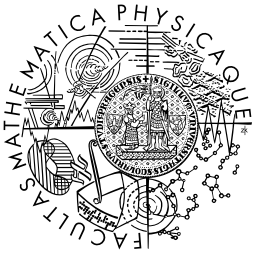


MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

DIPLOMOVÁ PRÁCE

Tomáš Hurt

Framework pro vývoj optimalizačních algoritmů

Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce: RNDr. Ing. Otakar Trunda
Studijní program: Informatika
Studijní obor: Softwarové systémy

Praha 2020

Prohlašuji, že jsem tuto diplomovou práci vypracoval(a) 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 má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

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

Rád bych na tomto místě poděkoval RNDr. Ing. Otakaru Trundovi za trpělivost a podnětné vedení mé práce. Také děkuji své rodině a přátelům za podporu a motivaci, bez níž by se tato práce nedočkala zdárného konce.

Název práce: Framework pro vývoj optimalizačních algoritmů

Autor: Tomáš Hurt

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce: RNDr. Ing. Otakar Trunda, Katedra teoretické informatiky a matematické logiky

Abstrakt: Cílem práce je navrhnout a implementovat efektivní nástroj pro vývoj a testování algoritmů v oblasti kombinatorické optimalizace. Bude vysvětlena problematika plánování a následně budou popsány kroky postupného návrhu a implementace vznikajícího programu. Framework bude podporovat dva hlavní vstupní formalismy pro popis optimalizačních problémů (PDDL, SAS$^+$). Zajištěna bude podpora zpracování vstupů, navrhnuty budou vhodné datové struktury a efektivní implementace prohledávacích algoritmů. Důraz bude kladen na dobrý objektový návrh programu z hlediska budoucího vývoje a snadné rozšiřitelnosti. K docílení toho budou využity ověřené principy z oblasti softwarového inženýrství.

Klíčová slova: plánování optimalizace algoritmy prohledávání PDDL SAS+

Title: Framework for development of optimization algorithms

Author: Tomáš Hurt

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: RNDr. Ing. Otakar Trunda, Department of Theoretical Computer Science and Mathematical Logic

Abstract: The aim of the thesis is to design and implement an efficient tool for research and testing of algorithms of the combinatorial optimization. The domain of the planning research will be explained and the steps of design and implementation of such program will be covered. The framework will support two primary formalisms for the description of optimization problems (PDDL, SAS$^+$). The inputs processing will be provided, suitable data structures and efficient implementations of search algorithms will also be included. The emphasis will be on a proper object design and easy extensibility for the future development. To achieve this goal, proven principles of software engineering will be used.

Keywords: planning optimization algorithms search PDDL SAS+

Obsah

Úvod

S tím, jak postupem času technologie stále více prorůstají do našich životů, roste i potřeba pro automatické zpracovávání a vyhodnocování informací. Obor umělé inteligence se zabývá rozličnými oblastmi, jejichž výsledky již dnes výrazně ovlivňují náš každodenní život. Jednou z kategorií umělé inteligence je i oblast kombinatorické optimalizace, konkrétněji plánování.

Plánování jako takové znamená promýšlení budoucích akcí. Jde o vědomý proces výběru a organizace kroků vedoucích k dosažení vytyčeného cíle. Umělá inteligence se snaží tento proces zautomatizovat. Potřeba vytvářet nějaký *plán* vznikne ve chvíli, kdy dostaneme komplexní problém, který nejsme schopni efektivně a rychle řešit jednoduchými prostředky, při kooperaci s jinými entitami, anebo vzniká riziko významných ztrát při volbě chybných kroků. Obvykle jde o náročný proces z hlediska času i financí, takže se provádí pouze tehdy, když benefity takového procesu převáží.

Motivací pro vývoj automatického plánování je tvorba nástrojů pro profesionály z rozličných oblastí (např. organizace záchranných akcí, řízení dopravy, logistika ve skladech a továrnách apod.), ale i posun umělé inteligence jakožto oboru bádání. Ta se ze své definice snaží podchytit a emulovat různé aspekty lidské inteligence – a racionální zvažování postupu pak evidentně musí být jedna z klíčových oblastí tohoto bádání.

Důsledkem je i vývoj autonomních inteligentních strojů schopných samostatného rozhodování. Příkladem budiž satelity a vesmírní roboti, kteří musejí být schopni postarat se o své přežití v momentech, kdy neexistuje možnost přímé komunikace s lidskou obsluhou. Pro využití v blízké budoucnosti zmiňme např. automobily bez řidiče.

Účelem této práce je vytvořit efektivní nástroj pro výzkumníky a badatele zabývající se oblastí plánování, který jim umožní snadný návrh a testování nových teorií a algoritmů. Vzniknout by postupně měl bohatý framework, dostupný pod jednou z volných licencí. Důraz je kladen na efektivnost a přehlednost implementace a snadnou rozšiřitelnost.

V první části této práce se obecně seznámíme s problematikou plánování, popíšeme základní reprezentace problémů a jejich specifika. Podíváme se na to, jak přistupovat k hledání řešení optimalizačních problémů a popíšeme základní prohledávací metody.

Následovat bude část popisující proces vzniku požadovaného frameworku z pohledu softwarového inženýrství. Blíže představeny budou podporované formalismy PDDL a SAS$^+$ a způsob jejich zpracování frameworkem. Významnou část díla bude tvořit popis implementace plánovače realizujícího hledání řešení plánovacího problému. Podíváme se na architektonický návrh produktu i hlubší implementační detaily jednotlivých komponent, možnosti rozšiřitelnosti a zmíníme se i o validaci a testování frameworku. Implementován a představen bude též nástroj pro automatizované provádění rozsáhlejších experimentů.

Část I

Co je plánování

1. Od reálného světa k modelu

V této a několika následujících kapitolách se blíže seznámíme s problematikou plánování, zadefinujeme si základní modely a reprezentace problémů. Později se zaměříme na metody prohledávání a hledání řešení, podíváme se i na odvozené modely a pokročilejší techniky plánování. Definice a pojmy čerpány zejména z knihy *Automated Planning: Theory and Practice (M. Ghallab, D. Nau, P. Traverso, 2004)*[1].

1.1 Doménově (ne)závislý přístup

Plánování se dotýká široké škály oblastí, od navigačních systémů, plánování pohybu robotů, logistických a manipulačních systémů, po sběr a vyhodnocování dat v počítačových sítích, při plánování urbanistické infrastruktury, ve finančním plánování a mnoha dalších.

V každé z těchto oblastí se v plánování uplatňuje tzv. *doménově závislý* přístup. To znamená, že každá oblast má své vlastní specifické reprezentace problému a techniky, kterými dosahují kýžených výsledků. Tyto techniky jsou postavené na specifikách příslušné domény. Například v ekonomickém sektoru se hojně využívají matematické programování a optimalizační techniky, při plánování pohybu robota se bude stavět významně na geometrii, kinematice apod.

Zjevným problémem je skutečnost, že je nepoměrně nákladnější vývoj plánovacích technik těsně spjatých s jednou konkrétní oblastí, přestože spousta aspektů je společná pro všechny domény. Hodil by se tedy nástroj na vyšší úrovni abstrakce, nezávislý na konkrétní doméně, nad kterým by následně jednotlivé oblasti mohly stavět a adaptovat ho pro své potřeby.

Proto je z hlediska výzkumu nejzajímavější tzv. *doménově nezávislý* přístup a vývoj obecných koncepcí a nástrojů v této problematice. Pro obecný popis problému se v doménově nezávislém plánování zavádí tzv. *konceptuální model*.

1.2 Konceptuální model

Konceptuální model slouží jako obecný nástroj pro popis plánovacího problému. Takový problém je dynamickým systémem, který v každém momentě umíme popsat nějakým *stavem*. K vývoji tohoto systému v čase (tj. systém přejde do nového stavu) přispívají buď přímé *akce* z naší strany nebo *události*, nad kterými nemáme kontrolu. Popsaný koncept vyjádříme formálně.

Definice 1 (přechodová funkce, přechodový systém)**.** *Nechť S, A, E jsou konečné, nebo rekurzivně spočetné množiny stavů, resp. akcí, resp. událostí. Pak $\gamma : S \times A \times E \to 2^S$ nazveme* přechodovou funkcí na stavech *a čtveřici $\mathbf{\Sigma} = (S, A, E, \gamma)$ nazveme* přechodovým systémem na stavech*.*

Přechodový systém si lze představit jako orientovaný graf[2], kde vrcholy tvoří množina stavů S a hrany představují přechody mezi stavy $\gamma(s, a, e)$, kde $s \in S$, $a \in A$, $e \in E$. Lze zavést též *neutrální akci* ι a *neutrální událost* ϵ pro případy,

kdy pro přechod z nějakého stavu stačí pouze akce nebo pouze událost – pak místo $\gamma(s, \iota, e)$ a $\gamma(s, a, \epsilon)$ můžeme psát $\gamma(s, e)$ a $\gamma(s, a)$.

Pokud $\gamma(s, a)$ je neprázdná množina, pak říkáme, že akce $a \in A$ je *aplikovatelná* na stav $s \in S$ a jejím aplikováním získáme potenciální nové stavy systému. Pokud $\gamma(s, e)$ je neprázdná množina, pak to znamená, že událost $e \in E$ *může nastat*, když je systém ve stavu $s \in S$, a výsledkem jsou opět nové stavy systému. Rozdíl mezi akcemi a událostmi je tedy pouze v tom, že nad akcemi máme přímou kontrolu, zatímco události vyjadřují vnitřní dynamiku systému.

Hlavním cílem plánování je najít sekvenci aplikovatelných akcí z nějakého počátečního stavu do nějakého stavu, který bude splňovat cílové podmínky. Cílovými podmínkami mohou být i požadavky na průchod určitými stavy nebo naopak požadavek na úplné vyhnutí se některým stavům apod.

Než se dostaneme ke konkrétním reprezentacím jednoduchých plánovacích problémů, potřebujeme pro tuto chvíli stávající model v několika ohledech omezit. Zavádíme proto následující sadu omezení pro přechodový systém Σ:

(a) Systém Σ má konečnou množinu stavů S.

(b) Systém Σ je plně pozorovatelný, tj. máme kompletní informaci o stavu systému.

(c) Systém Σ je deterministický, tj. $|\gamma(s, a)| \leq 1$ a $|\gamma(s, e)| \leq 1$ pro $s \in S$, $a \in A$, $e \in E$.

(d) Systém Σ je statický, bez vnitřní dynamiky. Tzn. pro množinu událostí $E = \emptyset$.

(e) Plánovač předpokládá pouze explicitně stanovené cíle, tj. nějakou množinu cílových stavů. Nelze stanovovat složitější cíle jako požadavek na průchod některými stavy apod.

(f) Výsledný plán je lineárně řazená sekvence akcí.

(g) Pro akce a události se neuvažují doby jejich trvání, jde o okamžité přechody v systému.

(h) Plánovač předpokládá stabilní systém, tj. nepředpokládá modifikaci systému za běhu plánovače (tzv. *offline plánování*).

Definice 2 (omezený přechodový systém)**.** *Nechť Σ je přechodový systém, pro který platí omezení (a) až (h) uvedená výše. Pak $\boldsymbol{\Sigma} = (S, A, \gamma)$ nazveme* omezeným přechodovým systémem na stavech.

Z definice Σ vypadla množina E, protože dle definice omezený přechodový systém nemůže mít žádné události. Rovněž budeme dále zjednodušovat notaci $\gamma(s, a) = s'$ namísto $\gamma(s, a) = \{s'\}$, díky deterministické povaze Σ.

Každé z výše uvedených omezení (a) až (h) lze relaxovat a vytvářet tzv. *rozšířené modely*, využívající pokročilejší techniky plánování. Základním referenčním kamenem však zůstává omezený přechodový systém, nad kterým je rozvinuta základní teorie plánování – tzv. *klasické plánování* (viz kapitola 2).

Nyní si ještě zadefinujeme přímo plánovací problém a řešení plánovacího problému nad omezeným přechodovým systémem.

Definice 3 (plánovací problém)**.** *Nechť* $\boldsymbol{\Sigma} = (S, A, \gamma)$ *je omezený přechodový systém nad stavy,* $s_0 \in S$ *je* počáteční stav *a* $S_g \subseteq S$ *je množina* cílových stavů*. Pak trojici* $\boldsymbol{P} = (\Sigma, s_0, S_g)$ *nazveme* plánovacím problémem *pro omezený přechodový systém nad stavy.*

Definice 4 (plán, délka plánu)**.** Plánem *nazveme libovolnou posloupnost akcí* $\pi = \langle a_1, \ldots, a_k \rangle$*, pro* $k \geq 1$*. Uvažujeme-li, že akce* a *má* cenu $cost(a)$*, pak* délka plánu *(někdy též* cena plánu*) označuje součet cen akcí v daném plánu, tj.* $\sum_{i=1}^{k} cost(a_i)$*. Pokud ceny akcí neuvažujeme, jsou implicitně rovné 1.*

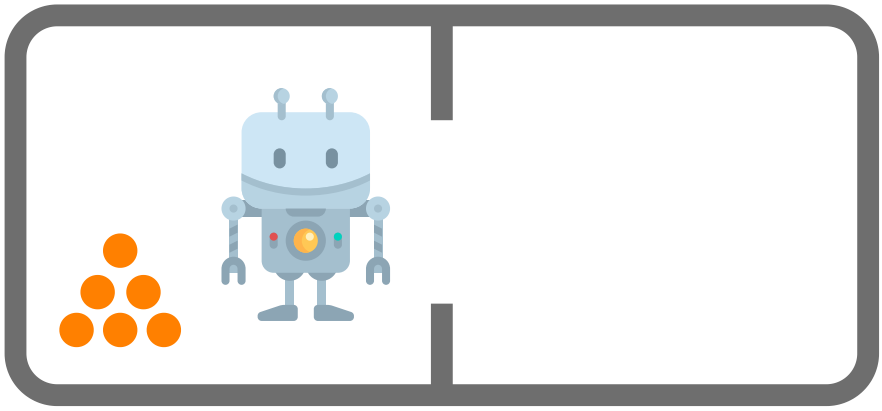
Definice 5 (konkatenace plánů)**.** *Nechť* $\pi_1 = \langle a_1, \ldots, a_k \rangle$ *a* $\pi_2 = \langle a'_1, \ldots, a'_j \rangle$ *jsou plány, pak jejich* konkatenací *je plán* $\pi_1 \cdot \pi_2 = \langle a_1, \ldots, a_k, a'_1, \ldots, a'_j \rangle$*. V případě, že plán je délky 1, např.* $\pi_2 = \langle a \rangle$*, pak lze zapsat* $\pi_1 \cdot \pi_2$ *také jako* $\pi_1 \cdot a$*.*

Definice 6 (řešení plánovacího problému)**.** *Nechť* $\boldsymbol{P} = (\Sigma, s_0, S_g)$ *je plánovací problém pro omezený přechodovým systém nad stavy a pro posloupnost akcí* $\langle a_1, a_2, \ldots, a_k \rangle$ *platí, že* $\gamma(\gamma(\ldots\gamma(\gamma(s_0, a_1), a_2), \ldots, a_{k-1}), a_k) \in S_g$*, pak tuto posloupnost akcí (plán) nazveme* řešením plánovacího problému.

Postupnou aplikaci sekvence akcí nějakého plánu $\pi = \langle a_1, a_2, \ldots, a_k \rangle$ na počáteční stav $s \in S$ můžeme zkráceně zapisovat jako $\gamma(s, \pi)$.

1.3 Motivační příklad – Gripper

Pro konkrétnější ilustraci jednotlivých konceptů využijeme klasický ilustrační plánovací příklad, zvaný *Gripper*[3]. V něm ovládáme robota s dvěma rukama, kde každá může nést nejvýše jeden míček. Několik míčků je na zemi v jedné místnosti a cílem je všechny míčky přenést do druhé místnosti. Robot umí pouze sebrat míček do levé či pravé ruky (*gripperu*), pustit míček z levé či pravé ruky a přesouvat se z jedné místnosti do druhé. Viz ilustrační obr. 1.1, jehož varianty nás budou provázet celým textem[1].



Obrázek 1.1: Ilustrační příklad domény Gripper

Problém se dá dále modifikovat, přidávat míčky, přidávat roboty, přidávat více místností, ale princip bude podobný. Na tento jednoduchý příklad budeme odkazovat v rámci následujících kapitol jako na Gripper, příp. *Gripper doménu.*

[1] Autor obrázku robota: Freepik (flaticon.com)

Pro popis stavu v Gripper doméně potřebujeme říci, kde se zrovna nacházejí jednotlivé míčky, kde se nachází robot, a co (pokud něco) drží v levé či pravé ruce. Pro přechody mezi stavy nám bude stačit pouze několik akcí – přesun robota z jedné místnosti do druhé, zvednutí míčku do levé či pravé ruky a puštění míčku z levé či pravé ruky. Při zvednutí míčku do ruky však musí být splněno, že je robot ve stejné místnosti jako míček, má danou ruku volnou apod.

Cílem plánovače je najít sekvenci akcí (plán), která dostane robota do cílového stavu (tj. takový, v kterém všechny míčky jsou v druhé místnosti) a bude v ideálním případě nejkratší možná.

2. Klasické plánování

Klasické plánování je teorie plánování nad omezeným přechodovým systémem, též nazývaná jako *STRIPS-plánování*. Tento model, jakkoli nerealistický, dal základ pro vývoj pokročilejších technik uplatnitelných nad reálnými problémy.

Klasické plánování se zaměřuje na dva problematické aspekty plánování. Za prvé, jak reprezentovat množiny S, A a γ, aniž bychom byli nuceni je explicitně vypisovat, a za druhé, jak najít řešení plánovacího problému efektivně – jaké prohledávací prostory, jaké techniky, algoritmy a heuristiky použít.

Studium klasického plánování vyvinulo dva základní způsoby reprezentace plánovacího problému – tzv. *klasickou reprezentaci* a *reprezentaci se stavovými proměnnými*.

2.1 Klasická reprezentace

Klasická reprezentace využívá k vyjádření vztahů v systému syntax odvozený z predikátové logiky prvního řádu[4]. Stavy pak reprezentuje jako množinu logických atomů, které mohou být pravdivé i nepravdivé v různých interpretacích.

Příklad. Predikáty v Gripper doméně:

- room(x) – pravdivé, právě když x je místnost,
- ball(x) – pravdivé, právě když x je míček,
- gripper(x) – pravdivé, právě když x je gripper (tj. ruka robota),
- at-robby(x) – pravdivé, právě když x je místnost a robot je v x,
- at-ball(x, y) – pravdivé, právě když x je míček, y je místnost a x je v y,
- free(x) – pravdivé, právě když x je gripper a x nedrží žádný míček,
- carry(x, y) – pravdivé, právě když x je gripper, y je míček a x drží y.

Některé vztahy se časem mohou měnit – např. poloha míčku v první místnosti at-ball(ball1, roomA) přestane platit, pokud robot přesune míček do druhé místnosti. Takovýmto stavům říkáme *fluentní* nebo *flexibilní*. Jiné vztahy zůstávají po celou dobu neměnné a vyjadřují uspořádání světa – např. ball(ball1) vyjadřující skutečnost, že konstanta ball1 je míček, se evidentně měnit nebude. Takovýmto vztahům v systému říkáme *rigidní*.

Definice 7 (stav v klasické reprezentaci)**.** *Vezměme jazyk L, který má konečně predikátových a konstantních symbolů a nemá žádné funkční symboly.* Stav v klasické reprezentaci *je množina groundovaných atomů (tj. uzavřených atomických formulí).*

Využívá se přitom předpoklad uzavřeného světa, tj. pro skutečnost, která není ve stavu explicitně uvedena, předpokládáme, že v tomto stavu neplatí. Pokud naopak řekneme, že nějaká relace (tj. predikát) p platí ve stavu s, znamená to, že $p \in s$.

Příklad. Použité konstanty a počáteční stav v doméně Gripper:

- Množina konstant v jazyce L: {roomA, roomB, ball1, ball2, ball3, ball4, left, right}
- Počáteční stav: s_0 = {room(roomA), room(roomB), ball(ball1), ball(ball2), ball(ball3), ball(ball4), gripper(left), gripper(right), free(left), free(right), at-robby(roomA), at-ball(ball1, roomA), at-ball(ball2, roomA), at-ball(ball3, roomA), at-ball(ball4, roomA)}

Přechody mezi stavy zajišťují akce, které jsou v klasické reprezentaci instancí nějakého *plánovacího operátoru*.

Definice 8 (plánovací operátor)**.** *V klasickém plánování je* plánovací operátor *trojice op = (name(op), precond(op), effects(op)), kde:*

- *name(op) je vyjádření jména operátoru, jde o designátor tvaru* $n(x_1, \ldots, x_k)$ *pro k-ární operátor pojmenovaný symbolem n, kde n je unikátní v rámci jazyka L,*
- *precond(op) a effects(op) jsou předpoklady a efekty operátoru op definované jako množiny literálů (atom nebo negace atomu).*

Ze své podstaty se v efektech operátoru nemohou vyskytovat rigidní relace, které jsou neměnné pro všechny stavy daného problému. Mohou však být součástí předpokladů. Každý predikát v efektech operátoru je tedy fluentní relace.

Příklad. Dostupné operátory v doméně Gripper:

- move(x, y) – robot se posune z místnosti x do místnosti y
  - precond(move) = {room(x), room(y), at-robby(x)}
  - effects(move) = {at-robby(y), ¬at-robby(x)}
- pick(x, y, z) – robot vezme míček x v místnosti y rukou z
  - precond(pick) = {ball(x), room(y), gripper(z), at-ball(x, y), at-robby(y), free(z)}
  - effects(pick) = {carry(z, x), ¬at-ball(x, y), ¬free(z)}
- drop(x, y, z) – robot pustí míček x v místnosti y z ruky z
  - precond(drop) = {ball(x), room(y), gripper(z), carry(z, x), at-robby(y)}
  - effects(drop) = {at-ball(x, y), free(z), ¬carry(z, x)}

Definice 9 (pozitivní/negativní předpoklady a efekty)**.** *Nechť L je množina literálů. Pak množinu všech atomů množiny L označme* L^+ *a množinu všech negací atomů množiny L označme* L^-*.*

Analogicky pro operátor (či instanci operátoru) op nazveme množiny značené $precond^+(op)$ *a* $precond^-(op)$ *pozitivními a negativními předpoklady operátoru a množiny* $effects^+(op)$ *a* $effects^-(op)$ *pozitivními a negativními efekty operátoru.*

Definice 10 (akce v klasické reprezentaci)**.** *V klasickém plánování je* akce *instance plánovacího operátoru, tj. nějaká substituce proměnných plánovacího operátoru. Nechť s je stav a a je akce, přičemž platí, že jsou splněny všechny pozitivní předpoklady, tj. $\text{precond}^+(a) \subseteq s$, a neplatí žádné negativní předpoklady, tj. $\text{precond}^-(a) \cap s = \emptyset$, pak řekneme, že je akce a* aplikovatelná *na stav s. Výsledkem aplikace je nový stav s', který získal pozitivní efekty a pozbyl negativní efekty akce a:*

$$s' = \gamma(s, a) = (s - \text{effects}^-(a)) \cup \text{effects}^+(a).$$

Příklad. Příkladem nějaké konkrétní instance operátoru, tj. akce, v Gripper doméně budiž např. pick(ball1, roomA, left) – míček číslo 1 je v místnosti A zvednut robotem do levé ruky.

Definice 11 (plánovací doména v klasické reprezentaci)**.** *Nechť L je jazyk v predikátové logice prvního řádu s konečně mnoha relačními symboly a konstantními symboly.* Plánovací doména *v jazyce L je omezený přechodový systém $\Sigma = (S, A, \gamma)$, kde:*

- $S \subseteq 2^{\{\text{groundované atomy jazyka } L\}}$,
- *$A = \{$všechny groundované instance operátorů v $Op\}$, kde Op je množina operátorů definovaných výše,*
- *$\gamma(s, a) = (s - \text{effects}^-(a)) \cup \text{effects}^+(a)$, pakliže akce $a \in A$ je aplikovatelná na stav $s \in S$ (jinak hodnota nedefinována),*
- *S je uzavřená pro přechodovou funkci γ, tj. pro všechny aplikovatelné akce $a \in A$ na stav $s \in S$ je $\gamma(s, a) \in S$.*

Definice 12 (plánovací problém v klasické reprezentaci)**.** *V klasické reprezentaci je* plánovací problém *trojice $\boldsymbol{P} = (\Sigma, s_0, g)$, kde:*

- *$\Sigma = (S, A, \gamma)$ je plánovací doména,*
- *s_0, tj. počáteční stav, je libovolný stav z množiny stavů S,*
- *g, tj. cílová podmínka, je libovolná množina groundovaných literálů,*
- *$S_g = \{s \in S \mid s$ splňuje $g\}$ je množina cílových stavů.*

Skutečnost, že stav s *splňuje* g ($s \models g$) platí právě tehdy, když existuje substituce σ taková, že každý pozitivní literál ze $\sigma(g)$ je v s a žádný negativní literál ze $\sigma(g)$ v s není.

Definice 13 (specifikace plánovacího problému)**.** *Specifikace plánovacího problému $\boldsymbol{P} = (\Sigma, s_0, g)$ je trojice $\mathcal{P} = (Op, s_0, g)$.*

Specifikace (angl. *statement*) plánovacího problému se používá k vyjádření plánovacího problému bez nutnosti explicitní enumerace všech stavů přechodového systému a všech jednotlivých přechodů. Veškerá potřebná informace, tj. do kterých stavů se lze během výpočtu dostat, je již obsažena v definici operátorů a potřebujeme pouze počáteční stav a cílové podmínky.

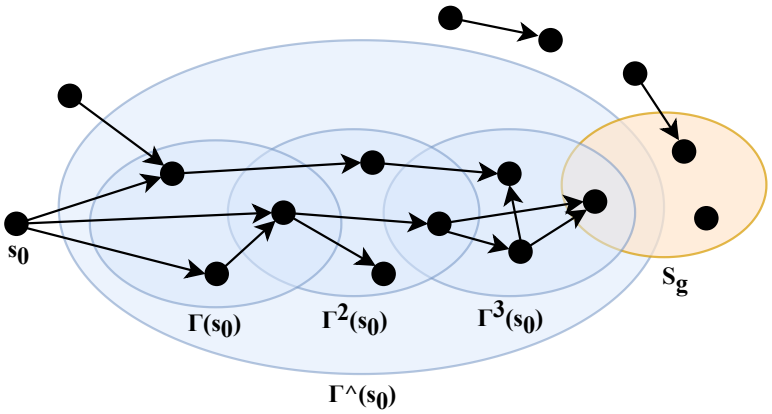
Může existovat více různých specifikací jednoho plánovacího problému, ale platí, že každá z nich bude mít stejnou množinu dosažitelných stavů i množinu řešení.

