano, pouze tuto negaci explicitně označí. V opačném případě vytvoří hranu odpovídající tomuto literálu a zároveň a priori hranu tvořící negaci tohoto literálu, jak je popsáno v sekci 2.2. Obě nové hrany se stanou součástí úložiště a jsou zařazeny do překladových tabulek. Zbytek výpočtu pak již může předpokládat, že pracujeme pouze se známými literály, pro něž máme vytvořené hrany.

Poté, co jsou všechna omezení načtena, nastává hlavní část programu. Během té postupně řešič dostává rozhodnutá ohodnocení literálů. Když nějaké obdrží, přidá

do omezujícího grafu hranu odpovídající tomuto ohodnocení. Následně proběhne prohledávání objevující všechny důsledky tohoto ohodnocení. Tyto důsledky jsou přeloženy zpět do formy `PtAsgn` a oznámeny frameworku.

Po nějaké sekvenci těchto ohodnocení buď nalezneme splňující ohodnocení celé formule, nebo se dostaneme do sporu. Spor můžeme rozpoznat tak, že se chystáme přidat do grafu hranu, jejíž negace byla buď dříve do grafu přidána, nebo byla nalezena jako důsledek dřívějšího ohodnocení. V takovém případě řešič oznámí selhání tohoto ohodnocení a přejde do chybového stavu. Jakmile je řešič v chybovém stavu, automaticky zamítá všechna nová ohodnocení. V této situaci následuje nalezení nesplnitelné množiny. Pokud byla objevená negace hrany explicitně přidána do grafu, je tato množina triviální. Jinak ji získáme modifikovanou verzí prohledání grafu (viz. 1.4). Jakmile je nesplnitelná množina nalezena a předána frameworku, rozhodne se podle ní úroveň backtrackingu. Ten je v OpenSMT řešen obecně pomocí systému záchytných bodů. Řešič může být v libovolnou chvíli požádán, aby uložil svůj aktuální stav na zásobník. V případě nutnosti je mu pak sděleno, aby odebral několik bodů z vrchu tohoto zásobníku a tím se vrátil do dřívějšího stavu.

Jakmile se řešič vrátí do konzistentního stavu, tento proces se opakuje, dokud není nalezeno nějaké splnitelné ohodnocení, nebo dokud framework nevyčerpá všechny možnosti ohodnocení. Splnitelnost formule je určena tím, který z těchto dvou případů nastane. V případě, že je formule splnitelná, může být řešič na závěr výpočtu ještě požádán, aby vytvořil její model, tzn. nalezl konzistentní hodnoty pro všechny proměnné obsažené v literálech teorie.

2.5 Srovnání reálné a celočíselné verze

Algoritmus uvedený v sekci 1.4 je s drobnými úpravami použitelný jak pro RDL, tak pro IDL. Přestože v této práci implementujeme pouze celočíselnou verzi problému, přišlo nám názorné zamyslet se nad rozdíly mezi oběma variantami.

Uveďme nejprve pro zajímavost, že pokud na vstupu podporujeme i rovnice a jejich negace, ověření konzistence je možné v reálných číslech provést polynomiálně, zatímco pro celožíselný obor je ověření NP-těžké — jsme schopni na něj převést např. problém k-barevnosti grafu [10].

Prvním zjevným rozdílem pro potřeby naší implementace je reprezentace čísel. Reálná čísla je potřeba reprezentovat jinak, než čísla celá. OpenSMT definuje vlastní datový typ pro reálná čísla nazvaný `FastRational`. V různých částech kódu jej můžeme najít také pod aliasy `Real` nebo `Number`. Tento typ má několik výhod oproti běžným primitivním typům s posuvnou desetinnou čárkou. Především se jedná o typ s teoreticky neomezenou velikostí. Jelikož využívá struktur větších než jedno procesorové slovo, není limitován kapacitami procesoru. Důsledkem toho je to také typ s libovolnou přesností. Netrpí tak zaokrouhlovacími chybami a ztrátou platných číslic u velkých hodnot jako např. `float` a `double`.