Definice 14 (množina následníků)**.** *Pro stav $s \in S$ je množina všech jeho* následníků*:*

$$\Gamma(s) = \{\gamma(s, a) \mid a \in A \text{ a zároveň } a \text{ je aplikovatelná na stav } s\}.$$

Definice 15 (množina dosažitelných stavů)**.** *Je-li $\Gamma(s)$ množina následníků ze stavu $s \in S$, pak buď $\Gamma^2(s) = \Gamma(\Gamma(s)) = \bigcup(\Gamma(s') \mid s' \in \Gamma(s))$ a analogicky pro $\Gamma^3(s)$, $\Gamma^4(s)$ atd.* Množinou dosažitelných stavů *ze stavu s pak označme tranzitivní uzávěr:*

$$\hat{\Gamma}(s) = \Gamma(s) \cup \Gamma^2(s) \cup \dots.$$



Obrázek 2.1: Příklad množiny dosažitelných stavů ze stavu s_0

Definice 16 (relevantní akce)**.** *Akce a je* relevantní *pro množinu cílových podmínek g (tj. může přenést systém do stavu, který splňuje cílové podmínky g), pokud platí:*

- *$g \cap effects(a) \neq \emptyset$, tj. efekty akce a přispívají k podmínkám g, a také*
- *$g^+ \cap effects^-(a) = \emptyset$ a $g^- \cap effects^+(a) = \emptyset$, tj. efekty akce a nejsou v konfliktu s podmínkami g.*

Definice 17 (regresní množina k cílovým podmínkám)**.** *Nechť a je relevantní akce pro množinu cílových podmínek g. Pak* regresní množina *k cílovým podmínkám g je*

$$\gamma^{-1}(g, a) = (g - effects^+(a)) \cup precond(a).$$

Definice 18 (množina všech regresních množin)**.** Množinu regresních množin *pro cílové podmínky g definujeme jako*

$$\Gamma^{-1}(g) = \{\gamma^{-1}(g, a) \mid a \in A \text{ je relevantní pro } g\}.$$

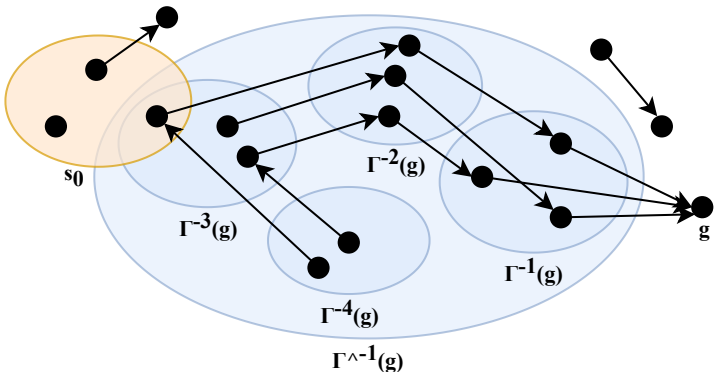
Definice 19 (dosažitelnost cílových podmínek)**.** *Předpokládáme-li, že $\Gamma^{-2}(g) = \Gamma^{-1}(\Gamma^{-1}(g)) = \bigcup(\Gamma^{-1}(g') \mid g' \in \Gamma^{-1}(g))$ označuje množinu všech regresních množin ve dvou krocích, analogicky $\Gamma^{-3}(g)$ ve třech krocích atd., pak tranzitivní uzávěr označíme:*

$$\hat{\Gamma}^{-1}(g) = \Gamma^{-1}(g) \cup \Gamma^{-2}(s) \cup \dots .$$

Zjevně platí, že nějaký cílový stav je dosažitelný v jediném kroku z nějakého stavu $s \in S$ právě tehdy, když existuje prvek $s' \in \Gamma^{-1}(g)$ takový, že $s' \subseteq s$. Analogicky, libovolný stav je zpětně dosažitelný z nějakého cílového stavu, pokud je nadmnožinou prvku z $\hat{\Gamma}^{-1}(g)$.

Z výše uvedeného vyplývají dvě pozorování. Plánovací problém $\boldsymbol{P} = (\Sigma, s_0, g)$ má řešení právě tehdy, když $S_g \cap \hat{\Gamma}(s_0) \neq \emptyset$. Viz příklad problému na obr. 2.1, kde jeden z cílových stavů je dosažitelný ze s_0 ve třech krocích.

Stejně tak platí, že $\boldsymbol{P} = (\Sigma, s_0, g)$ má řešení právě tehdy, když s_0 je nadmnožinou nějakého prvku z $\hat{\Gamma}^{-1}(g)$. Viz příklad na obr. 2.2 – tečky reprezentují nějaké množiny podmínek a přechody jsou aplikací relevantních akcí na tyto podmínky. Dá se říci, že každá z těchto teček reprezentuje všechny stavy, kterých jsou tyto podmínky součástí.



Obrázek 2.2: Příklad dosažitelnosti cílových podmínek g

Příklad. Popsané koncepty (stavy, cíle, akce, plány) nyní ještě jednou ilustrujme v doméně Gripper se 4 míčky:

- Systém je ve stavu s_k (viz obr. 2.3):

  s_k = {at-robby(roomB), carry(left, ball2), free(right), at-ball(ball1, roomA), at-ball(ball3, roomB), at-ball(ball4, roomB)}.

- Pro zjednodušení jsme oddělili rigidní relace (jsou platné v každém stavu):

  R = {room(roomA), room(roomB), ball(ball1), ball(ball2), ball(ball3), ball(ball4), gripper(left), gripper(right)}.

- Splněny jsou 2 ze 4 cílových podmínek g, kde:

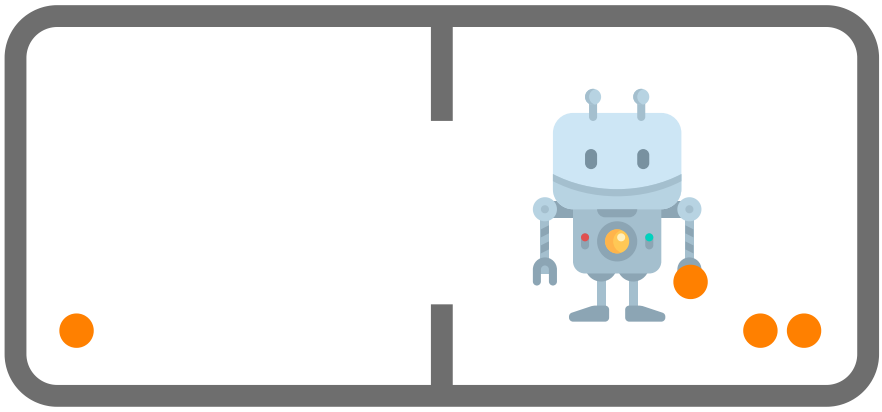
  g = {at-ball(ball1, roomB), at-ball(ball2, roomB), at-ball(ball3, roomB), at-ball(ball4, roomB)}.

- Aplikovatelné akce ze stavu s_k:

  - drop(ball2, roomB, left),
  - pick(ball3, roomB, right),
  - pick(ball4, roomB, right),
  - move(roomB, roomA).

- Pouze první ze jmenovaných akcí je relevantní, protože přímo přispívá k cílovým podmínkám, ostatní nikoli.
- Např. aplikace akce drop(ball2, roomB, left) vytvoří stav:

$$s_{k+1} = \{\text{at-robby(roomB), free(left), free(right), at-ball(ball1, roomA),}\\ \text{at-ball(ball2, roomB), at-ball(ball3, roomB), at-ball(ball4, roomB)}\}.$$



Obrázek 2.3: Doména Gripper ve stavu s_k

- Pokud $s_k = s_0$, tj. jde o počáteční stav plánovacího problému, pak řešením tohoto problému při daných cílových podmínkách g jsou např. tyto plány:
  - $\pi_1 = \langle$drop(ball2, roomB, left), move(roomB, roomA), pick(ball1, roomA, left), move(roomA, roomB), drop(ball1, roomB, left)$\rangle$,
  - $\pi_2 = \langle$move(roomB, roomA), pick(ball1, roomA, right), move(roomA, roomB), drop(ball1, roomB, left), drop(ball2, roomB, right)$\rangle$,
  - $\pi_3 = \langle$move(roomB, roomA), move(roomA, roomB), move(roomB, roomA), pick(ball1, roomA, right), move(roomA, roomB), drop(ball1, roomB, left), drop(ball2, roomB, right)$\rangle$.
- První dva uvedené plány mají stejnou (v tomto případě i minimální) délku, tj. 5 akcí. Třetí plán provádí navíc nějaké nepotřebné akce, a proto má délku 7.

2.2 Rozšíření klasické reprezentace

V klasické reprezentaci se postupně vyvinula další rozšíření dovolující snazší modelaci vstupních plánovacích problémů, ale i efektivnější funkci plánovačů. Ačkoli zůstáváme ve stejném modelu omezeného přechodového systému, umožníme navíc tyto aspekty:

- Použití *obecných a existenčních kvantifikátorů* v předpokladech operátorů a cílových podmínkách. Místo množin literálů tak budeme potřebovat pracovat s obecnějšími logickými výrazy. Např. požadavek v doméně Gripper, aby všechny míčky byly v místnosti B, lze místo výpisem všech relací zapsat formulí jako forall(x) (at-ball(x, roomB)).
- Zavedení *typování* proměnných v parametrech operátorů a relací. Např. v relaci at-ball(x, y) můžeme specifikovat, že první parametr x může nabývat pouze konstant typu *míček* a parametr y pouze konstant typu *místnost*.
- *Podmíněné efekty* operátorů – ve chvíli, kdy je operátor aplikovatelný na určitý stav, provedou se specifikované efekty operátoru, navíc se ale vyhodnotí další dodatečná podmínka, která při kladném vyhodnocení spustí aplikaci dodatečného efektu. Takto podmíněných pod-efektů může být definováno více.
- *Obecný kvantifikátor v efektech* operátorů – není potřeba vypisovat jednotlivé literály, ale obecně kvantifikovat nějaký parametr, např. forall(x) (free(x)), namísto {free(left), free(right)}.
- Použití *funkcí* – např. můžeme mít definovanou funkci *nosnost ruky robota* maxweight(x) a *váhu míčku* weight(b) a vrácené hodnoty pak použít v rámci predikátu less(maxweight(x), weight(b)), který se může dále hodit např. v předpokladech operátoru pro zvednutí míčku.

Poznámka. Zavedeným formalismem pro popis plánovacího problému v klasické reprezentaci (včetně výše zmíněných rozšíření) je zejména jazyk *PDDL*. Blíže se s tímto prostředkem seznámíme v části popisující implementaci plánovacího frameworku PAD, který jazyk PDDL plně podporuje (kapitola 5.1).

2.3 Reprezentace se stavovými proměnnými

Reprezentace se stavovými proměnnými (angl. *state-variable*) je alternativním modelem pro reprezentaci plánovacích problémů v klasickém plánování. Ve vyjadřovací síle je ekvivalentní klasické reprezentaci, ale namísto flexibilních relací tato reprezentace vsází na funkční hodnoty.

Na příkladu Gripper domény si vezměme relaci at-robby(x), reprezentující skutečnost, že robot je v místnosti x. Přitom platí, že nemůže být zároveň v místnosti A i v místnosti B, ale musí vždy někde být, tj. existuje právě jedno x, že platí predikát at-robby(x). Na základě tohoto zkonstruujme funkci, která nám pro stav $s \in S$ řekne, kde se robot nachází: $f_{\text{at-robby}} : S \to \{\text{roomA, roomB}\}$.

Tyto funkce nazveme *funkce pro stavové proměnné* (příp. zkráceně *stavové funkce*) a jejich hodnoty nám budou udávat konkrétní vlastnosti pro daný stav.

Definice 20 (množina konstant, třídy konstant, objektová proměnná)**.** *Označme D množinu všech* konstant *v plánovací doméně. Tuto množinu lze rozdělit na disjunktní* třídy *konstant. Pokud budeme hovořit o* objektové proměnné *v, pak půjde o proměnnou nabývající hodnot z D^v (= sjednocení jedné či více tříd konstant).*

Definice 21 (stavová proměnná)**.** *k-ární* stavová proměnná *(příp.* proměnná stavu*) je výraz ve tvaru $x(v_1, \ldots, v_k)$, kde x je symbol pro stavovou proměnnou*

a každý parametr v_i buď objektová proměnná, nebo konstanta. Stavová proměnná je vyjádřením stavové funkce

$$x : D_1^x \times \cdots \times D_k^x \times S \to D_{k+1}^x,$$

kde $D_i^x \subseteq D$ je sjednocením jedné či více tříd.

Stavové proměnné vždy závisí na aktuálním stavu, tudíž předpokládáme, že je stavový parametr implicitně daný a např. místo at-ball(ball2, s) píšeme pouze at-ball(ball2).

Definice 22 (groundované a negroundované stavové proměnné)**.** *Stavová proměnná $x(v_1, \ldots, v_k)$ je* groundovaná*, pakliže každý parametr v_i je konstantou z D_i^x, a je* negroundovaná*, pokud alespoň jeden z parametrů v_i je objektovou proměnnou.*

Stavová proměnná má být charakteristickým atributem stavu. Stav tedy popisujeme jako výpis hodnot všech groundovaných stavových proměnných.

Příklad. Ilustrujme si reprezentaci se stavovými proměnnými na Gripper doméně se 4 míčky. K dříve definovaným konstantám přidáme speciální konstantu *nil*, reprezentující prázdnou hodnotu a celý systém popíšeme za pomoci pouhých tří funkcí, resp. odpovídajícími stavovými proměnnými.

- Množina všech konstant:

  $$D = \{\text{roomA, roomB, ball1, ball2, ball3, ball4, left, right, nil}\}.$$

- Třídy konstant (de-facto typy pro konstanty):
  - $D_{\text{rooms}} = \{\text{roomA, roomB}\}$,
  - $D_{\text{balls}} = \{\text{ball1, ball2, ball3, ball4}\}$,
  - $D_{\text{grippers}} = \{\text{left, right}\}$.
  - $D_{\text{nil}} = \{\text{nil}\}$.

- Použité stavové funkce:
  - $f_{\text{at-robby}} : S \to D_{\text{rooms}}$,
  - $f_{\text{at-ball}} : D_{\text{balls}} \times S \to D_{\text{rooms}} \cup D_{\text{nil}}$,
  - $f_{\text{carry}} : D_{\text{grippers}} \times S \to D_{\text{balls}} \cup D_{\text{nil}}$.

- Stav s_k na obr. 2.3 zapsaný pomocí groundovaných stavových proměnných:
  - $s_k = \{$at-robby = roomB, carry(left) = ball2, carry(right) = nil, at-ball(ball1) = roomA, at-ball(ball2) = nil, at-ball(ball3) = roomB, at-ball(ball4) = roomB$\}$.

I v reprezentaci se stavovými proměnnými potřebujeme popsat v daném systému vztahy, které se za běhu nemění, tj. jsou invariantní – opět je nazveme *rigidní relace*.

Definice 23 (rigidní relace)**.** *k-ární* rigidní relace *je výraz tvaru* $r(v_1, \ldots, v_k)$*, kde* v_i *je konstanta z množiny* D_i^r *(sjednocení jedné či více tříd z* D*), přitom* $r \subseteq D_1^r \times \cdots \times D_k^r$*.*

Pokud bychom měli v doméně Gripper více místností, potřebovali bychom reprezentovat, které dvě místnosti mají mezi sebou spojení nebo dveře – např. adjacent(roomA, roomC). Čili jde o relaci adjacent $\subseteq D_{\text{rooms}} \times D_{\text{rooms}}$.

Rigidní relace se z definice nemění, není je tedy třeba uvádět v každém stavu. Mohou být však v předpokladech operátorů.

Definice 24 (operátor v reprezentaci se stavovými proměnnými)**.** Plánovací operátor *v reprezentaci se stavovými proměnnými je trojice op = (name(op), precond(op), effects(op)), kde:*

- *name(op) je syntaktický výraz tvaru* $n(u_1, \ldots, u_k)$ *pro k-ární operátor, kde n je unikátní* operátorový symbol *(tj. dva operátory nemají stejný symbol) a parametry* u_i *jsou objektové proměnné,*
- *precond(op) je množina výrazů na stavových proměnných a rigidních relacích reprezentující předpoklady operátoru,*
- *effects(op) je množina přiřazení nových hodnot stavovým proměnným reprezentující efekty operátoru – každé takové přiřazení zapisujeme ve tvaru* $x(t_1, \ldots, t_k) \leftarrow t_{k+1}$*, kde každé* t_i *je term z příslušné domény.*

Příklad. Ilustrujme na Gripper doméně definici dostupných operátorů. V předpokladech operátorů využíváme rigidních relací room(x), ball(x) a gripper(x) pro kontrolu typů vstupních parametrů.

- move(x, y) – robot se posune z místnosti x do místnosti y
  - precond(move) = {room(x), room(y), at-robby = x}
  - effects(move) = {at-robby $\leftarrow y$}
- pick(x, y, z) – robot vezme míček x v místnosti y rukou z
  - precond(pick) = {ball(x), room(y), gripper(z), at-robby = y, at-ball(x) = y, carry(z) = nil}
  - effects(pick) = {at-ball(x) $\leftarrow$ nil, carry(z) $\leftarrow x$}
- drop(x, y, z) – robot pustí míček x v místnosti y z ruky z
  - precond(drop) = {ball(x), room(y), gripper(z), at-robby = y, carry(z) = x}
  - effects(drop) = {at-ball(x) $\leftarrow y$, carry(z) $\leftarrow$ nil}

Podmínky v předpokladech operátorů se odkazují k aktuálnímu stavu, přiřazení nových hodnot v efektech operátorů pak odpovídá novému stavu $\gamma(s, a)$.

Akce v reprezentaci se stavovými proměnnými odpovídají groundovaným variantám operátorů, pokud jsou zároveň splněny rigidní relace v předpokladech operátorů. Např. move(roomA, ball2) není vůbec považována za akci, neboť neplatí relace room(ball2), tj. ball2 není místnost.

Akce a je aplikovatelná na stav s, pakliže hodnoty stavových proměnných stavu s odpovídají požadavkům v předpokladech precond(a). Při aplikaci akce je vytvořen nový stav $\gamma(s, a)$ změnou hodnot stavových proměnných specifikovaných v efektech effects(a), zbylé stavové proměnné zůstávají netknuté.

Množina stavů S je definována jako množina dosažitelných stavů z počátečního stavu za pomoci plánovacích operátorů s ohledem na definované rigidní relace.

Cílové podmínky se specifikují jako množina hodnot pro jednu či více stavových proměnných. Např. pro Gripper doménu budeme mít podmínky g = {at-ball(ball1) = roomB, at-ball(ball2) = roomB, at-ball(ball3) = roomB, at-ball(ball4) = roomB}.

Definice 25 (plánovací doména v reprezentaci se stavovými proměnnými)**.** *Nechť L je jazyk v reprezentaci se stavovými proměnnými specifikovaný konečnou množinou stavových proměnných X, konečnou množinou rigidních relací R a množinou konstant.* Plánovací doména *v L je omezený přechodový systém $\Sigma = (S, A, \gamma)$, kde platí:*

- *$S \subseteq \prod_{x \in X} D_x$, kde D_x je obor hodnot groundované stavové proměnné x. Stav s zapisujeme ve tvaru $s = \{(x = c) \mid x \in X\}$, kde $c \in D_x$.*
- *A = {všechny groundované instance operátorů z Op splňující relace z R}, kde Op je množina operátorů definovaných výše; akce a je aplikovatelná na stav s, právě když každý výraz $(x = c)$ v precond(a) je obsažen v s.*
- *$\gamma(s, a) = \{(x = c) \mid x \in X\}$, kde c je specifikováno přiřazením $x \leftarrow c$ v effects(a), pokud takové existuje, jinak $(x = c) \in s$.*
- *Množina S je uzavřena na operaci γ, tzn. pokud $s \in S$, pak pro každou akci $a \in A$ aplikovatelnou na s platí $\gamma(s, a) \in S$.*

Definice 26 (plánovací problém v reprezentaci se stavovými proměnnými)**.** *V reprezentaci se stavovými proměnnými je* plánovací problém *trojice $\boldsymbol{P} = (\Sigma, s_0, g)$, kde $s_0 \in S$ je počáteční stav a g množina cílových podmínek na stavových proměnných X. Množina cílových stavů je S_g = {$s \in S \mid s$ splňuje g}.*

Cílové podmínky mohou obsahovat i negroundované výrazy, např. g = {at-ball(b) = roomB} požadující libovolný míček b v místnosti B. Stav s *splňuje* cílové podmínky g, existuje-li taková substituce σ, že každý výraz $\sigma(g)$ je obsažen v s.

Definice 27 (relevantní akce v reprezentaci se stavovými proměnnými)**.** *Akce a je relevantní pro cílové podmínky g, právě když jsou splněny obě následující podmínky:*

- *$g \cap$ effects(a) $\neq \emptyset$.*
- *Pro každý výraz $(x = c)$ v g platí, že effects(a) neobsahuje žádné přiřazení tvaru $x \leftarrow d$ takové, že c a d jsou různé konstantní symboly.*

Pakliže a je relevantní pro g, pak $\gamma^{-1}(g, a) = (g - \upsilon(a)) \cup$ precond(a), kde $\upsilon(a) = \{(x = c) \mid (x \leftarrow c) \in$ effects(a)$\}$.

Definice 28 (specifikace plánovacího problému v reprezentaci se stavovými proměnnými)**.** Specifikace *(angl. statement) plánovacího problému* $\boldsymbol{P} = (\Sigma, s_0, g)$ *v reprezentaci se stavovými proměnnými je čtveřice* $\mathcal{P} = (Op, R, s_0, g)$*, kde* Op *je množina operátorů,* R *je množina rigidních relací,* s_0 *je počáteční stav a* g *jsou cílové podmínky.*

Na rozdíl od klasické reprezentace je třeba ve specifikaci problému uvést navíc množinu rigidních relací R, neboť tyto relace se neuvádí explicitně ve specifikaci stavu, a nelze je tedy odvodit z s_0.

O reprezentaci se stavovými proměnnými řekneme, že je *groundovaná*, pokud všechny stavové proměnné jsou bez argumentů. Pak neexistují žádné objektové proměnné, ani rigidní relace a vše je odvislé pouze od dobré definice operátorů.

Stav v reprezentaci se stavovými proměnnými je tedy dán výčtem hodnot všech stavových proměnných. Abychom nemuseli vypisovat všechny hodnoty, užívá se tzv. *předpoklad defaultní hodnoty* (obdoba předpokladu uzavřeného světa v klasické reprezentaci, kde předpokládáme, že explicitně neuvedená skutečnost jednoduše neplatí).

Jak již bylo zmíněno, klasická reprezentace i reprezentace se stavovými proměnnými jsou ve vyjadřovací síle ekvivalentní. Každá však může být výhodnější pro jiný specifický typ plánovacího problému.

Pro převod klasické reprezentace do reprezentace se stavovými proměnnými stačí nahradit každý pozitivní literál tvaru $p(t_1, \ldots, t_n)$ výrazem $p(t_1, \ldots, t_n) = 1$ a každý negativní literál $\neg p(t_1, \ldots, t_n)$ výrazem $p(t_1, \ldots, t_n) = 0$.

Pro převod reprezentace se stavovými proměnnými na klasickou reprezentaci nejprve nahradíme každý výraz tvaru $x(t_1, \ldots, t_n) = v$ atomem ve tvaru $x(t_1, \ldots, t_n, v)$ v počátečním stavu a předpokladech operátorů. Dále každé přiřazení $x(t_1, \ldots, t_n) \leftarrow v$ v efektech operátoru nahradíme dvojicí $\neg x(t_1, \ldots, t_n, w)$ a $x(t_1, \ldots, t_n, v)$ a přidáme $x(t_1, \ldots, t_n, w)$ do předpokladu operátoru.

Poznámka. Používaným formalismem pro popis plánovacího problému v reprezentaci se stavovými proměnnými je *SAS*$^+$. Blíže se s tímto prostředkem seznámíme v části popisující implementaci plánovacího frameworku PAD, který formalismus SAS$^+$ plně podporuje (kapitola 5.1).

3. Hledání plánu

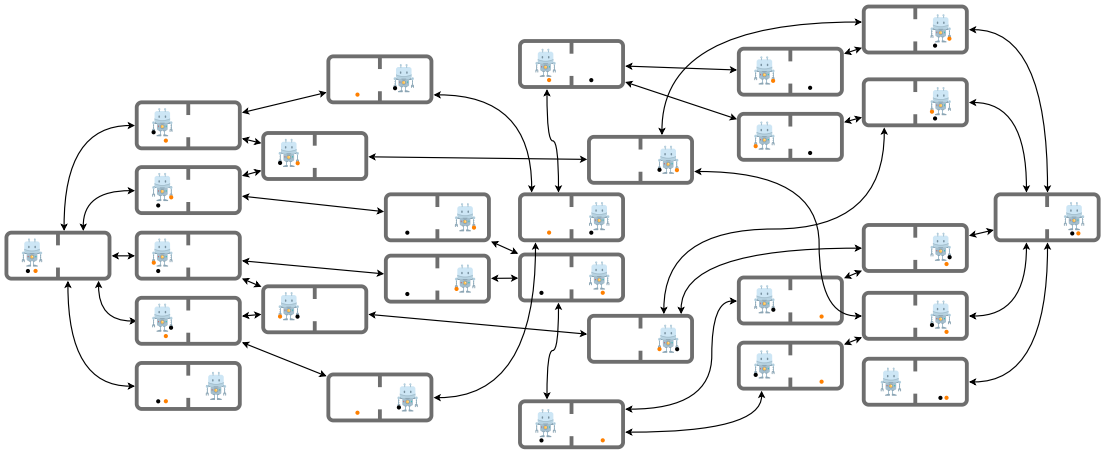
V této kapitole načrtneme základní techniky pro hledání řešení plánovacího problému. Společně s předchozími sekcemi rozebírajícími jednotlivé reprezentace tak dostaneme dobrý základ pro formulaci požadavků při návrhu implementace plánovacího frameworku.

Existují dva základní způsoby, jak k hledání řešení přistupovat – buďto skrze hledání ve stavovém prostoru plánovacího problému (tzv. *state-space planning*) nebo hledání v prostoru plánů (tzv. *plan-space planning*).

3.1 Plánování v prostoru stavů

Nejběžnějším přístupem při hledání řešení plánovacího problému je prohledávání prostoru stavů. Ten se dá formulovat jako orientovaný graf, kde množinu vrcholů tvoří jednotlivé stavy (reprezentující stav *světa*) a množinu hran jednotlivé přechody mezi stavy, tj. aplikace akcí. Nalezení plánu znamená v tomto grafu nalézt cestu na hranách z počátečního do cílového stavu.

Jak složitý může být prostor stavů i pro triviální doménu ilustruje obr. 3.1 na doméně Gripper s dvěma míčky. Jednotlivé přechody mezi stavy jsou vždy obousměrné (moveA/moveB nebo pick/drop). Cílem je najít sekvenci přechodů z počátečního do nějakého stavu splňujícího cílové podmínky.



Obrázek 3.1: Kompletní prostor stavů domény Gripper s dvěma míčky

Forward-Search

Přímočarou variantou hledání řešení v prostoru stavů je *Forward-Search* (*tj. dopředné hledání*). To bere na vstupu specifikaci plánovacího problému $\mathcal{P} = (Op, s_0, g)$ a v případě, že má $\mathcal{P}$ řešení, pak nějaké řešení vrátí. Pokud řešení neexistuje, oznámí *neúspěch*.

Forward-Search(Op, s_0, g):

$s \leftarrow s_0$
$\pi \leftarrow$ prázdný plán
loop
  if s splňuje g **then**
    return π
  end if
  $A \leftarrow \{a \mid a$ je ground. instance z Op a precond(a) platí v $s\}$
  if $A = \emptyset$ **then**
    return *neúspěch*
  end if
  nedeterministicky zvol akci $a \in A$
  $s \leftarrow \gamma(s, a)$
  $\pi \leftarrow \pi \cdot a$
end loop

Algoritmus tedy prochází stavovým prostorem tak, že opakovaně vybírá aplikovatelné akce pro aktuální stav a posouvá se do stavů nových. Pokud stav splňuje cílové podmínky, vrátí se nalezený plán (seznam doposud aplikovaných akcí).

Pro implementaci algoritmu potřebujeme co nejefektivněji realizovat tyto operace:

- pro libovolný stav říci, zda je cílovým stavem (tj. splňuje cílové podmínky),
- pro libovolný stav naleznout množinu všech na něj aplikovatelných akcí,
- vytvořit nový stav vzniklý aplikací nějaké akce.

Algoritmus může vrátit spoustu různých plánů, některé mohou být i nekonečně dlouhé (cyklus na grafu – např. kdyby robot v doméně Gripper jezdil pořád dokola z místnosti A do místnosti B a zpět).

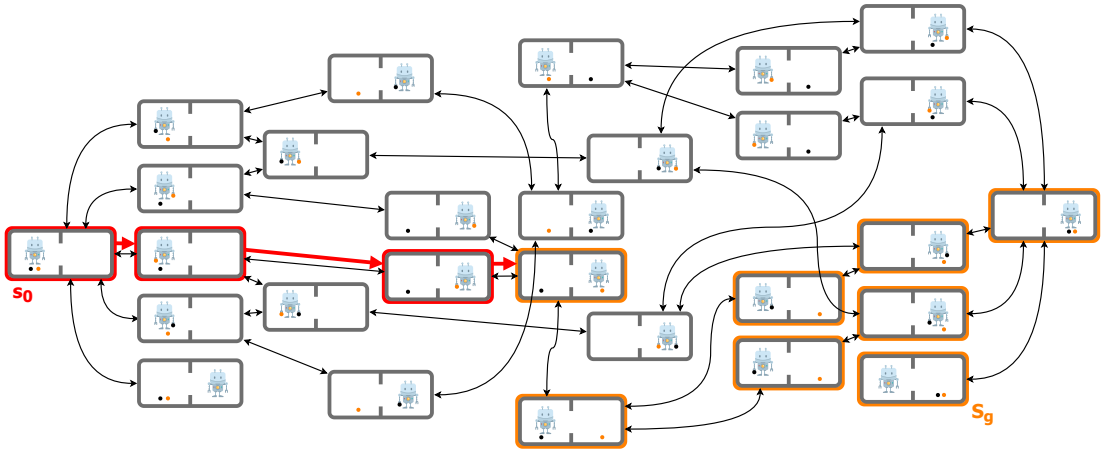
Deterministickou verzi algoritmu získáme např. prohledáváním grafu do šířky, do hloubky, A^* prohledáváním nebo hladovým algoritmem[5]. Prohledávání do šířky či A^* prohledávání dává optimální řešení, nicméně nároky na paměť rostou exponenciální řadou. Proto může být v praxi použitelnější prohledávání do hloubky či hladový algoritmus. Příklad postupu algoritmu ilustruje obrázek 3.2.

Prostor prohledávání je obvykle mnohem větší, než by bylo zapotřebí. Užívá se proto řada technik pro redukci větvícího faktoru při prohledávání, tzv. *prořezáváním* větví některé cesty prohledávání jednoduše zahodíme.

Nějakou techniku prořezávání nazveme *spolehlivou*, pakliže je garantováno, že nepřijdeme o všechna řešení daného problému. Pokud bereme navíc v potaz určitou míru optimality řešení, pak nazveme prořezávací techniku *silně spolehlivou*, pokud je zanecháno alespoň jedno optimální řešení.

Aby byl prohledávací algoritmus konečný, je třeba detekovat a odřezávat nekonečné větve prohledávání. Ve *Forward-Search* algoritmu nám k tomu postačí držet si seznam stavů, kterými jsme doposud prošli v aktuálním plánu, a pokud se dostaneme do již navštíveného stavu, tuto prohledávanou větev opustíme.

Dalším přístupem pro redukci větvícího faktoru je prohledávací algoritmus specifičtěji navádět, které akce při prohledávání vybírat. Toto navádění mohou zajišťovat tzv. *heuristické funkce*, jejichž použití podrobněji rozebereme později.



Obrázek 3.2: Příklad prohledávání algoritmu Forward-Search v doméně Gripper s dvěma míčky (cílovou podmínkou je oranžový míček v druhé místnosti).

Backward-Search

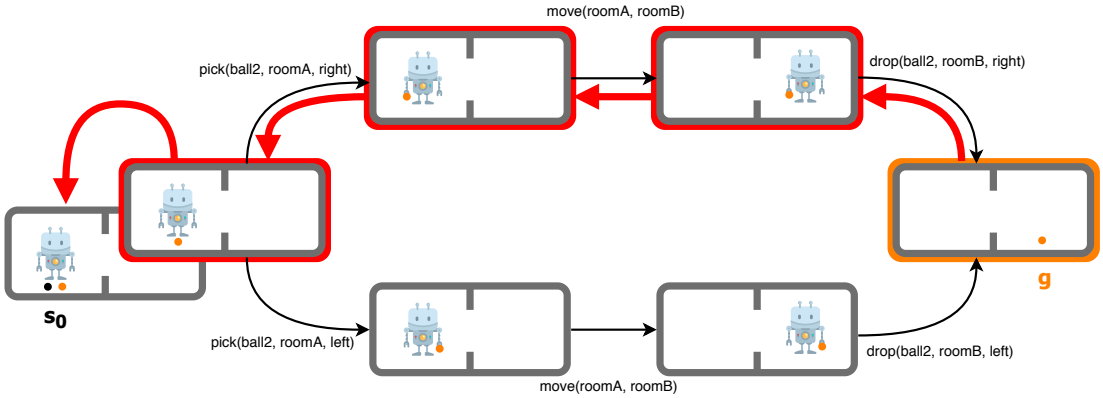
Druhou variantou prohledávání je *Backward-Search* (*zpětné hledání*). Hlavní myšlenkou je začít s cílovými podmínkami definovanými na vstupu a inverzními aplikacemi relevantních akcí získat nové cílové sub-podmínky, stejným způsobem pokračovat pro tyto sub-podmínky atd. Posouváme tedy potenciální cíle směrem k počátečnímu stavu a algoritmus skončí právě tehdy, když nějaká cílová sub-podmínka je splněna pro počáteční stav s_0.

Backward-Search(Op, s_0, g):
  $\pi \leftarrow$ prázdný plán
  loop
    if s_0 splňuje g **then**
      return π
    end if
    $R \leftarrow \{a \mid a$ je ground. instance z Op relevantní pro $g\}$
    if $R = \emptyset$ **then**
      return *neúspěch*
    end if
    nedeterministicky zvol akci $a \in R$
    $\pi \leftarrow a \cdot \pi$
    $g \leftarrow \gamma^{-1}(g, a)$
  end loop

Stejně jako pro Forward-Search lze detekovat a odřezávat nekonečné větve prohledávání tím způsobem, že budeme zaznamenávat již navštívené cílové sub-podmínky, a pokud dorazíme do nějaké g_k, pro kterou bude platit $g_i \subseteq g_k, i < k$, danou větev odřízneme.

Pro Forward-Search i Backward-Search lze snadno ukázat, že při těchto modifikacích zůstane algoritmus konečný a úplný. Je odříznuta každá nekonečná větev a zůstane alespoň jedna větev prohledávání obsahující nejkratší řešení.

Jak ilustruje obr. 3.3, algoritmus Backward-Search výrazně snížil větvící fak-



Obrázek 3.3: Příklad prohledávání algoritmu Backward-Search v doméně Gripper s dvěma míčky (cílovou podmínkou je oranžový míček v druhé místnosti).

tor oproti algoritmu Forward-Search. Aby se míček dostal na zem druhé místnosti, musí být v místnosti robot, který ho z jedné ruky pustí. Aby se tam robot s míčkem v ruce dostal, musí se přemístit z první místnosti. A konečně, aby mohl stát v první místnosti s míčkem v ruce, musel ho ze země sebrat – což už je podmnožina podmínek udávajících počáteční stav s_0 a skončili jsme.

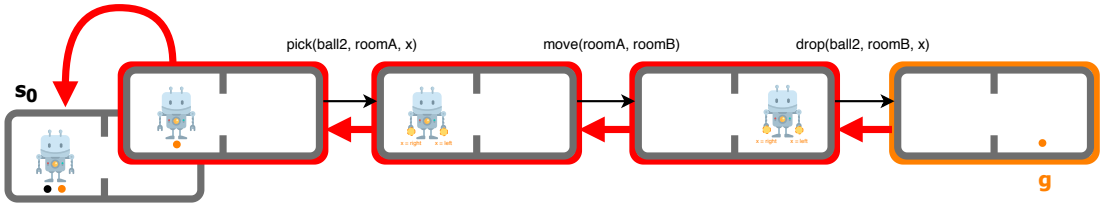
Vidíme, že jsme tímto postupem zcela odstínili akce spojené s černým míčkem, protože pro nás nejsou relevantní. Nicméně, také jde vidět, že např. nezáleží na tom, jakou rukou míček zvedneme. Můžeme jít proto v abstrakci ještě o kousek dále.

Lifted-Backward-Search

Adaptací zpětného prohledávání je tzv. *liftovaná* varianta. Místo groundovaných instancí operátorů relevantních pro g standardizujeme proměnné operátoru a unifikujeme ho s příslušnými atomy cílových podmínek g.

Lifted-Backward-Search(Op, s_0, g):
  $\pi \leftarrow$ prázdný plán
  loop
    if s_0 splňuje g **then**
      return π
    end if
    $R \leftarrow \{(op, \sigma) \mid op \in Op$ relevantní pro g, σ_1 je substituce standardizující proměnné op, σ_2 je nejobecnější společný unifikátor pro $\sigma_1(op)$ a ten atom z g, pro který je op relevantní, a $\sigma = \sigma_1\sigma_2\}$
    if $R = \emptyset$ **then**
      return *neúspěch*
    end if
    nedeterministicky zvol pár $(op, \sigma) \in R$
    $\pi \leftarrow \sigma(op) \cdot \sigma(\pi)$
    $g \leftarrow \gamma^{-1}(\sigma(g), \sigma(op))$
  end loop

Při prohledávání tedy groundujeme pouze některé proměnné a jiné necháváme



Obrázek 3.4: Příklad prohledávání algoritmu Lifted-Backward-Search v doméně Gripper s dvěma míčky (cílem je oranžový míček v druhé místnosti).

negroundované (tj. liftované). To nám ve výsledku opět zmenšuje prohledávaný prostor stavů, jak jde vidět na obr. 3.4. Z původního pekla algoritmu Forward-Search se tak stalo poměrně jednoduché vyhledávání. Nicméně, větvící faktor může být stále velký v případech, kdy existuje mnoho relevantních akcí pro cílové podmínky, přičemž mnoho z nich vůbec nemusí vést směrem ke stavu s_0. Jak už jsme zmínili, plánovači je v praxi potřeba pomoci např. heuristickými funkcemi.

Krátce zmiňme ještě jednu známou adaptaci, která se snaží dále prořezávat prostor hledání. Tzv. *STRIPS-algoritmus* je podobný algoritmu Backward-Search, ale v každém rekurzivním volání se snaží splnit předpoklady naposledy použitého operátoru přidaného do plánu coby cílovou sub-podmínku. Větvící faktor se sníží, ale algoritmus se stane neúplným. Známými příklady, se kterými má STRIPS-algoritmus problém, je prohození hodnoty dvou proměnných nebo tzv. *Sussmanova anomálie*[1] – jde o situace, kdy použití nějakého operátoru pro splnění jednoho cíle má jako vedlejší efekt ztrátu nějakého již dříve splněného cíle.

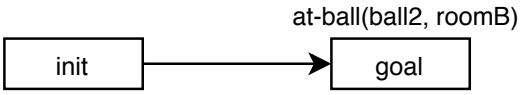
3.2 Plánování v prostoru plánů

Principiálně zcela odlišným přístupem k řešení problémů v klasickém plánování je plánování v prostoru plánů (*plan-space planning*). Motivací pro vznik byly nevýhody spojené s klasickým prohledáváním v prostoru stavů – i při nejlepších variantách algoritmu Backward-Search můžeme vždy dostat úlohu, u které bude větvící faktor příliš velký a algoritmus bude muset potenciálně projít všechny slepé větve výpočtu. Ideou nového přístupu je tzv. *least-commitment strategy*, tj. neprovádět instanciace, grounding, řazení akcí apod. až do doby, kdy to není nezbytně nutné.

Prohledávaný prostor již nebude obsahovat jednotlivé stavy, ale tzv. *parciální plány* (nebo též *částečně specifikované plány*). Orientovanými hranami v odpovídajícím grafu budou tzv. *operace vylepšující plán*. Parciální plány budeme neustále *vylepšovat*, dokud nebudou mít žádné *vady* – a takové parciální plány jsou pak řešením plánovacího problému.

Klasický plán je množinou akcí organizovanou do určité struktury. Parciální plán je podmnožinou těchto akcí, zároveň podmnožinou této organizační struktury (časové řazení akcí) a podmnožinou vázaných proměnných.

Definice 29 (parciální plán)**.** *Parciálním plánem nazveme* $\pi = (A, \prec, B, L)$*, kde* A *je množinou částečně instanciovaných akcí,* $\prec$ *je částečným uspořádáním na množině* A (časové řazení, ordering constraints)*,* B *je množinou* vazebních omezení *na proměnných* (variable bindings) *ve tvaru* $x = y$*,* $x \neq y$*, či* $x \in D_x$*, a* L



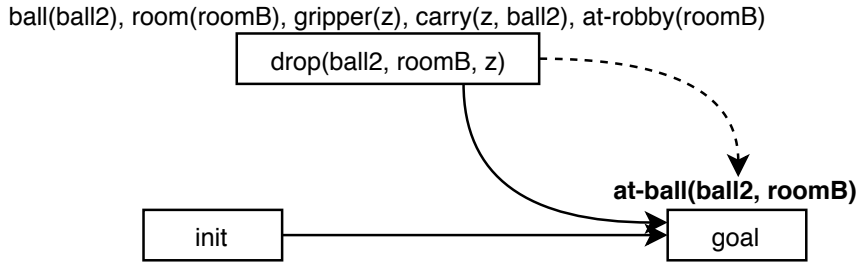
Obrázek 3.5: Startovní parciální plán domény Gripper

je množinou kauzálních spojení (causal links) *ve tvaru* $\langle a_i, a_j(p)\rangle$*, kde* $a_i, a_j \in A$*,* $a_i \prec a_j$*, predikát* p *je součástí efektů* a_i *a zároveň předpokladů* a_j *a vazebná omezení na proměnných v* p *jsou součástí* B*.*

Startovním plánem je $\pi_0 = (\{init, goal\}, \{(init \prec goal)\}, \{\}, \{\})$, viz též obr. 3.5 ilustrující doménu Gripper s dvěma míčky, kde cílovou podmínkou je druhý míček ve druhé místnosti. Akce init a goal jsou fiktivními akcemi – init je bez předpokladů a jeho efekty odpovídají počátečnímu stavu problému, zatímco akce goal má za předpoklady cílové podmínky a nemá žádné efekty. Dále vidíme jediné řadící omezení (init nastane před goal) a zatím žádná kauzální spojení a vázané proměnné.

Aplikací *vylepšující operace* (*plan refinement operation*) na parciální plán získáme nový parciální plán. Takovou operací může být: přidání nové akce do A, přidání řadící podmínky do $\prec$, přidání vazebné podmínky do B, nebo přidání kauzálního spojení do L.

Akce se přidávají do plánu pro naplnění doposud nesplněných předpokladů, viz obr. 3.6 – nový operátor realizuje cílovou podmínku a nově nenaplněnými předpoklady se stávají předpoklady této nové akce. Ve stejnou chvíli přidáváme kauzální spojení pro realizovanou cílovou podmínku (přerušovaná čára) a řadící podmínku (plná čára). Doposud nenaplněné předpoklady v parciálním plánu se též nazývají *otevřené cíle*.



Obrázek 3.6: Přidání operátoru realizujícího cílovou podmínku

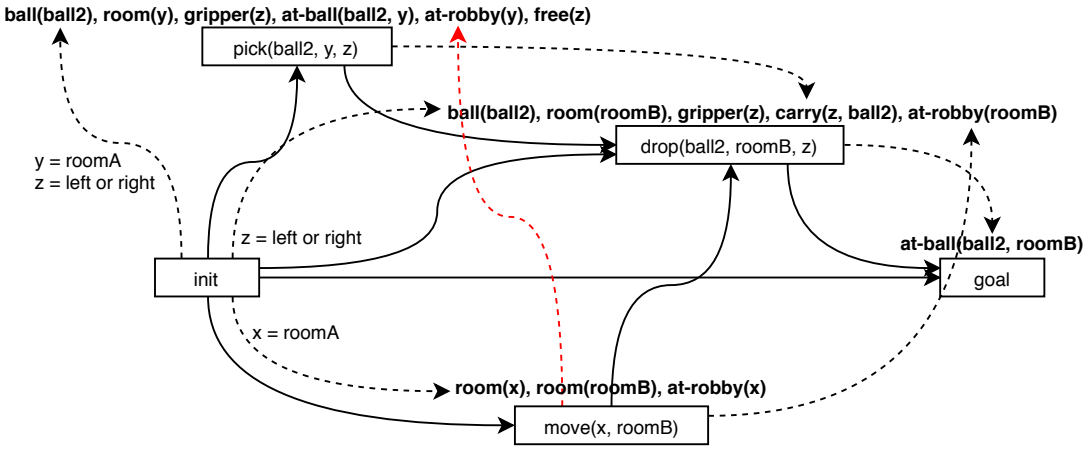
Kauzální spojení se do plánu přidávají pro zabránění interferencí s jinými akcemi a zároveň implikují řadící omezení. Vazby proměnných přidáváme do plánu, abychom groundovali operátory do akcí a unifikovali efekty akcí s naplňovanými předpoklady.

Během vytváření nových parciálních plánů mohou vznikat tzv. *hrozby*.

Definice 30 (hrozba v plánu)**.** *Akce* a_k *v parciálním plánu* $\pi = (A, \prec, B, L)$ *je* hrozbou (threat) *pro kauzální spojení* $\langle a_i, a_j(p)\rangle$*, právě když* a_k *má efekt* $\neg q$*, který je potenciálně nekonzistentní s* p *(tj.* p *a* q *jsou unifikovatelné), řadící omezení* $(a_i \prec a_k)$ *a* $(a_k \prec a_j)$ *jsou konzistentní s* $\prec$ *a vazebné podmínky pro unifikaci* q *a* p *jsou konzistentní s* B*.*

Definice 31 (vada plánu)**.** Vadou *parciálního plánu* $\pi = (A, \prec, B, L)$ *je buďto existující* otevřený cíl *(tj. nenaplněná podmínka v předpokladech některé z akcí v* A*), anebo hrozba.*

Příklad parciálního plánu s hrozbou ilustruje obr. 3.7 – efekt operace move může rozbít předpoklad operace pick. Této hrozby se zbavíme tak, že se přidá nová řadící podmínka (pick musí proběhnout před move).



Obrázek 3.7: Hrozba v plánu

Parciální plán (nedeterministicky) vylepšujeme tak dlouho, dokud není bez vad. Platí, že parciální plán $\pi = (A, \prec, B, L)$ je řešením plánovacího problému $\mathbf{\Sigma} = (S, A, \gamma)$, pokud π nemá žádné vady, řadící omezení neobsahují cykly a vazby proměnných B jsou konzistentní.

V parciálním plánu máme k dispozici stále pouze částečné uspořádání na částečně instanciovaných akcích, zatímco pro plán potřebujeme přesnou sekvenci groundovaných akcí. Nicméně platí, že částečné uspořádání odpovídá úplnému uspořádání, ve kterém jsou respektovány všechny podmínky z uspořádání částečného, a částečná instanciace odpovídá groundování, ve kterém jsou přiřazené hodnoty konzistentní s vázanými podmínkami na proměnných.

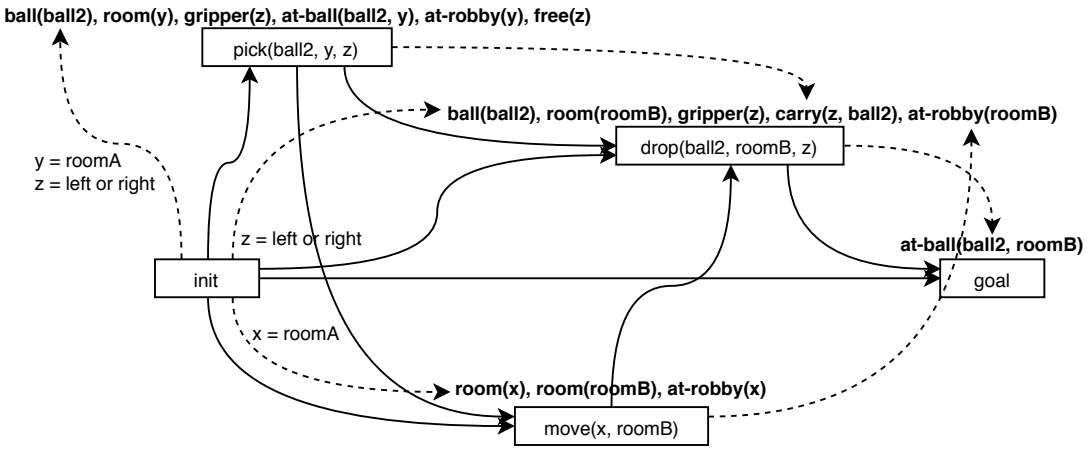
Obrázek 3.8 ilustruje kompletní parciální plán domény Gripper, z něhož se již snadno získá sekvence groundovaných akcí.

Postup algoritmu prohledávající prostor plánů jsme již víceméně načrtli výše, nicméně shrňme ho ještě jednou v následujícím pseudokódu. Principem je neustále vylepšovat plán odstraňováním vad, při udržování konzistentního řazení a vazeb proměnných.

Plan-Space-Planner(π):
  VadyPlánu $\leftarrow$ OtevřenéCíle(π) $\cup$ Hrozby(π)
  if VadyPlánu $= \emptyset$ **then**
    return π
  end if
  Vyber libovolné $\phi \in$ VadyPlánu
  Řešiče $\leftarrow$ Vyřešit(ϕ, π)
  if Řešiče $= \emptyset$ **then**

return *neúspěch*
end if
Nedeterministicky vyber $\rho \in$ Řešiče
$\pi' \leftarrow$ VylepšiPlán(ρ, π)
return Plan-Space-Planner(π')

Algoritmus je úplný a konečný. Pro výběr konkrétních vad i výběr konkrétního řešiče vady existují pokročilé techniky, které v tuto chvíli nebudeme dále rozvádět. Principem této sekce bylo pouze nastínit základní koncept vyhledávání v prostoru plánů.



Obrázek 3.8: Finální parciální plán pro doménu Gripper

Ve své době představovala koncepce plánování v prostoru stavů malou revoluci. Plánovač má při svém běhu k dispozici více flexibility, umožňuje lépe pracovat např. s omezenými zdroji, ale výpočet je komplikovanější. Zvláště výhodné je použití parciálních plánů v doméně distribuovaných a multi-agentních systémů, do kterých svým charakterem přirozeně zapadají.

Část II

Framework PAD

4. O frameworku PAD

Kapitola pojednává o obecných aspektech frameworku PAD – proč byl vyvíjen, k čemu by měl sloužit, jaké vlastnosti jsou od něho požadovány apod. Nastíněny jsou principy a metodiky z oboru softwarového inženýrství použité při návrhu frameworku a samotné implementaci a proč je dobré se jich držet.

4.1 Účel frameworku

Primárním cílem bylo vytvořit efektivní nástroj pro výzkumníky v oblasti plánování, který jim umožní snadný návrh a testování nových teorií a algoritmů. Vzniknout by postupně měl bohatý framework, dostupný volně pod licencí GNU GPL[6]. Důraz je kladen na kvalitní objektový návrh, umožňující snadnou rozšiřitelnost, a také efektivnost a přehlednost implementace. Projekt má své vlastní webové stránky s podrobnou dokumentací.

K dosažení výše uvedených kvalitativních aspektů byla využita celá řada principů a metodik z oblasti softwarového inženýrství, které budou níže dále popsány.

Vznikající software byl pojmenován *PAD Framework*, kde *PAD* je zkratka pro *Planning Algorithms Development*.

Framework podporuje dvě významné vstupní reprezentace – tou první je PDDL, de facto standard pro popis problémů v oblasti plánování (mimo jiné), druhou je formalismus SAS^+ využívající stavy s vícehodnotovými proměnnými. Obě reprezentace mají svá specifika a popisu jejich řešení se věnuje významná část této práce.

4.2 Využité prostředky

Prostředí

Pro vývoj frameworku byl zvolen programovací jazyk C# a platforma *.NET* (produkt je kompatibilní s .NET verzí 4.5.2 a vyšší) – zde chceme především vyjít vstříc uživatelům, pro které je PAD primárně vyvíjen, a ti pracují zejména na této platformě a programovacím jazyku. Alternativou pro spuštění a použití knihoven PADu může být multi-platformní open-source projekt Mono[7].