Samozřejmě bychom mohli `FastRational` použít i v celočíselném řešení. Usnadnilo by nám to návrh datových struktur a umožnilo větší znovupoužitelnost kódu. Testováním se však ukázalo, že aritmetické operace na `FastRational` jsou citelně pomalejší než u primitivních typů. Náš projekt tak používá primitivní celočíselný typ, konkrétně typ `ptrdiff_t`. Ten bohužel podporuje pouze hodnoty omezeného rozsahu. Protože ale v OpenSMT zatím neexistuje ekvivalent `FastRational` pro celá čísla, uznali jsme jej jako nejlepší volbu. Omezení rozsahu se navíc experimentálně

ukázalo jako zanedbatelné pro běžné použití — z 834(?) testů knihovny SMT-LIB náš řešič vrátil ve všech případech správný výsledek.

Zásadní algoritmický rozdíl je také v tvorbě negací. Máme-li například na vstupu nerovnici $(x - y < k)$, v IDL ji triviálně převedeme do tvaru $(x - y \leq k - 1)$. V RDL se ovšem ostrých nerovností tak snadno nezbavíme. Nieuwenhuis a Olivieras [9] navrhují pro tento případ postup, který postuloval Schrijver [12]. Ten je založen na zápisu nerovnosti jako $(x - y \leq k - \delta)$ pro nějaké dostatečně malé δ, které bude záviset pouze na ostatních přítomných nerovnicích. Hodnotu δ přitom nepočítáme přímo, ale ukládáme si ji pouze symbolicky. (Tohle vysvětlení bude úplnější, až pořádně pochopím jak to funguje) V kontextu OpenSMT už je tento přístup použit v řešiči teorie lineární aritmetiky.

Na závěr ještě připomeňme omezení týkající se datových struktur použitých v OpenSMT. Jelikož `FastRational` obsahuje ukazatele, nelze jej — ani typy, které ho obsahují — použít například jako prvek třídy `vec`. Na tato omezení je třeba dbát při přechodu z celočíselné verze na reálnou.

3. Programátorská dokumentace

3.1 Přidávání literálů

3.2 Hledání důsledků

3.3 Rozhodování o splnitelnosti

3.4 Hledání konfliktů a backtracking

3.5 Nalezení splňujícího ohodnocení

4. Experimentální měření

4.1 Metodologie

4.2 Výsledky

4.3 Srovnání

Závěr

Seznam použité literatury

[1] Anselma, L., Terenziani, P., Montani, S. a Bottrighi, A. Towards a comprehensive treatment of repetitions, periodicity and temporal constraints in clinical guidelines. *Artificial Intelligence In Medicine*, **38**(2):171 – 195, 2006.

[2] Cook, S. A. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, strana 151–158, New York, NY, USA, 1971. Association for Computing Machinery.

[3] Davis, M. a Putnam, H. A computing procedure for quantification theory. *J. ACM*, **7**(3):201–215, July 1960.

[4] Dechter, R., Meiri, I. a Pearl, J. Temporal constraint networks. *Artificial Intelligence*, **49**(1-3):61–95, 1991.

[5] Fukunaga, A., Rabideau, G., Chien, S. a Yan, D. Towards an application framework for automated planning and scheduling. *1997 IEEE Aerospace Conference, Aerospace Conference, 1997. Proceedings., IEEE*, **1**:375, 1997.

[6] Harald, G., George, H., Robert, N., Albert, O. a Cesare, T. Dpll(t): Fast decision procedures. *Computer Aided Verification : 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004. Proceedings*, strana 175, 2004.

[7] Mareš, M. a Valla, T. *Průvodce labyrintem algoritmů*. CZ.NIC, 2017.

[8] Moura, L. a Ruess, H. An experimental evaluation of ground decision procedures. *Computer Aided Verification : 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004. Proceedings*, strana 162, 2004.

[9] Nieuwenhuis, R. a Oliveras, A. *DPLL(T) with Exhaustive Theory Propagation and Its Application to Difference Logic.*, strany 321–334. 2005.

[10] Oliveras, A. a Rodríguez-Carbonell, E. Satisfiability modulo difference logic.

[11] Randal E., B., Shuvendu K., L. a Sanjit A., S. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. *Computer Aided Verification : 14th International Conference, CAV 2002 Copenhagen, Denmark, July 27–31, 2002 Proceedings*, strana 78, 2002.

[12] Schrijver, A. *Theory of linear and integer programming*. John Wiley & Sons, 1986.

A. Přílohy

A.1 První příloha