Vývojovým prostředím bylo zvoleno *Microsoft Visual Studio 2019* verze *Community* – volně dostupná edice IDE spolu s kolekcí vývojářských nástrojů, která je zdarma pro nekomerční použití. Pro testování byl použit integrovaný testovací framework *Visual Studio Unit Testing Framework*.

Detailům instalace, spuštění a používání frameworku PAD se detailně věnuje programová dokumentace, která je nedílnou součástí produktu.

Externí knihovny

Při vývoji byly použity externí knihovny projektu *Irony*[8], verze 0.9.1. Projekt Irony je vývojový toolkit pro implementaci jazyků v prostředí .NET pod volnou MIT licencí. Knihovny byly využity při konstrukci komponenty PDDL parseru – více detailů v příslušné kapitole 5.4 (*Načítání a validace PDDL*).

Dále byl použit balíček *Wintellect Power Collections*[9] pro efektivní implementaci jedné z hald dostupných v projektu `PAD.Planner` – viz kapitola 6.2.3.

4.3 Vývoj SW systému, použité principy

Následující sekce popisuje principy a metody použité při návrhu a implementaci vznikajícího frameworku.

Klasický proces vývoje softwarového systému (tzv. *waterfall*) se skládá z několika dobře definovaných fází[10] – sběr požadavků, analýza a tvorba specifikace, architektura a návrh, implementace, validace a testování, a příp. fáze údržby.

Tyto fáze si nyní krátce popíšeme a zasadíme do kontextu vývoje frameworku PAD. Tvorba knihoven či frameworku se logicky liší od komerční tvorby klasického softwarového systému, nicméně styčných ploch tu najdeme mnoho.

Specifikace, požadavky na systém

V první etapě SW vývoje je typicky cílem intenzivní komunikace se zákazníkem pro maximální vyjasnění požadavků na konečný produkt a stanovení rozsahu prací. Vytvářejí se různé vizuální uživatelské modely (např. UML diagramy případů užití[11]) a strukturované dokumenty, obsahující též požadavky na validaci konečného programu. Výstupem je pak dokument typicky nazývaný *specifikace systému*.

Nashromážděná data pak slouží jako podklad pro odhady prací a nákladů, jsou podkladem pro uzavření obchodní smlouvy a slouží též jako vstupní data pro další fázi vývoje. Jakmile je uzavřena smlouva, specifikace jasně říká, co se má udělat a mnohdy i též explicitně, co se udělat nemá (pro omezení šedé zóny – zákazník chce často vše, co není explicitně „ne“, zatímco dodavatel potřebuje dělat jen na tom, co je explicitně „ano“ – což má velký dopad na plánování, dostupné zdroje, čas, cenu díla atd.).

Naším pomyslným zákazníkem je katedra KTIML fakulty MFF UK a zadání této diplomové práce by se s trochou nadsázky dalo kvalifikovat jako poptávka od zákazníka.

Z rozhovorů s vedoucím práce vyplynul očekávaný rozsah prací a přesněji definované požadavky. Ty si nyní můžeme bodově uvést dle jejich kategorie (funkční/mimofunkční). Ve specifikaci bychom požadavky měly uvedeny v katalogu požadavků ve formě tabulky, detailněji rozepsané a doplněné identifikátorem a prioritou pro jednotlivé požadavky.

Funkční požadavky na framework PAD:

- Vytvoření systému pro vývoj optimalizačních algoritmů z oblasti plánování.
- Systém podporuje modelovací jazyky PDDL a SAS$^+$.
- Systém obsahuje vhodné nástroje pro práci s těmito vstupními formáty.
- Systém podporuje primárně prohledávání v prostoru stavů.
- Systém implementuje základní prohledávací algoritmy a heuristiky.
- Systém poskytuje nástroje pro tvorbu nových prohledávacích metod.

- Systém poskytuje nástroje pro prezentaci a analýzu výsledků.
- Systém umožňuje hromadné spouštění experimentálních výpočtů,
- Systém je ověřen na sérii optimalizačních problémů.

Mimofunkční a jiné požadavky na framework PAD:

- Cílovým jazykem systému je C# a platforma .NET.
- Systém podporuje rozšiřování pro nové metody, algoritmy a heuristiky.
- Systém má robustní, přehledný objektový návrh podporující rozšiřitelnost.
- Je kladen důraz na efektivní implementaci algoritmů a datových struktur.
- Systém je kvalitně dokumentován, na úrovni kódu i klasické dokumentace.
- Systém je umístěn ve vlastním repozitáři.
- Systém má vlastní wiki stránku s on-line dokumentací.
- Systém je volně dostupný pod jednou z licencí pro svobodný software.

Architektura a návrh

Následuje fáze analýzy nashromážděných dat a převod uživatelských modelů a požadavků do modelů programových. Navrhovaný systém se dekomponuje na části řešící požadované funkce, načrtne se komunikace mezi těmito komponentami a datové toky. Jednotlivé komponenty se dále dekomponují na menší části, dokud nejsme komponenty schopni relativně snadno implementovat. Pokud systém využívá nějaké databáze, navrhne se datový model. Obecně do této fázi zapadá vše, co je třeba udělat a rozmyslet, něž se začne psát kód.

Termíny architektura a návrh (*design*) jsou si významem dosti blízké a jejich náplň se částečně překrývá. Obecně lze říci, že do architektury spadá realizace nefunkčních požadavků (tj. požadavky na výkon, rozšiřitelnost, škálovatelnost apod.), zatímco návrh řeší naopak funkční požadavky.

Praxe ukazuje, že k popisu systému je velmi užitečný jazyk UML (*Unified Modeling Language*)[11] – což je sada standardizovaných diagramů pro popis struktury a chování systému. UML je dnes de facto již průmyslovým standardem a diagramové znázornění různých vztahů je často velmi elegantní, názorné a stručné.

Pro popis struktury systémů se často využívají UML diagramy komponent, digramy balíčků či diagramy tříd. Pro popis chování systému pak např. stavové diagramy nebo diagramy aktivit.

Osvědčenými nástroji při návrhu programových komponent a silnými nástroji pro dosažení těchto požadavků jsou tzv. *návrhové vzory*[12]. Jde o soubor pojmenovaných technik využívajících vlastností objektově orientovaného programování (OOP). Použití různých návrhových vzorů budeme dále v textu často zmiňovat a odkazovat se na ně.

Známý je i tzv. *SOLID* návrh – jde o akronym popisující pět klíčových návrhových principů pro dosažení pochopitelného, flexibilního a udržovatelného softwaru[13].

Principy S.O.L.I.D.:

- **S**ingle Responsibility Principle (princip jednotlivé odpovědnosti) – říká, že každá třída by měla mít pouze jednu jedinou odpovědnost; jedna potenciální změna ve specifikaci softwaru by měla mít dopad na specifikaci jedné třídy,
- **O**pen/Closed Principle (princip otevřenosti/uzavřenosti) – říká, že softwarové entity by měly být otevřené pro rozšiřování, ale uzavřené pro modifikace,
- **L**iskov Substitution Principle (Liskovové princip zastoupení) – říká, že objekty v programu by měly být nahraditelné instancemi svých podtypů, bez rozbití korektnosti programu,
- **I**nterface Segregation Principle (princip oddělených rozhraní) – říká, že mnoho rozhraní uzpůsobených pro jednotlivé klienty je lepší řešení než jedno obecné super-rozhraní,
- **D**ependency Inversion Principle (princip inverzní závislosti) – říká, že je záhodno tvořit závislosti na abstrakcích, nikoli konkrétních entitách.

Bližší rozvedení těchto principů necháme čtenáři na další studium odkazované literatury[13]. Existují ještě další, méně známé principy (např. GRASP), různé anti-patterny (čemu se při implementaci vyhnout) apod. Při implementaci frameworku PAD se výše zmíněnými technikami a postupy snažíme maximálně řídit.

Statistiky ukazují, že fáze analýzy i návrhu se často zanedbávají a nejsou vždy prioritou při reálné tvorbě SW systému, což má ovšem neblahé následky v pozdějších fázích, implementaci a údržbě. Čím později ve vývojovém cyklu se totiž přijde na potenciální chybu, tím dražší je její napravení.

Co je vlastně *framework*?

Pokud se zabýváme architekturou, je správný čas upřesnit, co vlastně znamená termín *framework* a jaký je rozdíl mezi frameworkem a *softwarovou knihovnou*.

Framework představuje nějaký znovupoužitelný návrh pro SW systém, je určitým základem pro vývoj nových aplikací, který diktuje architekturu tohoto nového systému – určuje dekompozici systému a jak budou jednotlivé komponenty komunikovat. Je tedy jakousi kostrou systému.

V základní dekompozici se rozlišují tzv. *frozen spots* (neměnné části, definující celkovou architekturu) a *hot spots* (abstraktní třídy, virtuální metody, zajišťující rozšiřitelnost).

Knihovna je oproti tomu obecné pojmenování pro kolekci nějaké znovupoužitelné funkcionality. Hlavní rozdíl mezi těmito dvěma přístupy je však princip volání – zatímco knihovnu volá přímo uživatel, framework de facto volá uživatele (tzv. *Inversion of Control*[14]) a očekává, že mu na některých místech uživatel doplní funkcionalitu (v opačném případě má nějaké defaultní chování).

Framework je tedy obecně komplexnější systém, ve kterém se uživatel frameworku nemusí zabývat architektonickými otázkami, ale přímo doplňuje funkcionalitu příslušnou k doméně, ve které pracuje.

Implementace

Při psaní samotného kódu je nutné dodržovat určité konvence, pro dosažení dobré čitelnosti a přehlednosti pro další uživatele SW systému. Je dobré si uvědomit, že kód píšeme pouze jednou, nicméně číst ho někdo bude typicky mnohokrát. Těžko se důvěřuje nesrozumitelnému, *ošklivému* kódu, který jsme nuceni použít (nebo hůře – opravit). Pokud bychom tedy měli nějak definovat kvalitní kód, pak to budou přívlastky srozumitelný a udržovatelný.

Pro doménu .NET je dobrým výchozím bodem soubor doporučení pro kódové konvence od Microsoftu na jejich oficiální on-line dokumentaci[15]. V týmu je téměř nutné nějaké konvence kódu mít zavedené a vynucovat je, ať už prostřednictvím revizí kódu (kontrola od jiného vývojáře) či automatizovaných nástrojů (např. statická analýza kódu).

Velmi kvalitním materiálem popisujícím dobré principy při návrhu a psaní kódu je kniha *Dokonalý kód* od Steva McConnella[16], která je referovaným zdrojem prakticky všech kurzů či článků týkajících se dobré kvality kódu.

Dokumentace

Dokumentace je důležitou součástí všech fází procesu vývoje SW systému, může mít však různou formu. Potřeba je dokumentovat prvotní rozhovory se zákazníkem, sepisovat katalog požadavků a specifikaci a dále příslušná rozhodnutí při tvorbě architektury a designu. Doporučení z praxe zní nevytvářet *viktoriánské romány*, ale jít co nejvíce cestou strukturovaného dokumentu a vizuálních reprezentací pomocí diagramů a obrázků.

Samotný zdrojový kód by v ideálním případě měl být samopopisný, dobré je mít okomentované hlavičky funkcí v některém z normovaných formátů, který následně umožňuje automatické vygenerování programové dokumentace (např. *doxygen*). Takto vygenerovaná dokumentace se mnohem snadněji udržuje v aktuálním stavu a lze ji umístit například na příslušné webové stránky projektu.

Velmi důležitá je dokumentace zejména ve fázi údržby, kdy je potřeba provést v systému opravu nalezené chyby nebo přidat novou funkcionalitu. Takovou práci dělají často úplně jiní lidé, než jsou autoři původního systému. Bez dokumentace prakticky není možno v dlouhodobém horizontu SW systém spravovat.

Součástí odevzdávaného projektu je kromě programové též uživatelská dokumentace popisující užití systému přímo externím uživatelem a měla by podle toho být formulována vhodnou formou.

Validace, testování

Po úspěšné implementaci přichází na řadu fáze testování (resp. určité formy testování mohou probíhat už během vývoje) – tzn. ověřování korektnosti fungování a verifikace výsledků, které program vrací. Typicky se konečná validace a verifikace provádí vůči požadavkům ze specifikace, mohou být uvedeny i verifikační příklady, které musí konečný produkt splňovat, aby mohl být schválen a odevzdán.

Dobrou praxí je při softwarovém vývoji tvorba automatických testů – jednou z forem mohou být tzv. *unit-testy*, ověřující funkci jednotlivých programových komponent, psané samotnými programátory. Z krátkodobého hlediska nemusí

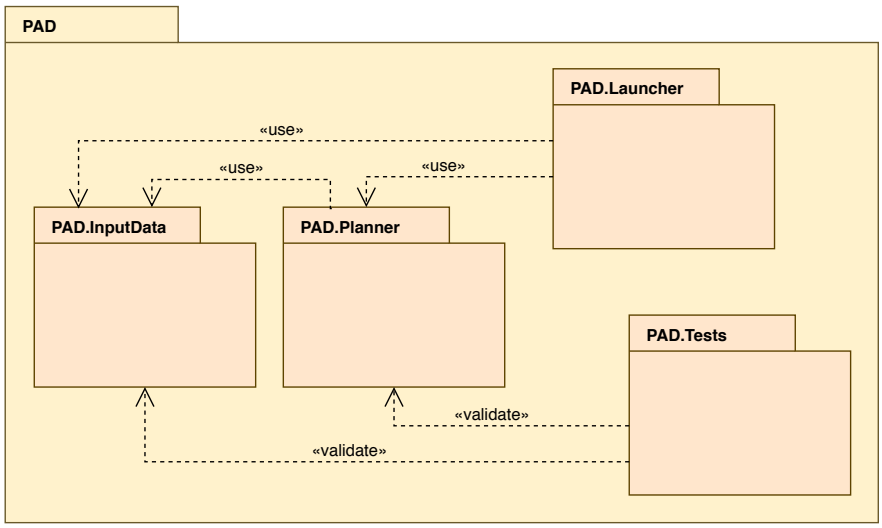
být přínos unit-testů patrný, nicméně při delším vývoji i následných modifikacích se jejich přítomnost značně projeví.

Přítomnost testovacího balíku u produktu zvyšuje důvěru v softwarové dílo a zvyšuje bezpečnost pro refactoring i budoucí změny a rozšiřování. Framework PAD přímo obsahuje speciální testovací projekt pokrývající unit-testy všechny klíčové komponenty frameworku.

Údržba

Fáze údržby (maintenance) je v praxi nejdelší doba životního cyklu SW systému. Zásah do systému, ať už z důvodu chyby či požadavku na novou funkcionalitu, je v době údržby nejdražší a nejnebezpečnější (vysoké riziko rozbití systému apod.). Pro redukci těchto nákladů je třeba dobře promyšlená architektura, kvalitní dokumentace, kvalitní implementace a automatické regresní testy.

4.4 Struktura frameworku PAD



Obrázek 4.1: Struktura frameworku PAD

Diagram balíčků na obrázku 4.1 zachycuje velmi hrubě strukturu frameworku PAD ve formě solutionu Visual Studia, obsahujícího následující čtyři hlavní součásti (projekty):

- `PAD.InputData` – projekt zapouzdřující reprezentace vstupních dat ve všech podporovaných vstupních formátech, významnou součástí jsou načítače dat ze souborů a validátory konzistence zadaných/načtených dat; projekt lze použít i externě pro pouhé načítání PDDL a SAS$^+$ vstupů, bez nutnosti dále používat další součásti frameworku,
- `PAD.Planner` – stěžejní část frameworku realizující plánovač, prohledávání stavového prostoru, s možností využití různých heuristik a metod prohledávání a umožňující implementace vlastních, nových řešení,

- `PAD.Launcher` – vstupní bod programu celého frameworku PAD, realizující komponenty pro hromadného spouštění rozsáhlejších experimentů; poskytuje možnosti pro parametrizované generování plánovacích úloh, které jsou následně paralelně zpracovány,
- `PAD.Tests` – projekt validující všechny kritické komponenty frameworku prostřednictvím unit-testů, jak z *black-box* tak i *white-box*[10] perspektivy, včetně komplexních dávkových testů založených přímo na oficiálních příkladech z *Mezinárodní plánovací soutěže* (IPC); projekt je určený mimo jiné pro zvýšení důvěry v korektnost a funkčnost frameworku, významně pomáhá též při rozšiřování funkcionality.

5. Vstupní data PADu

V této kapitole se seznámíme s podporovanými formalismy vstupních dat PDDL a SAS$^+$ a jejích reprezentací v rámci frameworku PAD. Popsán je dále *loader*, načítající v několika fázích data ze vstupních souborů, které následně transformuje do struktur PADu. Po načtení jsou vstupní data zkontrolována validátorem.

5.1 Podporované formalismy

Jazyk PDDL

PDDL (*Planning Domain Definition Language*) je de facto standardem pro specifikaci problémů v doméně plánování. První verze tohoto jazyka vznikla již v roce 1998 pod záštitou Mezinárodní plánovací soutěže (IPC) a je oficiálním jazykem této soutěže dodnes. Inspirován byl staršími formalismy STRIPS a ADL.[17]

PDDL data jsou rozdělena do dvou částí - první je definice *domény*, druhá pak definice *problému*, příslušející k dané doméně. Jednoduše řečeno, doména popisuje celkové univerzum a možnosti v tomto univerzu, zatímco problém určuje konkrétní počáteční a koncové podmínky plánovacího problému. Dá se říci, že PDDL problém je instancí PDDL domény. Dohromady tyto dva vstupy tvoří tzv. *PDDL-model plánovacího problému*. Pro jednu doménu může zjevně existovat mnoho specifických problémů.

Jazyk PDDL přímo reprezentuje klasickou reprezentaci popsanou v první části, v podkapitole 2.1. Čili si můžeme přímo ukázat, jak bude vypadat ilustrační *Gripper* doména v jazyku PDDL.

```
(define (domain gripper-strips)
  (:predicates (room ?r)
               (ball ?b)
               (gripper ?g)
               (at-robby ?r)
               (at ?b ?r)
               (free ?g)
               (carry ?o ?g))

  (:action move
    :parameters (?from ?to)
    :precondition (and (room ?from)
                       (room ?to)
                       (at-robby ?from))
    :effect (and (at-robby ?to)
                 (not (at-robby ?from))))

  (:action pick
     :parameters (?obj ?room ?gripper)
     :precondition (and (ball ?obj)
                        (room ?room)
```

```
                        (gripper ?gripper)
                        (at ?obj ?room)
                        (at-robby ?room)
                        (free ?gripper))
    :effect (and (carry ?obj ?gripper)
                 (not (at ?obj ?room))
                 (not (free ?gripper))))

  (:action drop
    :parameters (?obj ?room ?gripper)
    :precondition (and (ball ?obj)
                       (room ?room)
                       (gripper ?gripper)
                       (carry ?obj ?gripper)
                       (at-robby ?room))
    :effect (and (at ?obj ?room)
                 (free ?gripper)
                 (not (carry ?obj ?gripper)))))
```

Zdrojový kód 5.1: PDDL doména pro Gripper

```
(define (problem strips-gripper-x-1)
   (:domain gripper-strips)
   (:objects rooma roomb ball4 ball3 ball2 ball1 left right)
   (:init (room rooma)
          (room roomb)
          (ball ball4)
          (ball ball3)
          (ball ball2)
          (ball ball1)
          (at-robby rooma)
          (free left)
          (free right)
          (at ball4 rooma)
          (at ball3 rooma)
          (at ball2 rooma)
          (at ball1 rooma)
          (gripper left)
          (gripper right))
   (:goal (and (at ball4 roomb)
               (at ball3 roomb)
               (at ball2 roomb)
               (at ball1 roomb))))
```

Zdrojový kód 5.2: PDDL problém pro Gripper se 4 míčky

PDDL obsahuje též podporu typování objektů, podmíněné efekty, dále numerické fluenty, definování metriky, durative-akce[18], odvozené predikáty[19], časově omezující podmínky, preference a objektové fluenty[20][21]. Poslední verze jazyka je PDDL 3.1, jehož načítání framework PAD plně podporuje.

Formalismus SAS$^+$

SAS$^+$ je plánovací formalismus založený na použití více-hodnotových stavových proměnných. Zformulován byl již v roce 1995[22][23], a ačkoli se mu nedostává popularity jazyka PDDL, zájem o zkoumání možností tohoto systému časem roste. V základu jde o klasickou *state-variable* reprezentaci plánovacího problému popsanou v sekci 2.3, rozšířenou o možnosti zadání axiomů, podmíněných efektů, či mutexových skupin.

Formalismus SAS$^+$ má ustálený vstupní formát[24], který je však poměrně rigidní. Vstupní data jsou specifikována v několika blocích, kde jednotlivé řádky definují hodnoty pro pevně dané parametry. Kompletní specifikaci vstupního formalismu obsahuje též elektronická příloha této práce.

Reprezentaci SAS$^+$ používá jeden ze známých plánovačů *Fast Downward*[25], který zároveň obsahuje nástroj pro transformaci PDDL vstupů do SAS$^+$.

Vstupní soubor umožňuje definovat:

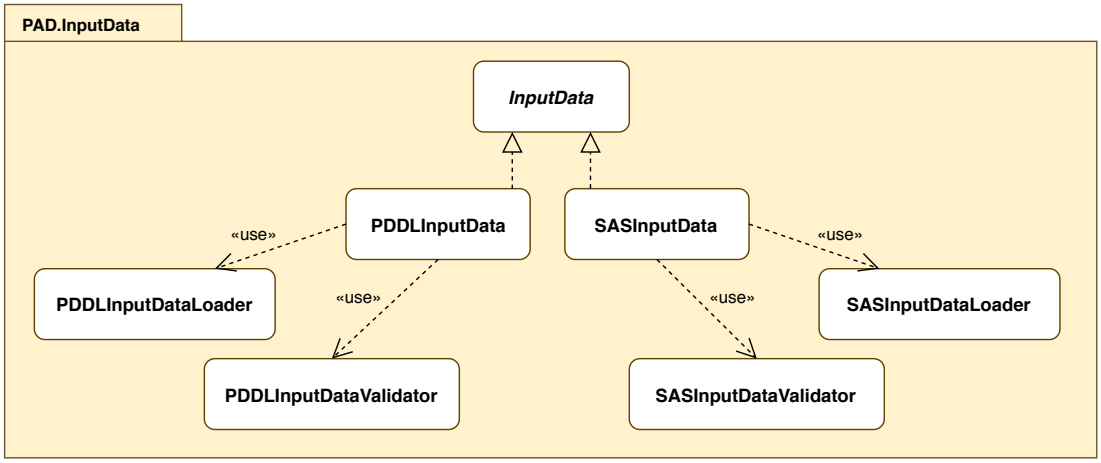
- verzi vstupního souboru,
- zda má být použita metrika – tj. zohlednění cen operátorů,
- proměnné – zda jde o axiomatickou proměnnou, rozsah domény a symbolickou reprezentaci jednotlivých hodnot,
- mutexové skupiny – vzájemně se vylučující podmínky, tj. které kombinace proměnná–hodnota nesmí nastat zároveň,
- počáteční stav,
- podmínky pro cílové stavy,
- operátory – podmínky aplikovatelnosti, efekty, příp. podmíněné efekty,
- axiomy – automaticky vyhodnocované vztahy.

```
begin_version
3
end_version
begin_metric
0
end_metric
1
begin_variable
var0
-1
3
Atom at(ball1, rooma)
Atom at(ball1, roomb)
<none of those>
end_variable
begin_state
0
...
```

Zdrojový kód 5.3: Ukázka části vstupu SAS$^+$

5.2 Projekt `PAD.InputData`

Pro reprezentaci vstupních dat a práci s těmito daty byl vytvořen speciálně projekt `PAD.InputData`. Hrubá struktura projektu je vyobrazena v diagramu tříd na obr. 5.1.



Obrázek 5.1: Struktura projektu `PAD.InputData`

Ústředními třídami projektu jsou `PDDLInputData` a `SASInputData` odvozené od abstraktní `InputData`, které samozřejmě zapouzdřují vstupní data pro PDDL a SAS$^+$ reprezentaci.

Projekt dále obsahuje třídy `PDDLInputDataLoader`, `SASInputDataLoader` poskytující funkcionalitu načítání vstupů ze souborů a `PDDLInputDataValidator`, `SASInputDataValidator` zapouzdřující validaci vstupních dat. Třídy vstupních dat si tuto funkcionalitu pak samy volají – rozdělení je logicky provedeno tak, aby byly seskupeny související služby a data v izolovaných komponentách. Uživatele typicky budou zajímat pouze naplněná vstupní data a od procesu načítání a validace by měl být co nejvíce odstíněn.

Zavoláním konstuktoru `PDDLInputData("domain.pddl", "problem.pddl")` program defaultně načte vstupní soubory, přetransformuje je, naplní struktury `PDDLInputData` a provede validaci dle specifikace PDDL. Program nicméně dává uživateli flexibilitu např. v tom, že si může vytvořit vstupní data prázdná, ručně si je vyplnit a zavolat validaci. Případně si později může k dané doméně načíst jiný vstupní problém bez nutnosti vytvářet celý objekt od začátku.

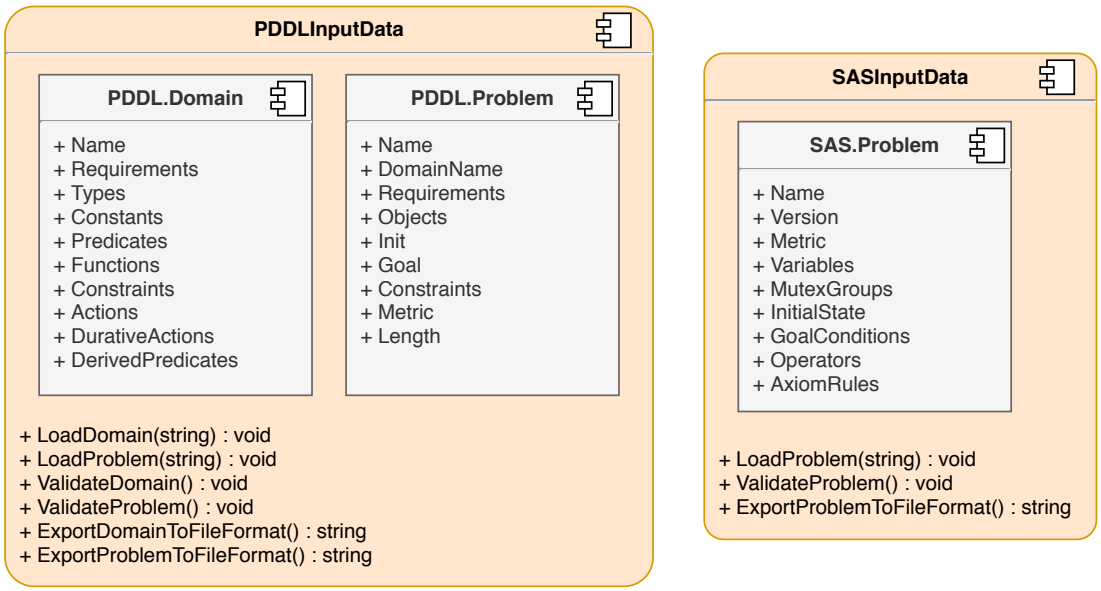
Pro SAS$^+$ reprezentaci je logika načítání analogická, pouze se použije jeden vstupní soubor problému (není zde zvlášť soubor domény jako v případě PDDL).

K podrobnějšímu popisu jednotlivých komponent se dostaneme v následujících sekcích. Komponenty loaderů jsou podpořeny unit-testingem důležitých součástí v rámci speciálního testovacího projektu `PAD.Tests`.

5.3 Struktura vstupních dat

Objekty `PDDLInputData` a `SASInputData`, zapouzdřující data pro oba vstupní formalismy, se ve své stromové hierarchii přirozeně podobají obsahu PDDL/SAS$^+$ souborů, viz obr. 5.2.

Kromě metod pro jednoduché načítání dat ze vstupních souborů a validaci dat, podporují tyto třídy i funkcionalitu zpětného převodu do původních PDDL/SAS$^+$ formalismů. Lze je tedy klidně použít takovým způsobem, že si uživatel manuálně naplní tyto struktury a na základě nich si vygeneruje validní PDDL/SAS$^+$ soubory.



Obrázek 5.2: Struktura vstupních dat PDDL a SAS$^+$

Ačkoli samotný plánovač PADu nebude využívat všech dostupných prostředků jazyka PDDL (ne všechny jsou určeny pro klasické plánování), plná podpora pro načítání PDDL 3.1 byla implementována za účelem možnosti využití i kolegy z příbuzných oblastí výzkumu (zabývajícími se např. oblastí rozvrhování), kteří by měli zájem používat jazyk PDDL a chybí jim v současnosti robustní parser/loader vstupních dat.

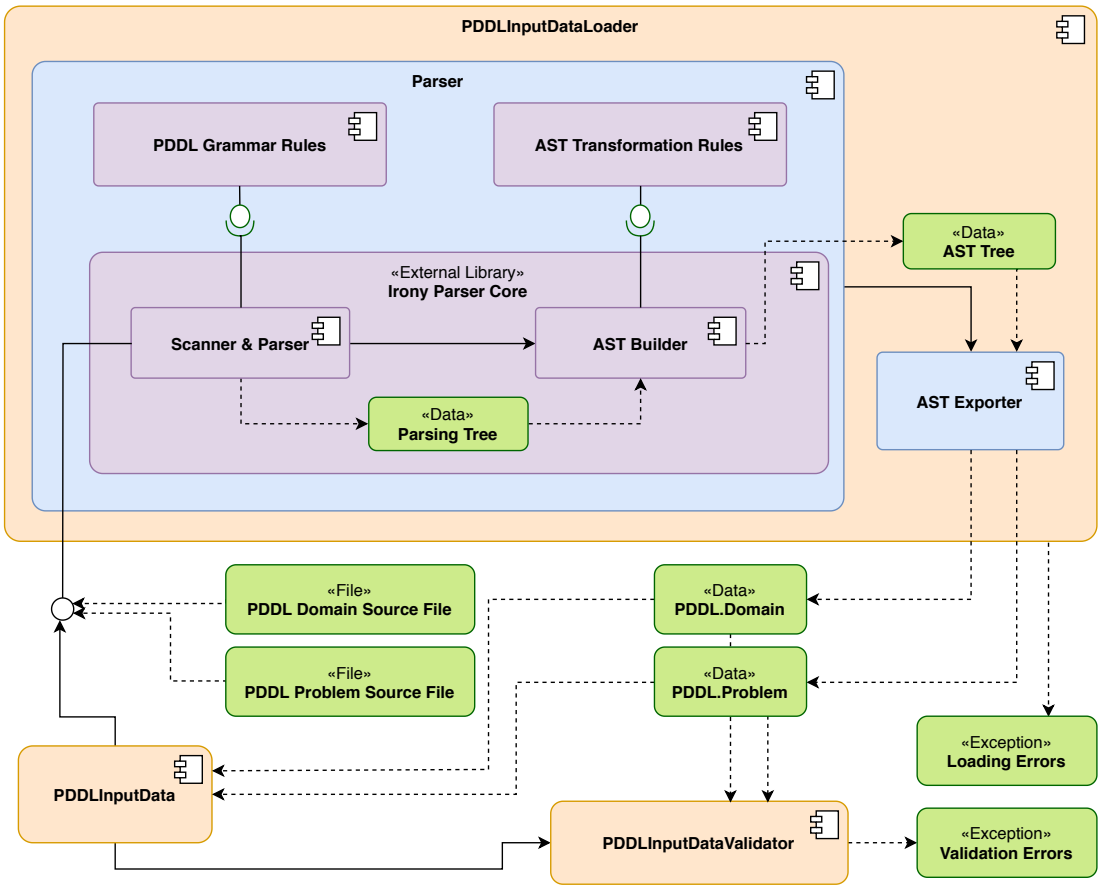
5.4 Načítání a validace PDDL

Následující sekce podrobněji popisuje proces načítání vstupu ze souborů zapsaných v jazyce PDDL. Načíst je potřeba nejprve soubor PDDL domény a následně soubor PDDL problému. Klíčové komponenty a standardní informační workflow je zachyceno v diagramu komponent na obr. 5.3.

Proces načítání zabezpečuje komponenta `PDDLInputDataLoader` (dále též zkráceně *loader*) a je rozdělen do několika fází: nejprve je načten vstupní soubor a *Parser* vytvoří AST (*abstraktní syntaktický strom*)[26], ten je dále zpracován speciálním *AST Exporterem*, který je schopen za pomocí kontextu již vytvářet koncová data. Ta jsou pak předána volajícímu (`PDDLInputData`) a nakonec zvalidována v komponentě `PDDLInputDataValidator`.

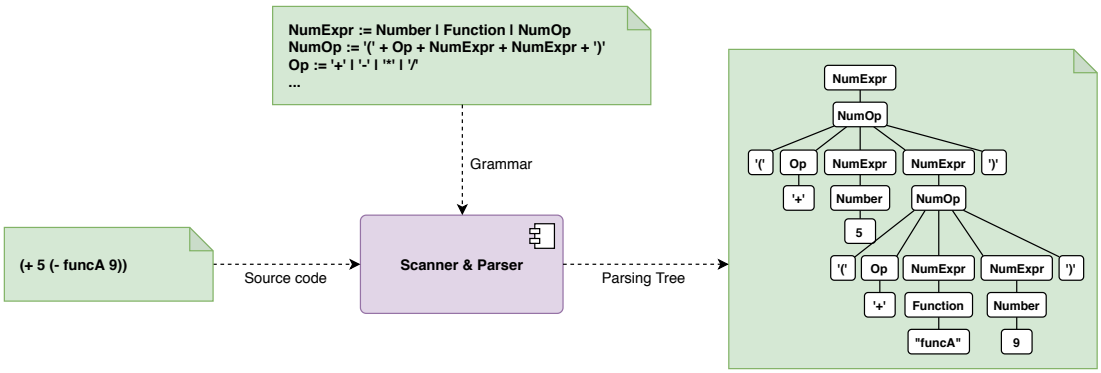
Parser

Vzhledem k tomu, že PDDL je plnohodnotný jazyk s jasně definovanou syntaxí, je rozumné použít k jeho načtení plnohodnotný parser. Ten vytvoříme s pomocí externí knihovny projektu *Irony*[8].



Obrázek 5.3: Architektura načítání a validace PDDL vstupu

Projekt Irony je vývojový toolkit pro implementaci jazyků v prostředí .NET. Na rozdíl od jiných „těžkotonážních“ nástrojů (*yacc*, *lex*) negeneruje Irony na základě vstupní gramatiky (ve formě nějakého meta–jazyka) kód scanneru/parseru, ale dovoluje gramatiku přímo zadávat v C# kódu ve formě přetěžování operátorů a vnitřní moduly scanneru/parseru se pak samy starají o proces parsování[8].



Obrázek 5.4: První fáze načítání – parsing

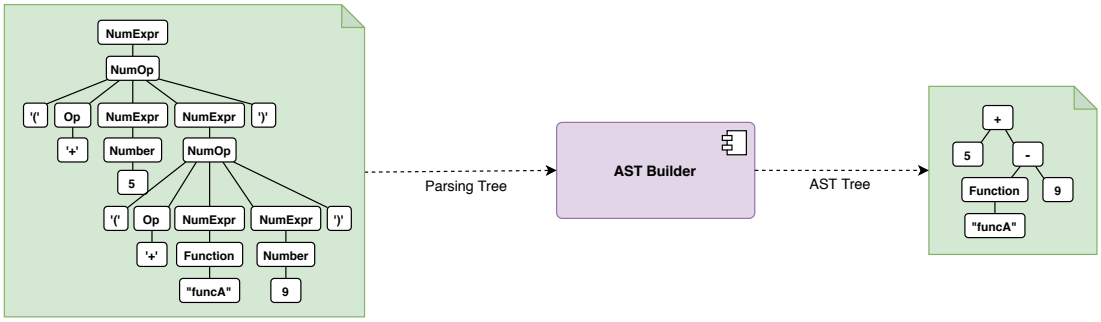
Irony dává k dispozici poměrně jednoduchý mechanismus pro specifikaci gramatiky vstupního jazyka ve formě velmi blízké BNF formátu (*Backus–Naur form*, soubor derivačních pravidel)[27]. Gramatika PDDL je popsána v dokumentu[21] na stránkách IPC právě ve formátu BNF. Takže ji stačí celkem přímočaře pře-

psat do struktur Irony tak, aby to vyhovovalo interně používanému *LALR(1)*[28] parseru.

V první fázi načítání je předán knihovnám Irony vstup v textové formě, ta ho na základě specifikované gramatiky nejprve ve *Scanneru* převede na sekvenci syntaktických tokenů a *Parser* následně vytvoří *parsovací (též derivační) strom*[26] – stromovou datovou strukturu, která dá syntaktickým tokenům strukturální kohezi, viz obr. 5.4.

AST

Parsovací strom je možno dále interně přímo v rámci knihoven Irony převést do AST stromu (komponenta *AST Builder*). AST se od parsovacího stromu liší tím, že již odstiňuje uživatele od vstupní syntaxe – nezajímají nás určité tokeny, oddělovače, závorky apod.



Obrázek 5.5: Druhá fáze načítání – tvorba AST

Zatímco struktura parsovacího stromu je dána odvozovacími pravidly gramatiky, AST se již svou podobou výrazně přibližuje koncovým datům. Struktura dat je přímo dána strukturou AST stromu a původní „košatý“ strom se velmi zjednodušuje. Převod ilustruje diagram na obr. 5.5.

Pravidla pro převod jednotlivých větví parsovacího stromu do AST jsou Irony předána společně s definovanou gramatikou jazyka – viz digram komponent na obr. 5.3.

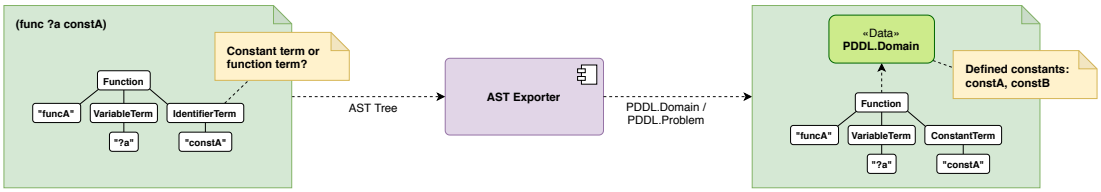
Nicméně, AST je stále jen souborem syntaktických informací, z nichž spoustu nelze interpretovat bez kontextu dříve načtených dat – k sémantické analýze a konverzi do koncových dat tak dochází až o krok později v komponentě *AST Exporteru*.

Transformace AST do koncových dat

Výstupem komponenty parseru je traverzovatelný AST strom. Komponenta *AST Exporter* nyní projde AST strom pomocí návrhového vzoru visitor[12] a postupně plní koncová data, která již mají jasně definovanou stromovou strukturu.

Dříve načtená data se často používají jako sémantický kontext pro načtení dat následujících. Výsledkem jsou podstruktury `PDDL.Domain` a `PDDL.Problem`, které jsou již přímo součástí hlavní struktury vstupních dat `PDDLInputData`.

Obr. 5.6 ilustruje proces převodu AST stromu, kdy není ještě na úrovni AST stromu jasné, zda druhý z argumentů funkce je konstantním termem nebo funkčním termem (voláním vnořené objektové funkce) – gramatika totiž dovoluje obě varianty a je potřeba zjistit, které konstanty a funkce jsou v doméně definovány.



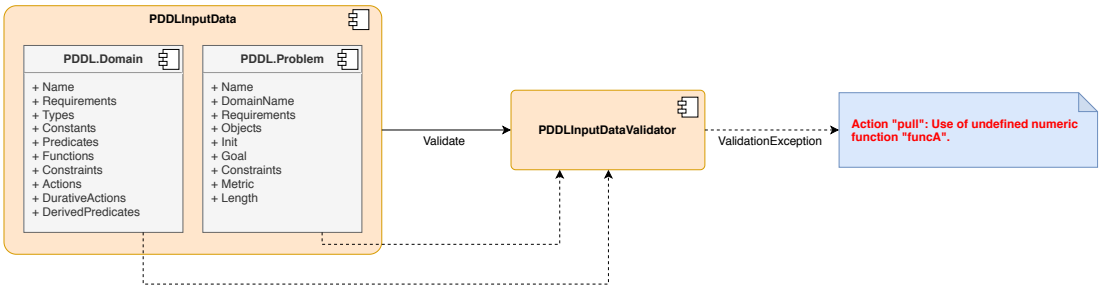
Obrázek 5.6: Třetí fáze načítání – export z AST

Validace dat

Celá komponenta `PDDLInputDataLoader` může vyhazovat výjimky specifického typu `LoadingException` – jde zejména o chyby parseru (neplatné konstrukce ve vstupním souboru, s přesnou lokalizací chyby) a případně též I/O chyby (prázdný či poškozený soubor apod.). Chyby parseru jsou vráceny s popiskem problému ve formátu *Syntax error, expected ':parameters' (file: TC_Actions.pddl, line: 7, column: 13).*

Korektně načtená data parserem jsou tedy v souladu s definovanou gramatikou PDDL, což ovšem ještě neříká nic o tom, zda jsou korektní i sémanticky. *AST Exporter* dělá maximum pro to, aby na základě již načtených informací provedl sémantickou analýzu a exportoval data do koncových struktur. Stále však mohou být data navzájem nekonzistentní a porušovat pravidla jazyka PDDL. Pro tuto kontrolu existuje separovaná komponenta `PDDLInputDataValidatoru` (dále též pouze *validátor*).

Je třeba poznamenat, že velkou část funkcionality validátoru by bylo možné zakomponovat do procesu načítání dat (konkrétně v transformaci dat *AST Exporterem*). K rozdělení na samostatnou komponentu vedlo několik důvodů.



Obrázek 5.7: Závěrečná fáze načítání – validace vstupních dat

V prvé řadě je to obecný princip modularity, kdy se logicky oddělí zodpovědnosti za určité role v procesu načítání a validace – a hlavní úlohou *AST Exporteru* je z AST dostat koncová data.

Validace je již v zásadě nezávislá na původním vstupu ve formátu PDDL. Potenciálně si můžeme chtít koncová data naplnit *manuálně*, služeb loaderu vůbec nevyužívat a pouze si zavolat validaci pro ověření, že máme konzistentní definici dat. Anebo naopak, pokud si jsme jisti korektností vstupních dat, je možno služby validace vůbec nevyužívat a ušetřit nějaký čas.

Validátor je implementován formou visitoru, traverzujícího datové stromy `PDDL.Domain` a `PDDL.Problem`. V případě problému je vyhozena výjimka typu `ValidationException` s popisem chyby.

Co konkrétně validátor kontroluje:

- specifikace funkčních požadavků (*requirementů*) – např. pro používání typů je nutné mít v datech specifikován requirement `:typing`,
- korektně zadaná typová hierarchie (při použití typování),
- použití nedefinovaných typů, konstant, objektů, funkcí a predikátů,
- konflikty v jménech konstant, objektů a nulárních funkcí,
- duplikátní definice predikátů, funkcí a akcí,
- používání proměnných v daném rozsahu platnosti (*scopu*) a konflikty s proměnnými z jiných rozsahů platnosti,
- validní používání cen akcí (*action-costů*) a validní definice metriky,
- při volání predikátů a funkcí souhlasí typy předaných parametrů a případně i typ návratové hodnoty funkce,
- validní použití objektových a numerických fluentů,
- odpovídající doména specifikována v problému skutečně souhlasí s předanou doménou.

Oficiální specifikace jazyka PDDL neexistuje, spíše jde o soupisy pravidel a funkcí publikovaných v mnoha článcích – takže ani o našem validátoru nemůžeme říci, že by byl validací vůči oficiální specifikaci jazyka. Nicméně pokud některý z výše uvedených bodů validace neprojde, je vysoká pravděpodobnost, že data nejsou konzistentní a následný proces plánování selže. Proto je ve většině případů užitečné validaci pro ověření konzistence provádět, aby se případná chyba na vstupu lokalizovala co nejdříve.

Testování

Celý proces načítání dat je pokryt unit-testingem v projektu `PAD.Test`. Testovány jsou jak jednotlivé větve gramatiky (a jejich správná transformace do AST), tak i ucelené příklady z black-box perspektivy ve formě validních PDDL souborů, kde se ověřují jednotlivé funkční bloky.

Nakonec se testují i velké komplexní vstupy, které jsou převzaty přímo z jednotlivých ročníků soutěže IPC, dostupné např. na webových stránkách projektu *Planning.Domains*[29].

Validace je obecně při načítání defaultně zapnutá a je tomu tak i v případě zmíněných komplexních testů. Bohužel, i oficiální příklady ze soutěže IPC v několika málo případech neprocházejí validací – jde zejména o chybějící specifikace requirementů. Tyto případy byly v testovacích případech opraveny a patřičně okomentovány.

5.5 Načítání a validace SAS$^+$

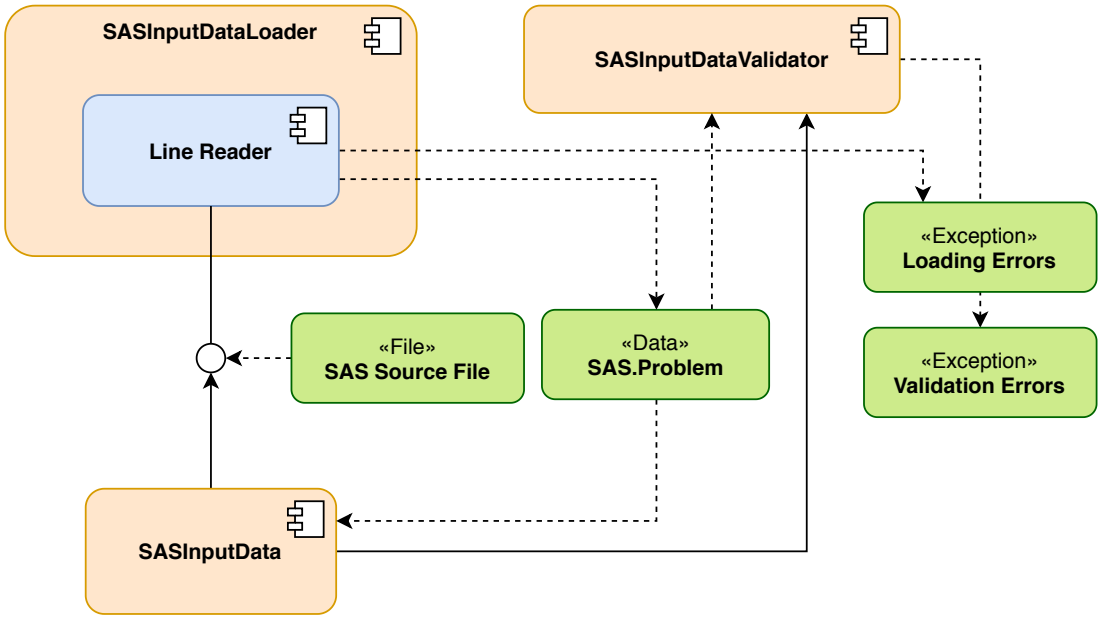
Jak již bylo zmíněno výše, pro formalismus SAS$^+$ neexistuje plnohodnotný vstupní jazyk a jako vstupní formát jsme tak zvolili souborový formát, který užívá projekt *Fast Downward*[24]. Jde o sekvenci bloků, které po řádcích specifikují určité vlastnosti.

Např. blok definující proměnné je ohraničen *begin_variable/end_variable*, na prvním řádku bloku má být definováno jméno proměnné, na druhém index udávající axiom-vrstvu, na třetím velikost oboru hodnot a následně symbolické vyjádření jednotlivých hodnot.

Jde vidět, že každý řádek daného bloku je těsně spjat s nějakou vlastností. Neočekává se podpora pro různá odsazení, komentáře apod. Tedy nejde o vstupní jazyk, ale pouze velmi rigidní vstupní formát.

Vzhledem k tomuto charakteru nedává smysl vytvářet složitý loader a v několika fázích data převádět do koncových dat, jako jsme to dělali v případě PDDL. Zde si vystačíme s řádkovou čtečkou, která si udržuje kontext právě načítaných dat a plní strukturu `SAS.Problem`, která je již součástí koncových dat `SASInputData`.

I zde je k dispozici validační komponenta `SASInputDataValidator`, která ověřuje konzistenci načtených (příp. ručně naplněných) dat. Proces načítání a validace ilustruje diagram na obrázku 5.8.



Obrázek 5.8: Architektura načítání a validace SAS$^+$ vstupu

Řádková čtečka

Vstupní soubor je předán k načtení přímo čtečce `LineReader`. Tato komponenta je pouze odvozenou variantou standardní čtečky proudových dat v C# (třída `System.IO.StreamReader`), která si ze souboru načítá data po řádcích – s dodatečnou schopností *nakouknutí (peak)* na následující řádek, bez nutnosti plného nahrání (*fetchnutí*) tohoto řádku.

Při zpracovávání každého bloku jsme v určitém režimu – nicméně v některých případech může i nemusí být blok přítomen (např. specifikace verze SAS$^+$ dat není v prvních verzích přítomna). Použijeme *peak* a podle obsahu následujícího řádku přepneme režim.

Komponenta *loaderu* může vyhazovat výjimky typu `LoadingException`, pokud čtečka narazí na nečekaná data (např. na řádku je očekávána celočíselná

hodnota a je přítomen textový řetězec, blok není řádně ukončen apod.). Výjimka obsahuje informaci o charakteru chyby a její přesné lokalizaci ve vstupním souboru.

Validace dat

Důvody pro existenci izolované komponenty pro validaci vstupních dat jsou analogické jako v PDDL části – chceme mít izolovanou funkcionalitu pro kontrolu konzistence dat, chceme ji mít možnost použít potenciálně zcela bez načítače dat a chceme mít i možnost validaci jednoduše potlačit.

Ověřovány jsou sémantické vztahy mezi jednotlivými částmi načtených dat. V případě chyby validátor vyhodí výjimku typu `ValidationException` s popisem problému.

Validator ověřuje ve struktuře `SAS.Problem` následující problémy:

- nevalidní verze SAS$^+$ dat,
- duplikátní jména proměnných,
- proměnná má neplatný axiom-index,
- proměnná má prázdný obor hodnot (nemůže tedy nabýt žádné hodnoty),
- hodnoty proměnné mají duplikátní jména,
- mutexová skupina obsahuje duplikované fakty (a nelze ji tedy splnit),
- počáteční stav nemá inicializované hodnoty všech proměnných,
- hodnoty počátečního stavu nejsou v souladu s definicí mutexových skupin,
- neplatná cena operátoru,
- efekt operátoru má modifikovat axiomatickou proměnnou,
- axiom má modifikovat neaxiomatickou proměnnou,
- při přiřazení hodnoty je použita nedefinovaná proměnná,
- proměnné je přiřazována hodnota mimo její obor hodnot.

Testování

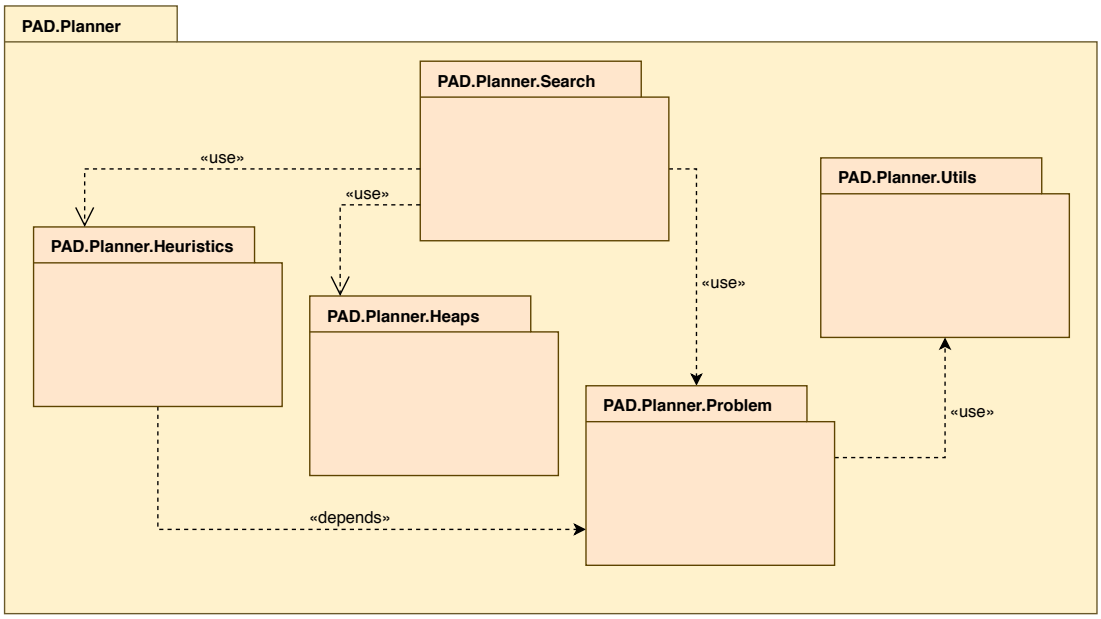
I načítání a validace formalismu SAS$^+$ je pokryto unit-testy. Ověřována je schopnost korektně načíst jednotlivé funkční bloky a následně i komplexní příklady. Unit testy jsou opět součástí speciálního projektu `PAD.Tests`.

6. Plánovač PADu

V následující kapitole popíšeme nejdůležitější část frameworku PAD, a sice plánovač (*planner*). Tato komponenta zprostředkovává základní techniky pro prohledávání stavového prostoru plánovacího problému (zadaného v PDDL či SAS$^+$). Umí nalézt řešení, a to za pomocí různých heuristických metod a podpůrných datových struktur.

Podíváme se na jednotlivé komponenty projektu `PAD.Planner`, popíšeme prohledávací metody a detailně zanalyzujeme specifické implementační části plánovacích problémů zadaných ve formalismech PDDL a SAS$^+$.

6.1 Struktura projektu `PAD.Planner`



Obrázek 6.1: Jednotlivé součásti projektu `PAD.Planner`

Diagram balíčků na obr. 6.1 zachycuje hlavní součásti projektu plánovače, které jsou v kódu totožně odděleny i na úrovni jmenných prostorů (*namespaces*). V dalším textu budou jednotlivé komponenty detailněji rozebrány, nyní uveďme pouze jejich krátký popis a závislosti mezi nimi:

- `PAD.Planner.Search` – část projektu poskytující nástroje pro prohledávání stavového prostoru a hledání řešení optimalizačních problémů; skrze jednotné rozhraní se komunikuje s plánovacím problémem (bez ohledu na použitou vstupní reprezentaci), používají se heuristické funkce a haldy (*heaps*), coby datové struktury uchovávající uzly prohledávaného prostoru,
- `PAD.Planner.Heuristics` – část projektu poskytující sadu heuristických funkcí pro efektivní prohledávání stavového prostoru; vyhodnocení mnoha heuristik závisí na použité reprezentaci problému, jakkoli je v programu snaha o maximální míru abstrakce použité reprezentace,

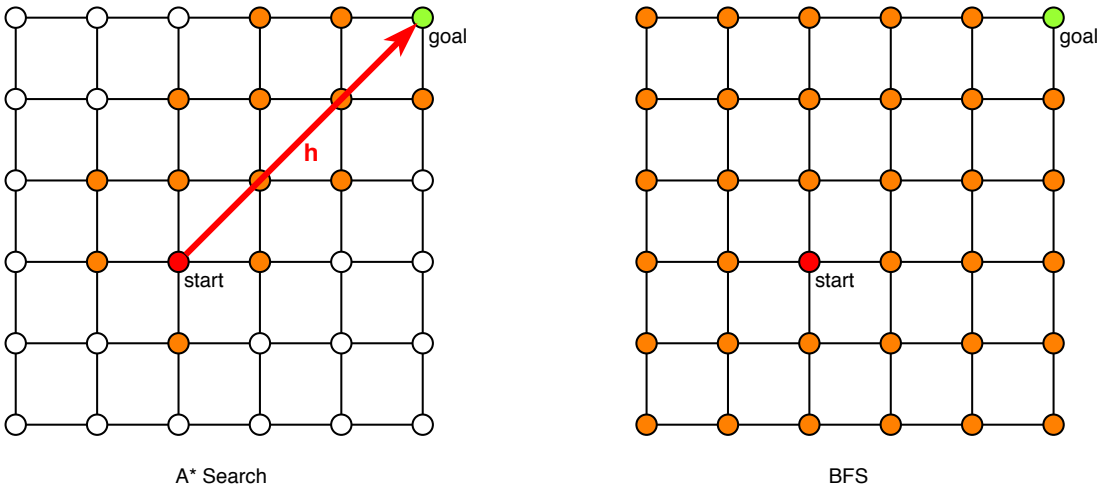
- `PAD.Planner.Heaps` – část projektu poskytující sadu datových struktur vhodných pro prohledávání stavového prostoru; haldy coby částečně uspořádané stromové struktury optimalizované pro získávání minimální/maximální hodnoty jsou pro tento účel ideální,
- `PAD.Planner.Problem` – část projektu zapouzdřující plánovací problém; jednotlivé komponenty mohou být specifické pro PDDL/SAS$^+$ reprezentaci, nicméně z vnějšího pohledu by se pro prohledávání měl plánovací problém vždy tvářit jako abstraktní entita,
- `PAD.Planner.Utils` – část projektu obsahující pomocné funkce, jako je výpočet hashování pro kolekce, výpočet efektivní randomizace, nebo komparátory kolekcí.

6.2 Heuristické prohledávání

V kapitole 3.1 jsme rámcově popsali, jak v prostoru stavů hledat řešení plánovacího problému. Zmíněno nicméně bylo i to, že větvící faktor bývá vždy velký a pro efektivnější prohledávání je potřeba prohledávací algoritmus nějakým způsobem navádět k cíli. Takové navádění zabezpečují tzv. *heuristické funkce*. Ukážeme jejich použití v rámci algoritmu *A* Search*[30].

6.2.1 A* Search

Algoritmus A* Search je zavedenou technikou pro procházení grafů a hledání cesty mezi dvěma vrcholy v grafu. Podobá se klasickému prohledávání do šířky (BFS), ale při procházení grafu prioritizuje určité uzly, na základě zvolené heuristiky.



Obrázek 6.2: Srovnání postupu algoritmů A* a BFS

Rozdíl postupu algoritmů A* a BFS ilustruje obrázek 6.2. Na jednoduchém grafu ve formě mřížky hledáme nejkratší cestu z červeného startovního vrcholu do cílového zeleného vrcholu. Zatímco algoritmus BFS za pomocí fronty prohledává prostor po vrstvách a k nalezení řešení potřebuje projít doslova celý graf, algoritmus A* postupuje při výběru následujících vrcholů chytřeji a prioritizuje ty,

které se blíží cíli. Heuristickou funkcí zde je Eukleidovská vzdálenost: při výběru následujícího vrcholu se vždy spočítá Eukleidovská vzdálenost k cílovému vrcholu a vybere se ten, který je cíli nejblíže.

Stejný princip použijeme při prohledávání prostoru stavů v rámci plánovacího problému. Heuristickou funkcí pro stav může být např. počet splněných cílových podmínek apod. Konkrétní heuristiky detailněji popíšeme vzápětí, nyní však zformulujme algoritmus A* v kontextu prohledávání stavového prostoru.

```
AStarSearch(problem, h):
  init ← počáteční uzel z problem
  nodes ← {(init, h(init))}
  previous[init] ← null
  closed ← ∅
  g[∗] ← ∞
  g[init] ← 0
  while nodes ≠ ∅ do
    node ← vyndej uzel z nodes s minimální hodnotou klíče
    if node ∈ closed then
      pokračuj dalším cyklem
    end if
    if node je cílovým uzlem problému problem then
      return řešení sestavené pomocí předchůdců v previous
    end if
    successors ← všechny možní následníci z aktuálního node
    for successor ∈ successors do
      g′ ← g[node] + cost(node, successor)
      if g′ ≤ g[successor] then
        previous[successor] ← node
        g[successor] ← g′
        if successor ∉ closed then
          nodes ← nodes ∪ {(successor, g[successor] + h(successor))}
        end if
      end if
    end for
    closed ← closed ∪ {node}
  end while
  return řešení neexistuje
```

Algoritmus A* v našem plánovači tedy bude fungovat tak, že začne v počátečním uzlu problému, pro tento uzel najde jeho možné přechody (následníky) a ty uloží mezi otevřené uzly. Funkce *cost* udává cenu daného přechodu. Počítaná hodnota $g(node)$ značí skutečnou cenu dosavadní cesty z počátečního uzlu do uzlu *node*, hodnota $h(node)$ je heuristický odhad vzdálenosti uzlu *node* do cíle. Nové otevřené uzly se pak ukládají s klíčem $g(node) + h(node)$. Opakovaně se pro zpracování vybírá otevřený uzel s minimální hodnotou tohoto klíče a prochází se stavový prostor (ideálně ve směru k cílovým podmínkám). Zpracované uzly se označují jako uzavřené a znovu se již neotevírají.

Prohledávání může skončit buď nalezením uzlu, který splňuje cílové podmínky,

nebo neúspěšným projitím všech dosažitelných uzlů daného problémů, kdy algoritmus zahlásí, že řešení problému neexistuje. Při nalezení cílového uzlu se snadno zkonstruuje celkové řešení zpětným procházením mapy předchůdců *previous*.

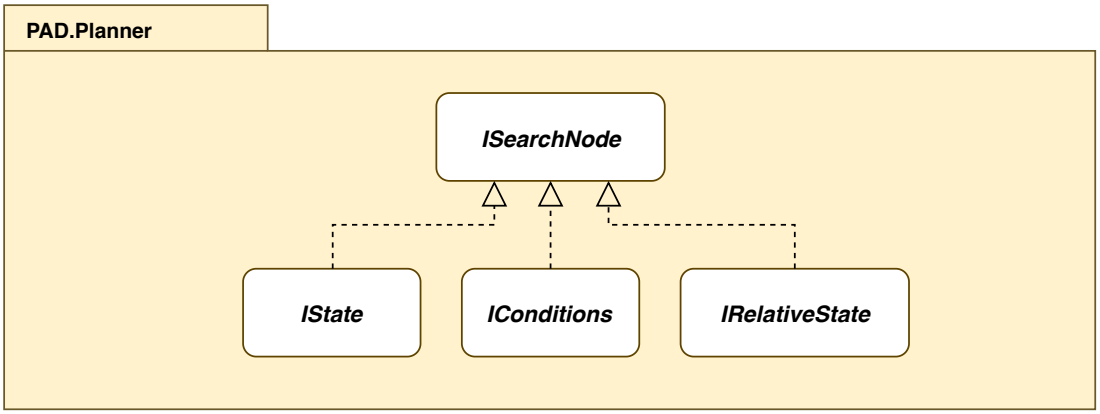
Algoritmus má na vstupu pouze plánovací problém a zvolenou heuristiku a takto popsaný přímo diktuje požadované funkce pro rozhraní plánovacího problému. Ten musí umět vrátit počáteční uzel problému, ověřovat splnění cílových podmínek a vracet možné přechody ze zvoleného uzlu.

Zobecněné prohledávací entity

V kapitole 3.1 jsme zmiňovali dva základní druhy prohledávání, a sice dopředné hledání (*Forward-Search*) nad stavy a zpětné hledání (*Backward-Search*) nad zobecněnými podmínkami.

Náš algoritmus A* však toto nijak nezohledňuje, což je způsobeno tím, že nepracuje s konkrétními stavy, ani podmínkami, ale se zobecněnými entitami, které zde nazýváme *prohledávací uzly*. Z pohledu A* je řešen abstraktní grafový problém nalezení cesty z počátečního do jednoho z cílových vrcholů a algoritmus v zásadě neví, zda zrovna probíhá dopředné, či zpětné prohledávání.

Technicky jsou tyto prohledávací uzly společným rozhraním pro stavy i podmínky a v programu se nazývají `ISearchNode`. Jak ilustruje obr. 6.3, ve frameworku existuje vedle stavů a podmínek ještě třetí, dosud nezmíněná, entita *relativní stav*, kombinující vlastnosti stavů a podmínek a zprostředkovávající rovněž zpětné prohledávání. K jejímu bližšímu popisu se dostaneme později.

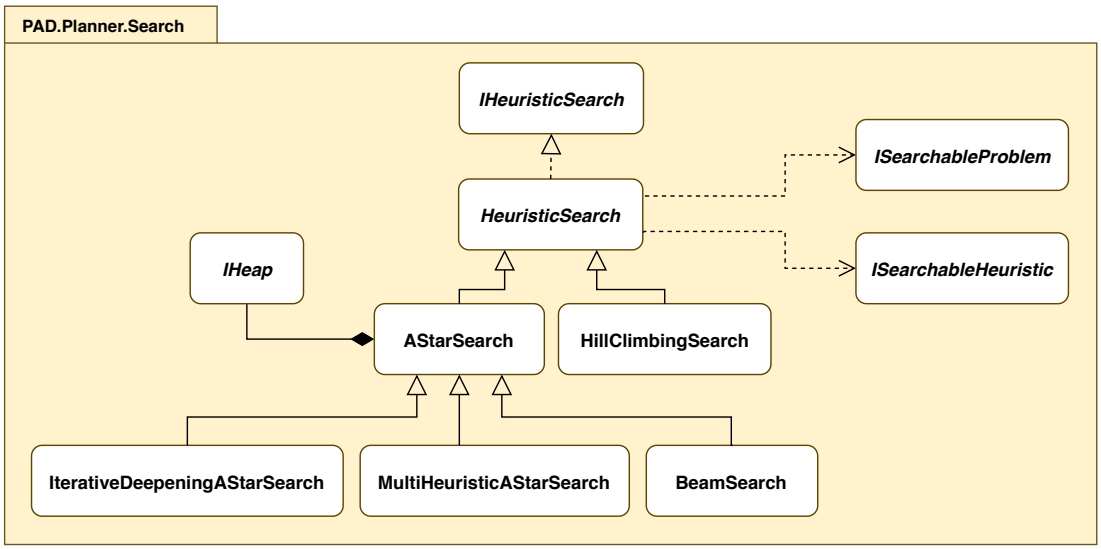


Obrázek 6.3: Prohledávací entita `ISearchNode`

Ve skutečnosti tedy počáteční uzel prohledávání je v případě dopředného hledání opravdu počátečním stavem problému, ale při zpětném prohledávání jde o cílové podmínky, podle kontextu se jako přechody mezi prohledávacími uzly používají dopředné či zpětné aplikace akcí apod.

6.2.2 Další prohledávací algoritmy

Kromě algoritmu A* Search framework obsahuje další prohledávací algoritmy, typicky používané pro specifické účely. Jak ilustruje diagram struktury těchto algoritmů v rámci frameworku PAD na obr. 6.4, spousta z nich vychází ze standardní implementace A* Search. Bez toho, abychom zacházeli do přílišných detailů, si nyní tyto komponenty krátce popišme.



Obrázek 6.4: Struktura prohledávacích algoritmů frameworku

- `IHeuristicSearch` – společné rozhraní pro všechny heuristické prohledávací procedury ve frameworku PAD; určuje elementární požadované metody pro práci s prohledávací procedurou a vracení požadovaných výsledků či statistik hledání,
- `HeuristicSearch` – bázová třída pro všechny heuristické prohledávací procedury; obsahuje společné části a nastavení prohledávacích procedur; její funkce závisí na abstraktním plánovacím problému a zvolené abstraktní heuristice,
- `AStarSearch` – základní implementace A* Search prohledávání detailně popsaného v této kapitole výše; používá datovou strukturu haldy pro udržování seznamu otevřených uzlů[30],
- `IterativeDeepeningAStarSearch` – implementace varianty algoritmu iterativního zanořování využívající myšlenky použití heuristik jako to dělá A* (název algoritmu zkracován jako IDA*); v zásadě se jedná o prohledávání do hloubky (DFS), s omezením hloubky pomocí výpočtu heuristické funkce[31],
- `MultiHeuristicAStarSearch` – varianta A* prohledávání využívající více heuristik najednou; algoritmus si drží více seznamů otevřených uzlů pro jednotlivé heuristiky a iterativně je vyhodnocuje, zatímco seznam uzavřených uzlů je v rámci algoritmu sdílený[32],
- `BeamSearch` – prohledávací algoritmus, který na rozdíl od A* mezi nové otevřené uzly přidává pouze několik vybraných uzlů s nejlepší hodnotou heuristiky; obecně není kompletní[33],
- `HillClimbingSearch` – jednoduchý prohledávací algoritmus (též zvaný jako *gradientní algoritmus*), který z aktuálního uzlu vybere pro další postup vždy pouze jeden uzel s nejlepším odhadem; tento přístup je rychlý a paměťově nenáročný, ale není obecně kompletní, protože může uváznout v nějakém lokálním maximu, aniž by nalezl maximum globální[30].

Bázová třída `HeuristicSearch` obsahuje podporu pro logování, tj. sledování průběhu výpočtu a zaznamenávání statistik výpočtu, pro snazší analýzu efektiv-

nosti aktuálně probíhajícího prohledávání. Díky tomu se i lépe analyzuje dopad prohledávacích metod a heuristik, např. při konstrukci nové heuristiky nebo nového prohledávacího algoritmu.

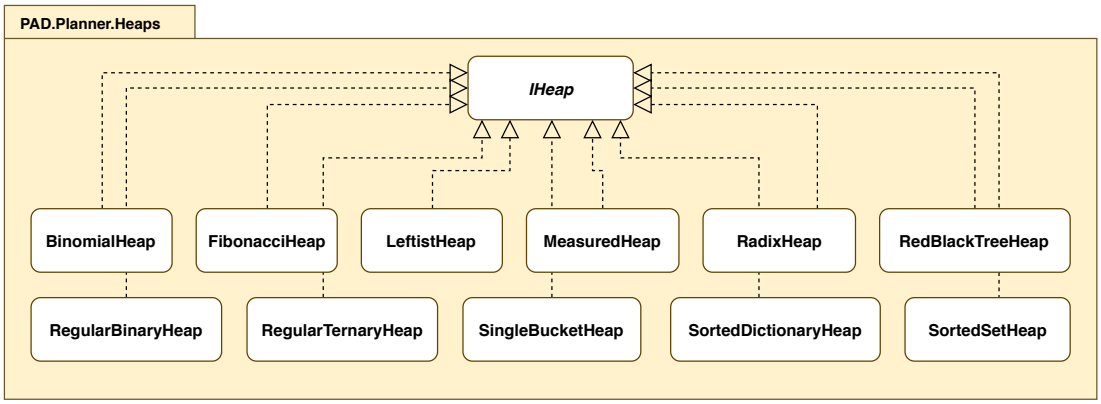
K dispozici je i možnost nastavit časový a paměťový limit výpočtu. Při překročení takového limitu výpočet skončí s příslušným statusem.

6.2.3 Haldy

Velmi důležitým momentem algoritmu A* Search je uchovávání otevřených uzlů. V popisu algoritmu jsme viděli, že při zpracování nějakého uzlu se najdou jeho následníci, pro ty se vypočítá ohodnocující funkce $f = g + h$, neboli součet skutečné ceny cesty z počátečního vrcholu a heuristicky odhadnuté vzdálenosti do nějakého cílového uzlu, a toto ohodnocení se použije coby klíč při přidání tohoto nového uzlu do kolekce otevřených uzlů.

Při následné iteraci se pak vždy z této kolekce vybere ten uzel, který má ohodnocení f minimální. Potřebujeme tedy kolekci, která bude umět efektivně vracet prvek s minimální hodnotou klíče a zároveň efektivně zpracovávat nově přidané prvky. Takovou kolekcí je přirozeně *halda*, částečně uspořádaná stromová struktura přesně splňující tuto vlastnost[5].

Diagram na obr. 6.5 ilustruje sadu dostupných implementací hald ve frameworku PAD, jejich efektivita se může lišit dle typu řešeného problému. Nechybí implementace nejznámější typů hald jako je *binomiální halda*, *Fibonacciho halda*, *leftist halda*, nebo klasické *regulární haldy*[5]. K dispozici jsou i další experimentální či ladící haldy. Např. efektivní halda nazvaná *RedBlackTreeHeap*, která používá kolekci z externího balíčku *Wintellect Power Collections*[9].



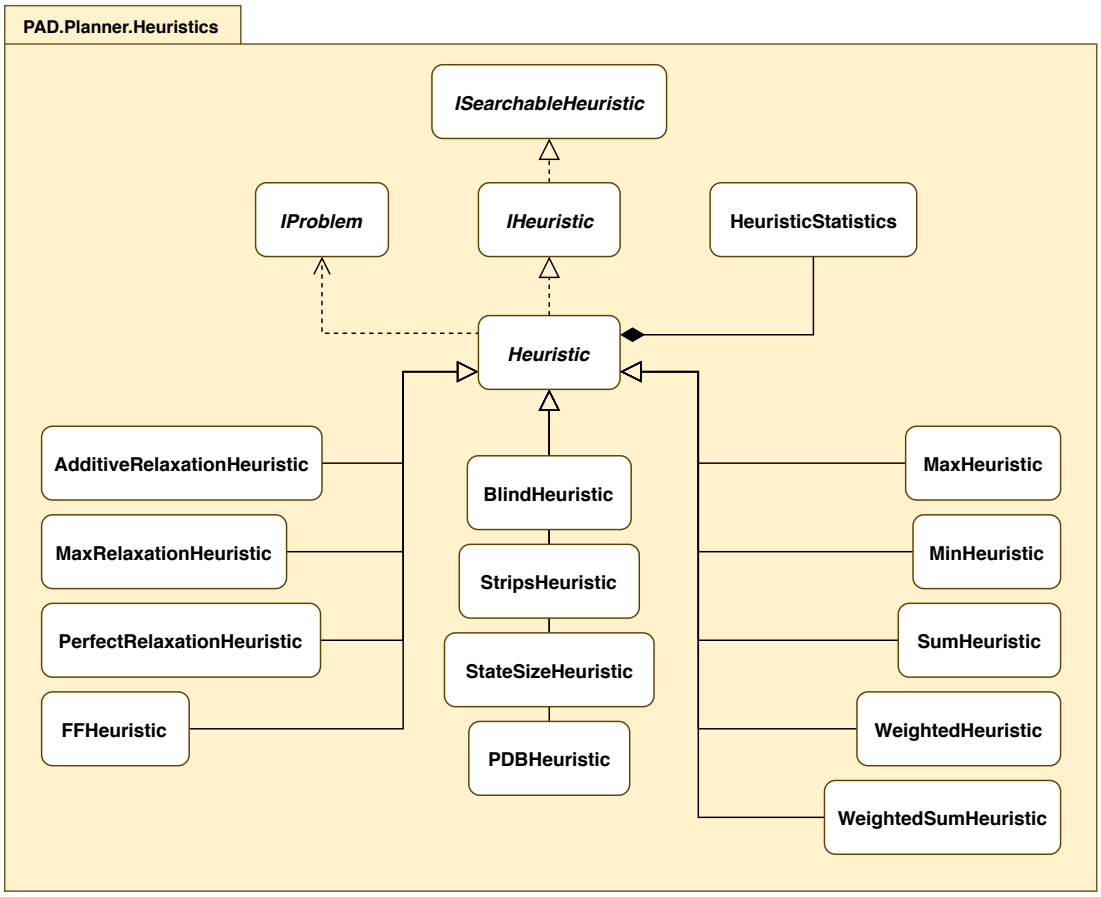
Obrázek 6.5: Sada dostupných hald ve frameworku PAD

6.2.4 Heuristiky

Framework PAD obsahuje sadu implementovaných heuristik pro prohledání prostoru stavů plánovacího problému. Diagram na obr. 6.6 zachycuje strukturu těchto heuristik včetně závislostí.

Vyhodnocení mnoha heuristik přímo závisí na implementačních detailech plánovacího problému (včetně toho, zda je vstup v PDDL nebo SAS$^+$, proto v následujícím výčtu pouze stručně popíšeme princip a význam jednotlivých heuristik

a k mnoha z nim se vrátíme později v části popisující implementační detaily plánovacího problému.



Obrázek 6.6: Sada dostupných heuristik ve frameworku PAD

- `ISearchableHeuristic` – nejvyšší rozhraní pro heuristiky v PADu, požadující vyčíslení heuristické hodnoty pro zobecněné vyhledávací uzly; pouze s tímto maximálně abstraktním interfacem by měly pracovat prohledávací algoritmy pro dosažení nejmenší možné závislosti na zvolené reprezentaci,
- `IHeuristic` – společné rozhraní pro všechny PDDL a SAS$^+$ heuristiky ve frameworku PAD; požadovanými funkcemi jsou především vyčíslení heuristické hodnoty pro libovolný stav, podmínku či relativní stav v plánovacím problému,
- `Heuristic` – bázová třída pro všechny dostupné heuristiky, může zaznamenávat statistiky heuristických volání pro analýzu průběhu prohledávání,
- `HeuristicStatistics` – statistiky volání heuristik; zaznamenává se údaj o nejlepší a průměrné heuristické hodnotě a počtu heuristických volání,
- `BlindHeuristic` – slepá heuristika, tj. vrací vždy heuristickou hodnotu nula; v takovém případě se např. A* algoritmus redukuje na *Dijkstrův algoritmus*,
- `StripsHeuristic` – jedna z prvních zkonstruovaných heuristik, která udává odhad vzdálenosti k cíli pomocí počtu nesplněných cílových podmínek[34]; bude dále rozebrána v části 6.4.6,

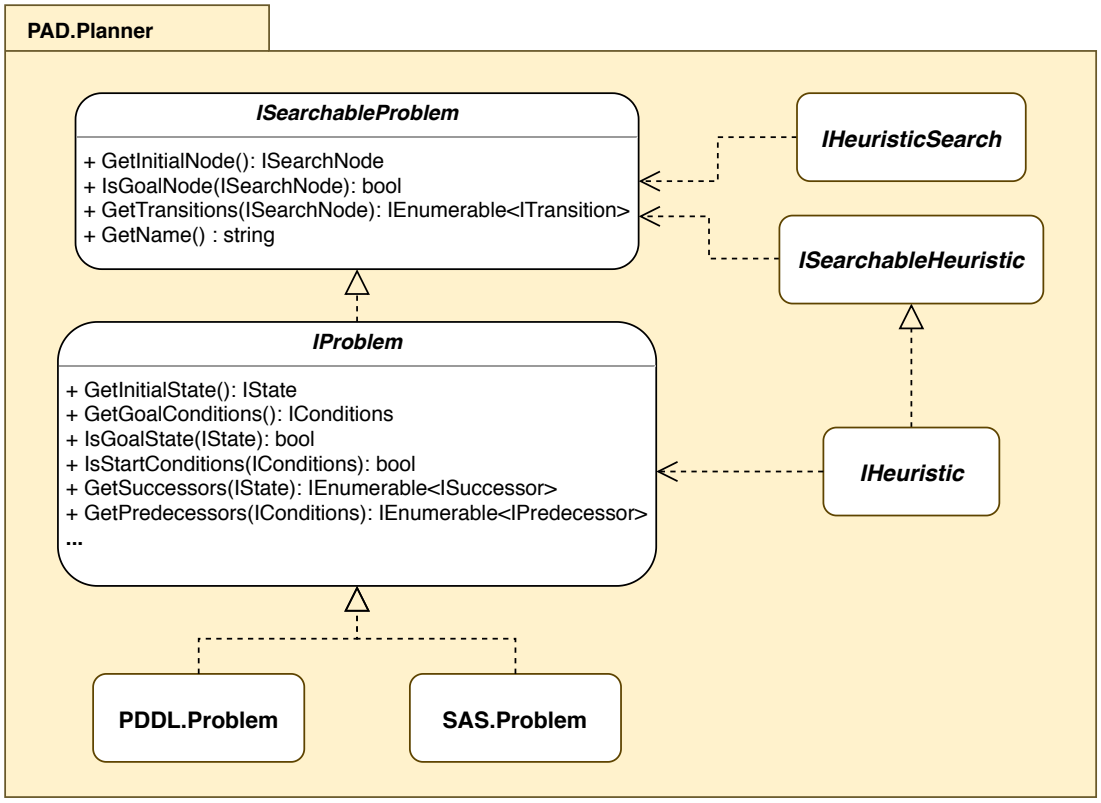
- `StateSizeHeuristic` – pomocná heuristika fungující na základě velikosti velikosti uzlu (např. počtu predikátů),
- `PDBHeuristic` – heuristika využívající principu tzv. *abstrakce*; heuristické hodnoty jsou dopředu spočítány pro různé *vzory* a uloženy do *databáze vzorů*[35][36]; bude blíže rozebráno v části 6.5.7,
- `AdditiveRelaxationHeuristic` – heuristika využívající principu tzv. *relaxace*; z původního problému se vytvoří odpovídající *relaxovaný plánovací graf*, který je následně analyzován, jednotlivé pod-odhady ceny jsou sčítány[37][38],
- `MaxRelaxationHeuristic` – varianta předchozí heuristiky, kdy se ve zmíněných pod-odhadech bere maximální odhad ceny operátorů[37]; obě heuristiky budou dále rozebrány v části části 6.4.6,
- `PerfectRelaxationHeuristic` – další varianta relaxačních heuristik; původní problém je převeden do odpovídajícího relaxovaného problému a je nalezeno jeho optimální řešení – cena tohoto řešení je pak vrácenou heuristickou hodnotou[37],
- `FFHeuristic` – důmyslnější varianta relaxační heuristiky, konstruující relaxovaný plánovací graf a následně odzadu analyzující ceny relevantních *podpůrných* akcí[39]; více v části 6.4.6,
- `MaxHeuristic` – složená heuristika, kde se heuristická hodnota spočítá pro seznam heuristik a vrátí se maximum z těchto heuristických hodnot,
- `MinHeuristic` – složená heuristika, kde se heuristická hodnota spočítá pro seznam heuristik a vrátí se minimum z těchto heuristických hodnot,
- `SumHeuristic` – složená heuristika, kde se heuristická hodnota spočítá pro seznam heuristik a vrátí se součet těchto heuristických hodnot,
- `WeightedHeuristic` – složená heuristika, která heuristickou hodnotu rodičovské heuristiky násobí specifikovanou váhou,
- `WeightedSumHeuristic` – složená heuristika operující s váženým součtem heuristických hodnot pro seznam použitých heuristik.

6.3 Rozhraní plánovacího problému

V této a následujících dvou podkapitolách popíšeme komponentu plánovacího problému (dále též pouze *problém*) jak ze strukturálního pohledu, tak z funkčního hlediska. Předchozí část o heuristickém prohledávání naznačila závislosti ostatních komponent na plánovacím problému a požadovanou funkcionalitu této komponenty.

Heuristické prohledávání pracuje s maximální možnou abstrakcí plánovacího problému, tj. rozhraním `ISearchableProblem`. Tento interface požaduje po implementaci pouze vracení počátečního bodu vyhledávání, ověřování cílového bodu a generování přechodů. Díky tomu vzniká pouze minimální nutná závislost na této komponentě, a je tím významně podpořena rozšiřitelnost. Jak snadno lze vytvořit novou reprezentaci problému je ilustrováno v kapitole 8.3.

Interface `IProblem` je pak hlavní entitou pro práci s problémem v kontextu klasického plánování a diktuje již více specifickou funkcionalitu, blízkou reprezentacím PDDL a SAS$^+$. S tímto rozhraním pak pracují i primární heuristiky frameworku. Celou koncepci rozhraní plánovacích problémů ve frameworku PAD ilustruje obr. 6.7.



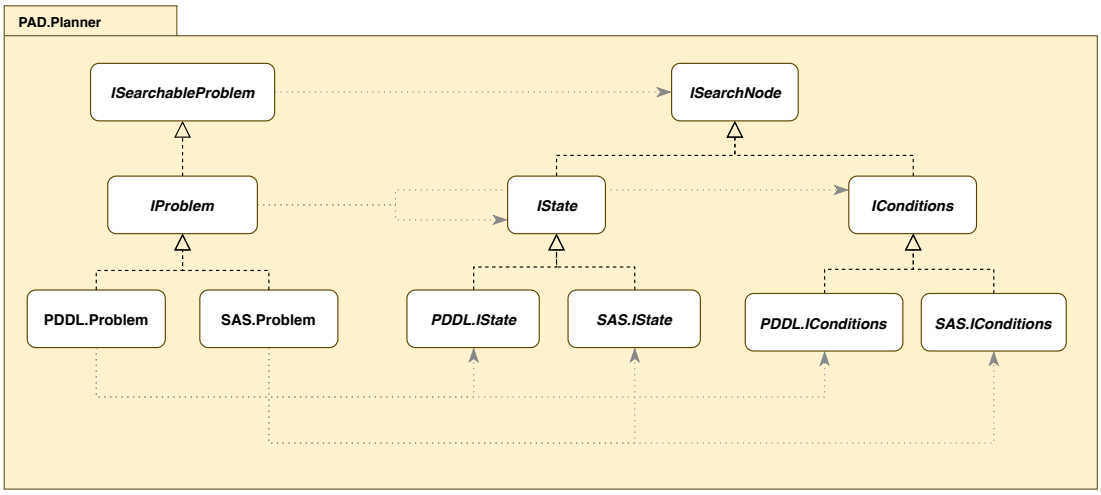
Obrázek 6.7: Rozhraní plánovacího problému

Reálné rozhraní `IProblem` je o něco rozsáhlejší, mimo zmíněných metod obsahuje další podpůrnou funkcionalitu, např. vracení jména domény/problému, cesty ke vstupním souborům, možnosti modifikace vstupních a cílových podmínek, metody pro heuristické výpočty, vracení relaxovaného problému apod. Dále je zde i několik metod vyžadovaných od zadavatele pro budoucí použití, mezi které patří např. vracení náhodného následníka/předchůdce a jiné.

Diagram 6.7 ukazuje navíc také entity obecných přechodů `ITransition` a specifičtějších přechodů `ISuccessor` a `IPredecessor`. Jde o zapouzdřené struktury obsahující typicky referenci na původní entitu a použitý operátor. K vlastní aplikaci operátoru na referencovanou entitu dojde až v momentě použití. Jazyk C# navíc umožňuje vracet kolekce formou enumerátoru, což nám dovoluje generovat i nové přechody *líně*, tj. až v momentě, kdy s nimi program chce reálně pracovat.

Třídy `PDDL.Problem` a `SAS.Problem` jsou již konkrétními implementacemi závislými na vstupních reprezentacích PDDL a SAS$^+$. Společně s ostatními PDDL a SAS$^+$ specifickými komponentami jsou umístěné ve vlastních jmenných prostorech `PAD.Planner.PDDL` a `PAD.Planner.SAS`.

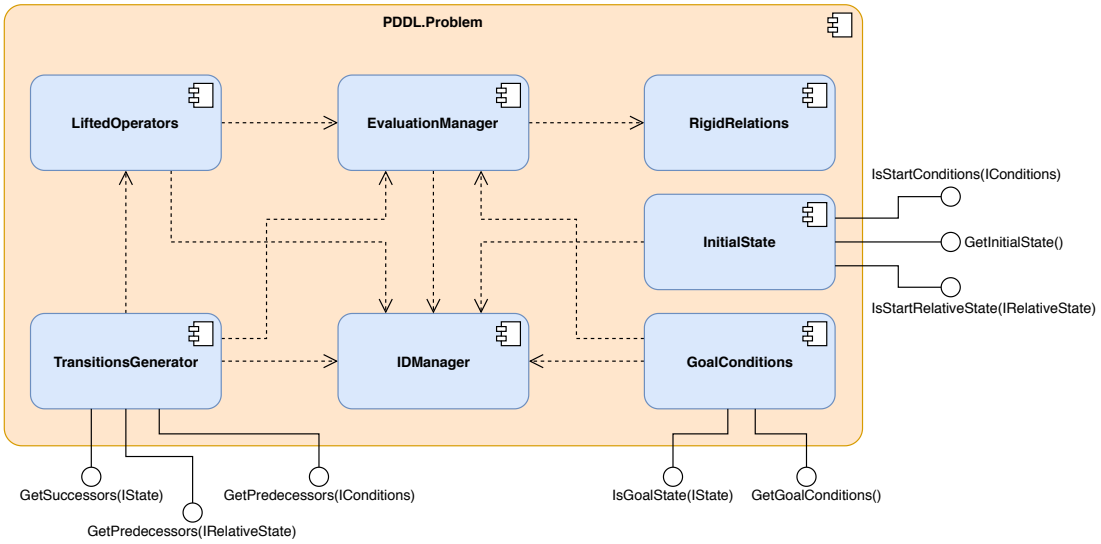
Obrázek na diagramu 6.8 ilustruje hierarchii plánovacích problémů a jejich odpovídajících základních entit.



Obrázek 6.8: Hierarchie plánovacích problémů a základních entit

6.4 Plánovací problém PDDL

V následující podkapitole popíšeme strukturu a fungování plánovacího problému v reprezentaci PDDL. Diagram na obr. 6.9 popisuje klíčové vnitřní komponenty `PDDL.Problem`, spolu s naznačenými závislostmi a poskytovaným rozhraním. Mnoho doplňkových služeb problému je zanedbáno, podobně jako v případě diagramu na obr. 6.7 výše.



Obrázek 6.9: Komponenty PDDL plánovacího problému

Stručně si nyní jednotlivé komponenty popišme, v dalším textu se k nim budeme dále vracet:

- `LiftedOperators` – komponenta spravující kolekci liftovaných operátorů daného problému; podrobněji o operátorech v sekci 6.4.4 níže,
- `EvaluationManager` – část zajišťující vyhodnocování vztahů v rámci plánovacího problému, např. evaluace logických podmínek; více v sekci 6.4.2,

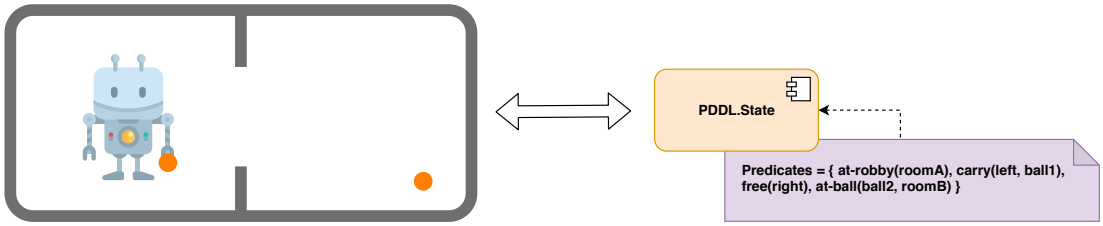
- `RigidRelations` – komponenta spravující rigidní relace v rámci problému, tj. vztahy, které se v průběhu prohledávání stavového prostoru nemění,
- `TransitionsGenerator` – komponenta zajišťující generování nových přechodů v rámci plánovacího problému, tj. následníky pro stavy či předchůdce pro podmínky a relativní stavy; více o generování v části 6.4.5,
- `IDManager` – centrální autorita pro spravování identifikátorů v rámci jednoho PDDL problému, udržuje informace o ID jednotlivých predikátů, konstant, typů, funkcí či proměnných; dává též k dispozici funkcionalitu pro zpětný převod jednotlivých entit do původní textové formy,
- `InitialState` – počáteční stav plánovacího problému; po načtení je zanalyzován a definované vztahy, které nemají šanci se měnit, jsou identifikovány jako rigidní a jsou následně zaznamenány v `RigidRelations` (a z daného stavu jsou odebrány); počáteční stav je možné manuálně měnit, ale ve většině případů zůstane stejný po celou dobu životnosti plánovacího problému,
- `GoalConditions` – cílové podmínky plánovacího problému; opět je možné je v případě potřeby změnit, ale v typickém případě se tak dít nebude a cílové podmínky zůstávají po celou dobu stejné.

6.4.1 Základní entity problému

Uvedeme základní entity používané v rámci PDDL problému používané na prohledávání a podíváme se detailněji, jaké mezi sebou mají vztahy.

Stavy

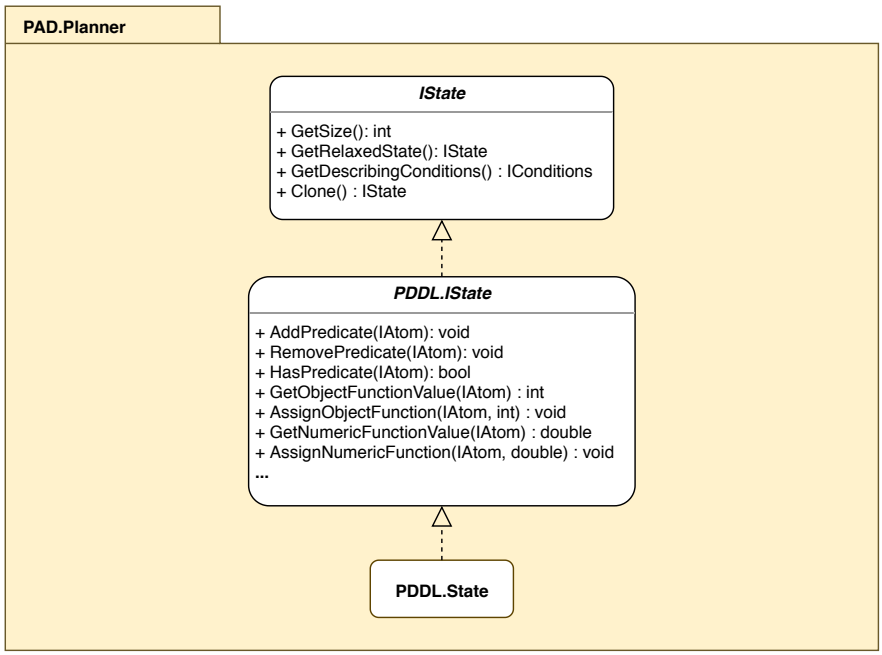
Primární entitou problému je stav (*state*). Udává nám přesné rozpoložení systému v nějakém bodě prohledávání. Stavy v PDDL jsou analogií stavů popsaných v klasické reprezentaci (kapitola 2.1), nicméně umožňují i držet informace o fluentech, tj. hodnotách numerických a objektových funkcí v daném PDDL problému.



Obrázek 6.10: Jednoduchý PDDL stav v doméně Gripper

Vzhledem k tomu, že stav je entita, která se při prohledávání bude používat velmi často, bude docházet k jejímu kopírování, editaci, vyhodnocování vůči podmínkám a uchovávání v nějaké kolekci, je efektivní implementace stavu kritická pro výslednou rychlost prohledávání i paměťovou stopu (v paměti můžeme mít klidně stovky tisíc stavů).

Všechny části programu v kontextu PDDL pracují s rozhraním `PDDL.IState` a případné vytvoření nové implementace pro PDDL stav je tak velmi jednoduché. Diagram na obr. 6.11 ilustruje požadované funkce, které jsou součástí rozhraní.



Obrázek 6.11: Hierarchie PDDL stavů

Obecné rozhraní `PAD.Planner.IState` vyžaduje po implementacích metody pro vrácení velikosti stavu (v kontextu PDDL počet predikátů), převod na relaxovanou verzi stavu (viz podkapitola 6.4.6), klonování stavu a převod stavu na odpovídající podmínku.

Rozhraní `PAD.Planner.PDDL.IState` pak přidává požadavky na manipulaci s obsahem stavu v kontextu PDDL, tj. přidání/odebrání predikátů, test na přítomnost predikátu a funkce pro přístup k případným hodnotám objektových a numerických funkcí.

Podmínky a relativní stavy

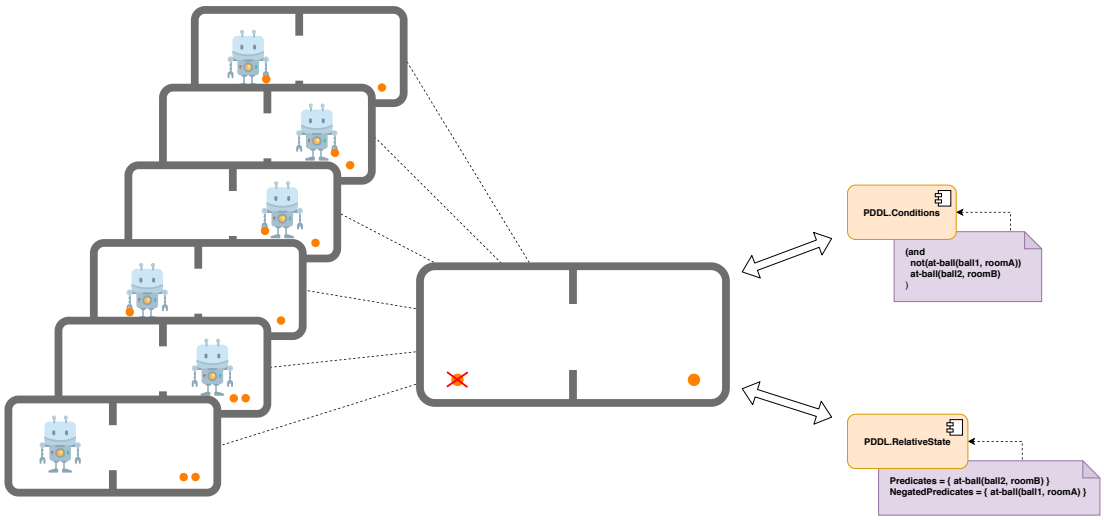
Druhou nejdůležitější entitou jsou tzv. podmínky (*conditions*). Podmínka je zobecněním stavu, jde primárně o seznamem nějakých omezujících vlastností, které ve výsledku pokrývají celou třídu odpovídajících stavů.

Diagram na obr. 6.12 ukazuje jednoduchou podmínku („Míček 2 je v druhé místnosti a míček 1 není v první místnosti.“) a odpovídající stavy této podmínce. Obecně může být podmínka libovolný logický výraz, včetně konjunkcí, disjunkcí, negací a všech dalších logických operací. Více v podkapitole 6.4.2 popisující logické výrazy.

Přirozenou vlastností, kterou budeme po podmínkách chtít, je vyhodnocování vůči stavům. Např. pro cílové podmínky problému chceme ověřit, zda jsou pro předaný stav splněny. O tyto a další operace spojené s logickými výrazy se stará primárně komponenta `EvaluationManager`.

Samostatnou kapitolou jsou pak podmínky v CNF (*konjunktivně-normální formě*), které jsou vhodné pro určité operace. Více detailů opět v podkapitole 6.4.2.

Doposud téměř zamlčovanou entitou jsou tzv. *relativní stavy*. Jde o alternativní vyjádření podmínek v rámci plánovacího problému, přičemž daná entita je skutečně stav a jde s ní pracovat jako se stavem.

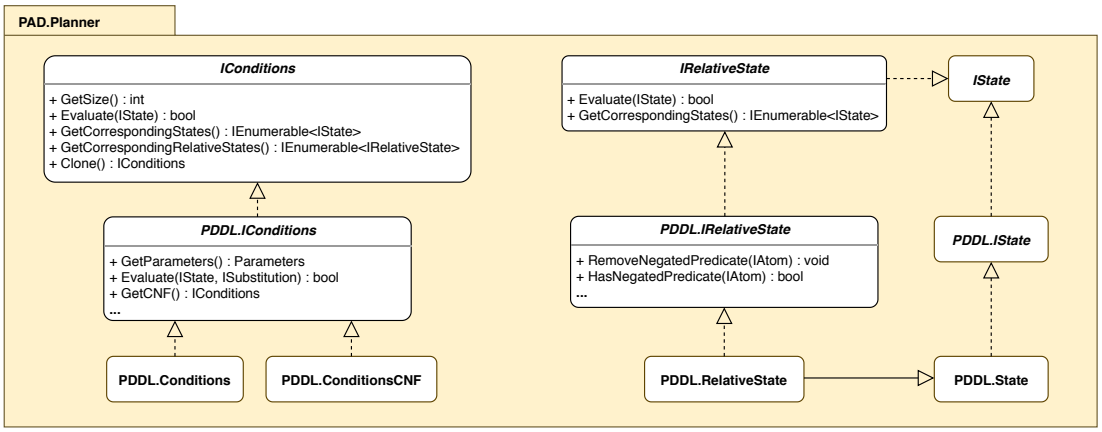


Obrázek 6.12: PDDL podmínka a ekvivalentní PDDL relativní stav v doméně Gripper se dvěma míčky (nalevo odpovídající stavy)

Rozdíl oproti klasickému stavu je však v sémantice: zatímco u klasického stavu předpokládáme, že skutečnosti, které nejsou ve stavu explicitně uvedené, implicitně neplatí, pak u relativního stavu skutečnosti, které nejsou explicitně uvedeny, mohou a nemusí platit. Navíc jsou oproti stavům rozšířeny o vyjádření negovaných faktů, tj. explicitně můžeme uvést, že něco neplatí.

Relativní stavy tedy určují, stejně jako podmínky, celou třídu odpovídajících stavů. Digram na obr. 6.12 ukazuje příklad takové podmínky a ekvivalentní relativní stav (společně s odpovídajícími klasickými stavy).

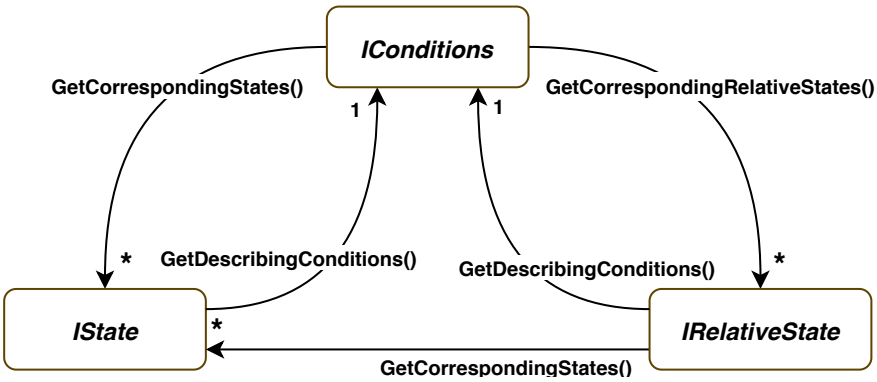
Vyjadřovací silou však nejsou relativní stavy ekvivalentní s podmínkami. Jsou konstruovány pro vyjádření konjunkce a negace, ale disjunkce a případné kvantifikace neumějí. Pokud bychom tedy chtěli vyjádřit např. podmínku *A or not(B)*, pak k tomu budeme potřebovat dva relativní stavy, jeden odpovídající *A* a druhý odpovídající *not(B)*.



Obrázek 6.13: Hierarchie PDDL podmínek a PDDL relativních stavů

Diagram na obr. 6.13 přibližuje hierarchii PDDL podmínek a PDDL relativních stavů. Jde vidět, že relativní stav je rozšířením implementace klasického stavu s přidáním podpory pro manipulace s negovanými predikáty.

Zatímco stavy se používají jako vyhledávací uzly při algoritmu dopředného hledání (*Forward-Search*), podmínky a relativní stavy se podobným způsobem používají při zpětném hledání (*Backward-Search*). Jak se dopřednou/zpětnou aplikací akce vytvoří nový stav, podmínka nebo relativní stav, popisuje podkapitola 6.4.4.



Obrázek 6.14: Možnosti převodů mezi jednotlivými entitami problému

Diagram na obr. 6.14 ilustruje vzájemnou převoditelnost tří zmíněných entit plánovacího problému. Pro podmínky lze vygenerovat všechny stavy a relativní stavy, které patří do třídy stavů určenou podmínkami (neboli všechny stavy, které jsou pro dané podmínky splněny). Totožné vygenerování lze provést pro relativní stavy. Navíc speciálně lze vygenerovat na základě stavu a relativního stavu popisující podmínky (která však obecně určí větší třídu než samotný původní stav). Tyto metody však slouží spíše pro speciální účely, než pro účely efektivního prohledávání.

6.4.2 Výrazy a jejich vyhodnocování

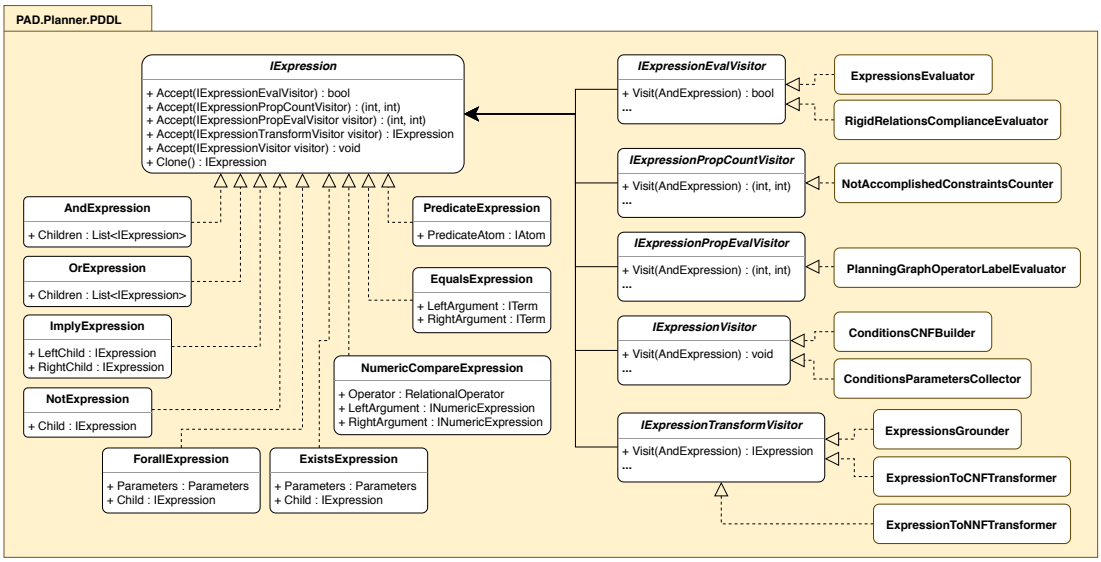
Jak již bylo uvedeno, podmínky jsou definovány nějakými logickými výrazy. Takový logický výraz je stromovou strukturou, kde vnitřní uzly odpovídají logickým operacím a listy stromu jsou logickými primitivy (např. predikáty), které mohou nabývat logických hodnot *true* nebo *false*.

Diagram na obr. 6.15 ukazuje hierarchii standardních logických výrazů. Každý z nich implementuje rozhraní `IExpression` a podporuje vyhodnocování různými *evaluátory*. Tyto evaluátory jsou realizovány pomocí návrhového vzoru visitor[12]: strom výrazu je postupně traverzován a výsledná hodnota se pak skládá od listů stromu zpět nahoru ke kořenu. Jak jde vidět, dostupných evaluátorů je celá řada a budeme se k nim v dalších částech dále vracet a blíže je popisovat.

Všechny evaluátory jsou umístěny v komponentě `EvaluationManager`, která tyto objekty inicializuje *líně*, až v momentě reálného použití, čímž nedochází ke zbytečné režii vytvářením objektů, které v daném běhu programu nebudou nikdy použity.

Vyhodnocování logických výrazů

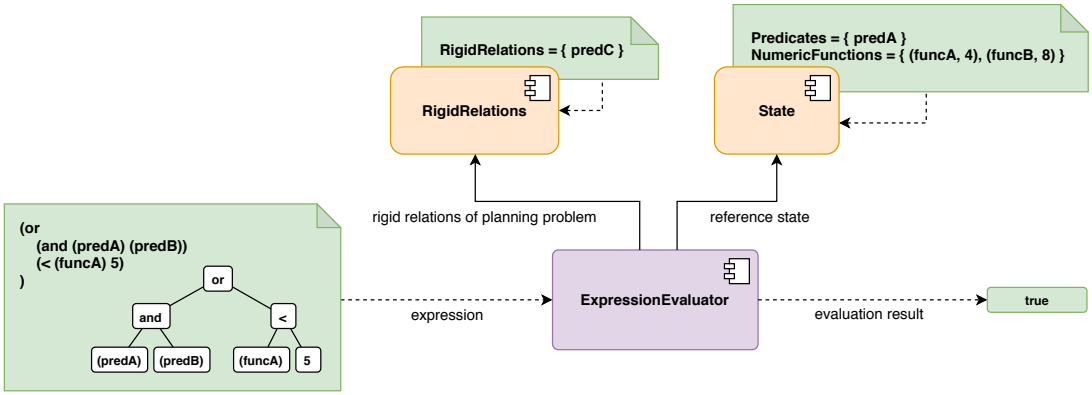
Nejzákladnějším evaluátorem je `ExpressionEvaluator` provádějící elementární vyhodnocení logického výrazu, který je mu předán na vstupu, vůči nějakému stavu. Vyhodnocení ilustruje obr. 6.16. Evaluátor se při procházení stromu



Obrázek 6.15: PDDL logické výrazy a jejich evaluátory

dostane až do listu, kde může narazit na tři typy výrazů: predikátový výraz, numerické porovnání, nebo porovnání na objektovou rovnost.

V případě predikátového výrazu dojde k ověření, zda daný predikát platí v daném referenčním stavu, příp. v rigidních podmínkách plánovacího problému.



Obrázek 6.16: Vyhodnocování logických výrazů v PDDL

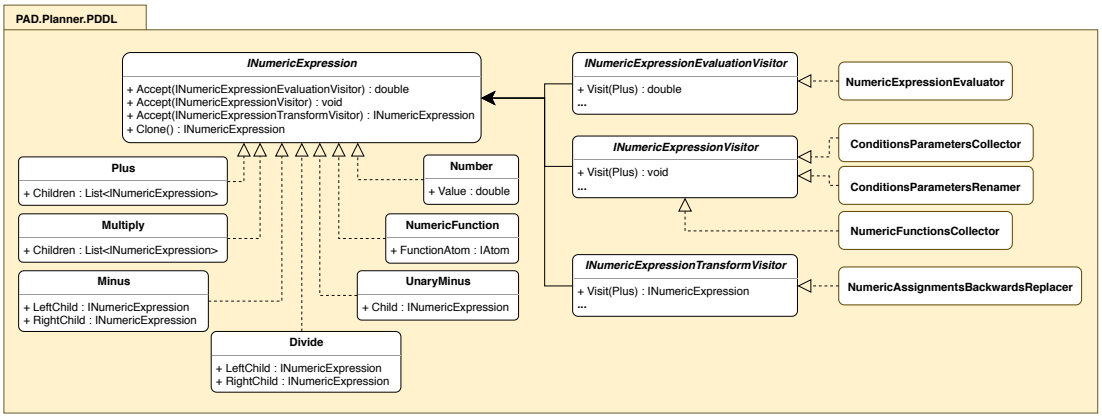
U numerického porovnání dochází nejprve k vyhodnocení obou argumentů (numerických výrazů) a výsledné hodnoty jsou porovnány na daný relační operátor. Výsledkem je logická konstanta, kterou již lze dále použít v rámci vyhodnocení logického výrazu.

Kromě numerických fluentů (tj. numerických funkcí) podporuje PDDL 3.1 také objektové fluenty, což jsou funkce, jejichž návratová hodnota je konstanta. Operátor = pak umožňuje porovnávat tyto hodnoty na objektovou rovnost, příp. porovnat hodnotu funkce přímo s nějakou konstantou.

Numerické výrazy

Jak bylo zmíněno, v rámci logických výrazů mohou být použity i numerické výrazy, které jsou vyhodnoceny a následně porovnány relačním operátorem. Numerické výrazy se však mohou vyskytovat i v rámci efektů operátorů apod.

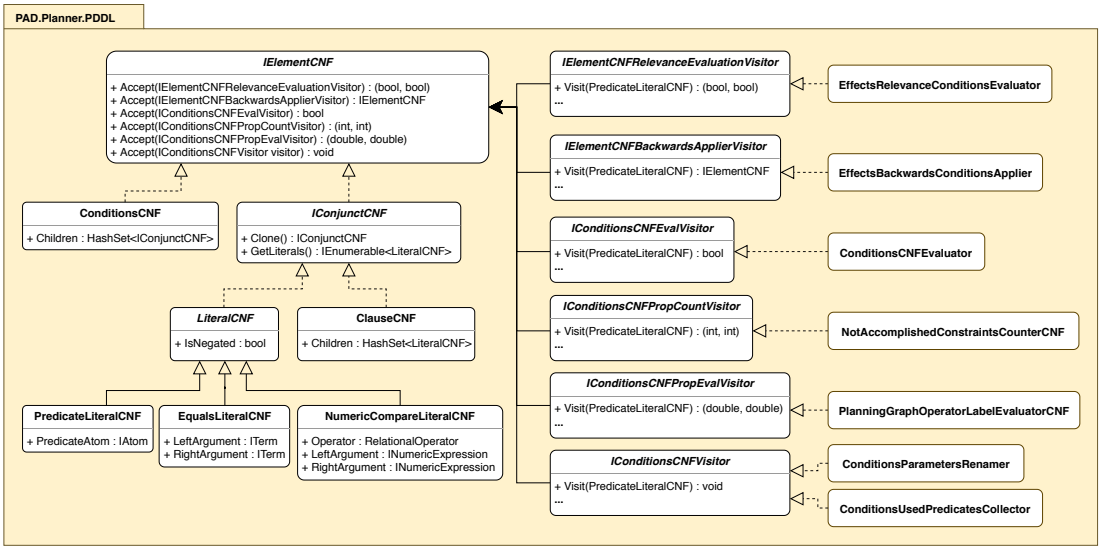
Diagram na obr. 6.17 popisuje hierarchii numerických výrazů. Stejně jako u logických výrazů půjde o stromovou datovou strukturu, podporující vyhodnocování visitory. Numerické výrazy mají své vlastní evaluátory, které (jako v případě logických výrazů) buď daný výraz určitým způsobem vyhodnocují, počítají nad ním nějakou vlastnost, nebo je transformují do jiných výrazů.



Obrázek 6.17: PDDL numerické výrazy a jejich evaluátory

Logické výrazy v CNF

V určitých případech není efektivní používat standardní implementaci logických výrazů popsanou výše a je potřeba převést ji do CNF[4] reprezentace.



Obrázek 6.18: PDDL logické výrazy v CNF a jejich evaluátory

Logický výraz v CNF má striktnější formu, kdy je původní logická formule vyjádřena ve formě konjunkcí klauzulí (kde klauzule je v tomto smyslu buď přímo literál, nebo disjunkce literálů). V našem kontextu jsou literály predikátové výrazy, či porovnávací výrazy popsané výše.

Logické výrazy v CNF mají svou vlastní hierarchii, popsanou na obr. 6.18. Výraz v klasické logické formuli lze ve frameworku PAD snadno převést do reprezentace v CNF pomocí komponenty `ConditionsCNFBuilder`, která interně

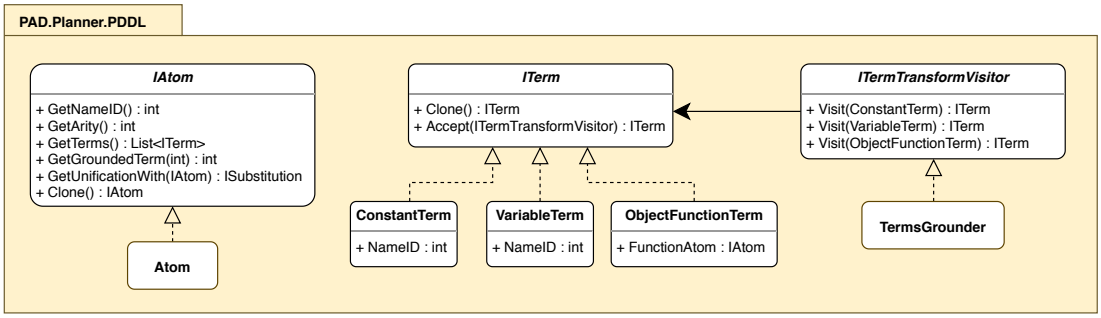
používá transformery pro převod původní formule nejprve do NNF, následně do CNF a nakonec ji převede do CNF hierarchie.

Výrazy v CNF se ve frameworku PAD používají zejména při zpětném prohledávání, kdy se reverzní aplikací akcí získávají nové podmínky, které se odzadu přibližují počátečnímu stavu.

6.4.3 Atomy, termy a groundování

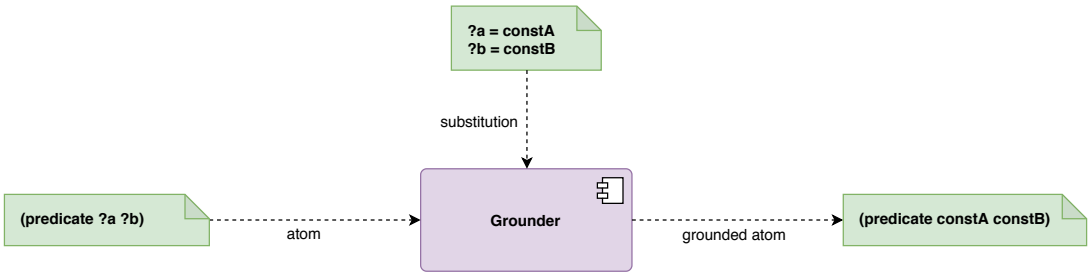
V logických a numerických výrazech a stavech jsme již viděli použití predikátů a numerických či objektových funkcí. Tyto predikáty a funkce jsou ve frameworku identifikovány a referencovány jako zapouzdřená entita zvaná *atom*.

Atom je definován jménem predikátu/funkce a seznamem *termů*, tj. argumentů atomu. Termem může být buďto konstanta, proměnná nebo objektová funkce. Pokud jsou všechny termy atomu konstantami, pak je atom *groundovaný*, v opačném případě je atom negroundovaný (nebo též říkáme *liftovaný*). Hierarchii tříd pro atomy a termy ve frameworku PAD ilustruje diagram na obr. 6.19.



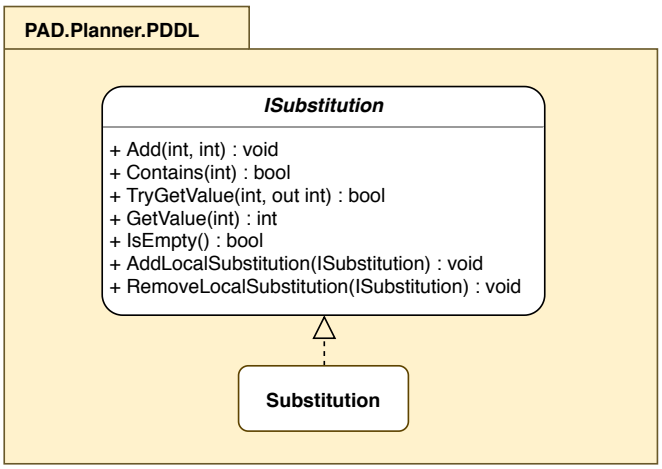
Obrázek 6.19: PDDL atomy a termy

Liftované atomy (resp. obecně celé logické a numerické výrazy) lze chápat jako třídu atomů, zatímco *groundingem* dostaneme konkrétní instance. Pro grounding nějakého liftovaného atomu potřebujeme dostat příslušnou substituci proměnných. Příklad groundingu atomu na základě substituce ilustruje obr. 6.20.



Obrázek 6.20: Příklad groundingu PDDL atomu

O grounding atomů, termů, ale i všech výrazů se stará komponenta `Grounder`, která je součástí komplexnějšího `GroundingManageru`. Více grounding přiblížíme hned v následující podkapitole 6.4.4 o operátorech, kde dostane konkrétní obrysy. Samotná substituce se skládá z jednoduchého mapování proměnných na příslušné substituované konstanty, viz požadované rozhraní pro substituce na obr. 6.21.

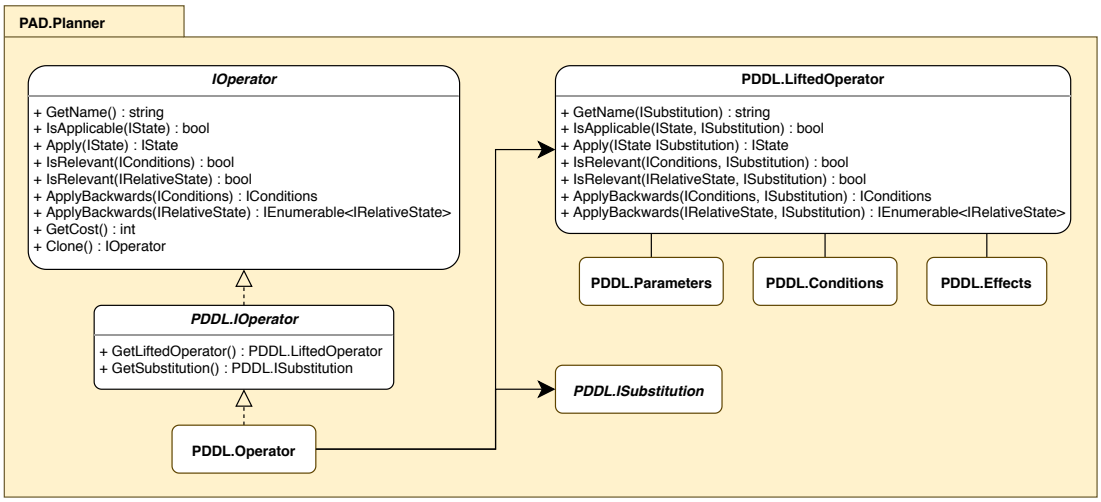


Obrázek 6.21: PDDL substituce proměnných

V celé této části zmiňujeme jména proměnných, konstant, predikátů a funkcí, nicméně v programu se jejich původní textová označení nepoužívají. Místo toho se užívají příslušná ID, která zprostředkovává `IDManager`. Ten podporuje i funkce pro zpětný převod, např. při extrakci řešení zpět do původní reprezentace.

6.4.4 Operátory

Jedinými prostředky v plánovacím problému, které mění stav systému, jsou *operátory*, potažmo *akce*. V kapitole 2.1 o klasické reprezentaci byly akce definovány jako jednotlivé instance plánovacích operátorů. Tento princip je v reprezentaci PDDL plně zachován, pouze si ve frameworku PAD lehce upravíme názvosloví.



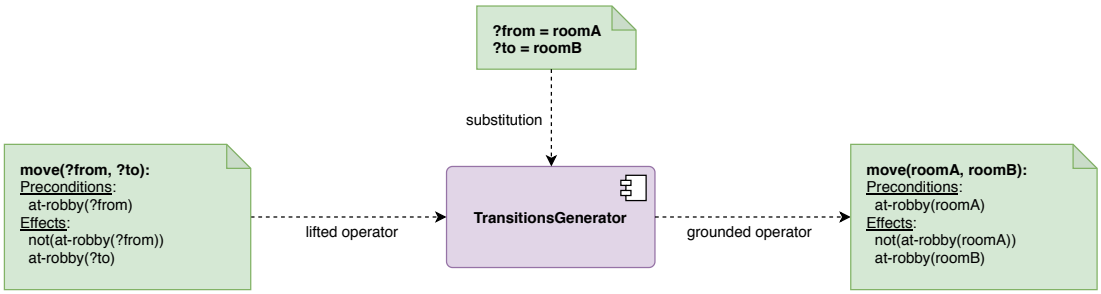
Obrázek 6.22: PDDL operátory

Operátory z klasické reprezentace budeme nazývat *liftovanými operátory* a akcím budeme říkat *groundované operátory*. Jedním z důvodů je to, že v reprezentaci SAS$^+$ je zvykem používat pouze označení *operátor* (ve smyslu groundovaný operátor, liftované operátory SAS$^+$ nemá).

Definici liftovaných operátorů dostaneme přímo z PDDL vstupu, groundovaná varianta pak bude technicky realizovaná pouze jako reference na liftovaný operátor a příslušný grounding, tj. substituci proměnných v parametrech operátoru. Tyto vztahy a strukturu operátorů ilustruje diagram na obr. 6.22.

Počet liftovaných operátorů tedy bude od začátku neměnný, zatímco groundované varianty se za běhu programu dynamicky vytvářejí, dle potřeby. Většina funkcionality je však delegována právě na referencované liftované operátory, jak jde z obr. 6.22 vidět. Liftovaný operátor v sobě dále zapouzdřuje komponenty parametrů, předpokladů ve formě podmínek a efekty.

Příklad vytvoření groundovaného operátoru na základě substituce ilustruje obr. 6.23. Jde vidět, že výsledkem je grounding jak předpokladů operátoru, tak efektů operátoru. Pro tvorbu všech groundovaných instancí jednoho liftovaného operátoru bychom použili všechny možné substituce proměnných v parametrech operátoru.



Obrázek 6.23: Příklad groundingu PDDL operátoru

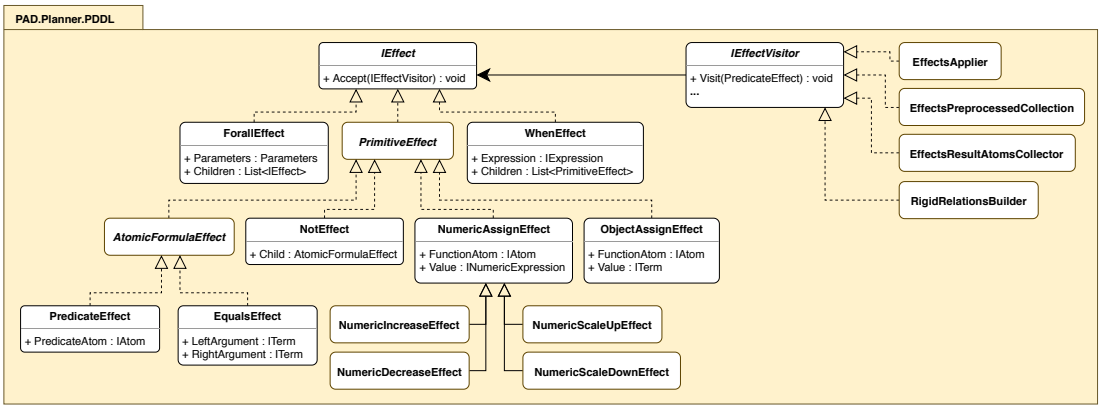
Výše jsme již zmínili, že liftovaný operátor si drží komponentu parametrů `PDDL.Parameters`, což je jednoduchá struktura, která udržuje informace o použitých proměnných v kontextu celého operátoru (dané parametry navíc mohou být typované). Tato komponenta může být kromě operátoru dále použita např. v logických výrazech typu `forall` nebo `exists`, kde jsou dané parametry kvantifikované pro celý další logický podvýraz.

Aplikace operátoru

Aby bylo možné groundovaný operátor aplikovat na nějaký stav, musí být splněna podmínka *aplikovatelnosti*, tzn. musí být splněny předpoklady operátoru. Vzhledem k tomu, že předpoklady jsou zapsány ve formě PDDL podmínky, použije se standardní vyhodnocení popsané v části 6.4.2. Pokud podmínky předpokladů operátoru pro daný stav platí, pak je operátor na daný stav aplikovatelný.

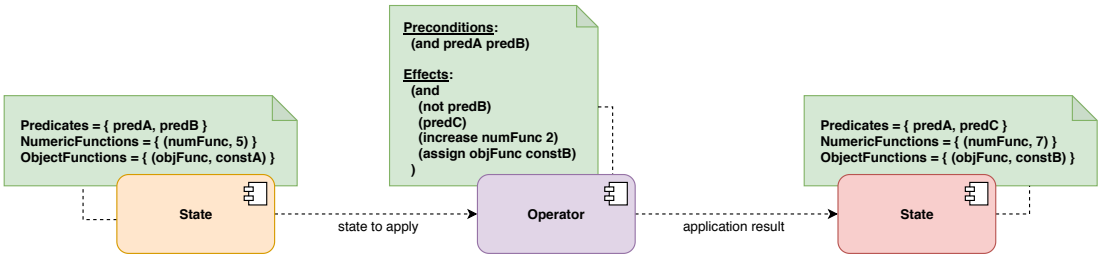
Samotná aplikace operátoru pak znamená aplikování příslušných efektů. Na obr. 6.22 vidíme, že operátor má pod sebou komponentu `PDDL.Effects`, což je kolekce jednotlivých instancí efektů. Strukturu PDDL efektu ilustruje diagram na obr. 6.24.

Jednotlivými efekty v rámci efektů jednoho operátoru mohou být kromě klasického přidání/odebrání predikátu také přiřazení a modifikace hodnot numerických funkcí, přiřazení hodnot objektových funkcí, kvantifikované *forall* efekty a nakonec podmíněné efekty (tj. efekty, které mají dodatečnou podmínku, a u kterých se provede aplikace pouze tehdy, pokud je tato dodatečná podmínka splněna).



Obrázek 6.24: Struktura PDDL efektu

Proces aplikace na konkrétní stav zajišťuje komponenta `EffectsApplier`, což je implementace návrhového vzoru visitor, traverzujícího postupně jednotlivé efekty operátoru a modifikující daný stav určený pro aplikaci. Příklad aplikace operátoru ilustruje obr. 6.25, který obsahuje přidání predikátu, odebrání predikátu a modifikace numerické a objektové funkce.



Obrázek 6.25: Příklad aplikace PDDL operátoru

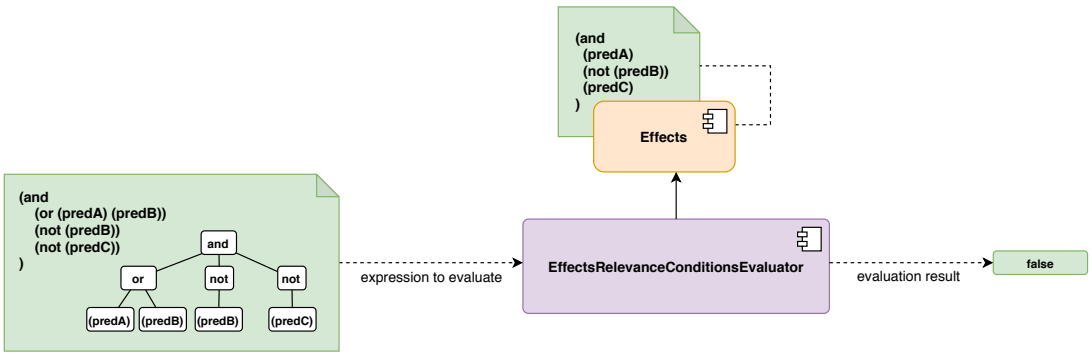
Reverzní aplikace operátoru

V kapitole 2.1 jsme uváděli kromě dopředné aplikace akcí na stav rovněž zpětnou aplikaci akcí na podmínky. Takovéto zpětné aplikace se následně používají ve zpětném prohledávání (Backward-Search), kdy posouváme cílové podmínky směrem k počátečnímu stavu problému.

Groundovaný operátor v PDDL bude zpětně aplikovatelný na nějakou podmínku za předpokladu, že je pro dané podmínky *relevantní*. Vyjdeme z původní definice relevantnosti operátoru (definice 16, kapitola 2.1): operátor je pro dané podmínky relevantní, pokud efekty operátoru pozitivně přispívají k podmínkám a navíc žádný efekt není s podmínkami v konfliktu.

Původní podmínky převedeme do CNF reprezentace a ověříme relevantnost komponentou efektů `EffectsRelevanceConditionsEvaluator`, což je visitor traverzující CNF výrazy a ověřující podmínky relevantnosti. V kontextu CNF výrazu to znamená, že v alespoň jednom konjunktu (literál nebo disjunkce literálu) musí platit, že nějaký literál je pozitivně ovlivněn nějakým efektem, a zároveň v žádném konjunktu nesmí dojít ke konfliktu, tj. očekáváme platnost nějakého faktu a efekt chce nastavit opak.

Příklad vyhodnocení relevantnosti operátoru ilustruje obr. 6.26: první dva konjunkty podmínek jsou v pořádku, třetí je v konfliktu s jedním z efektů, takže celý operátor je nerelevantní vůči daným podmínkám.

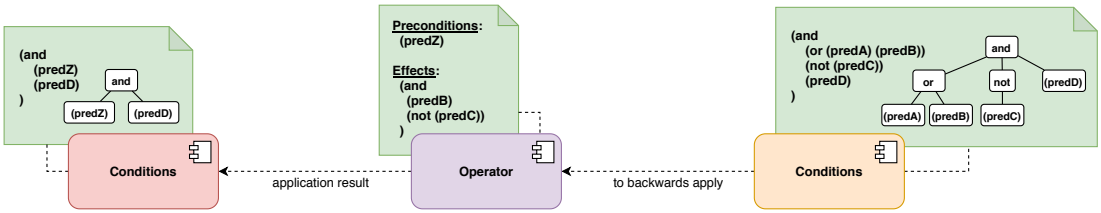


Obrázek 6.26: Příklad vyhodnocení relevantnosti PDDL operátoru

Zcela analogicky probíhá kontrola relevantnosti pro relativní stav, pouze netraverzujeme CNF výraz, ale procházíme všechny elementy relativního stavu (množinu predikátů, množinu negovaných predikátů atd.).

Je-li je daný operátor pro dané podmínky relevantní, můžeme provést zpětnou aplikaci, díky které z původních podmínek dostaneme podmínky nové, tj. předcházející. Zpětná aplikace opět bude probíhat analogicky dle definice popsané v části o klasické reprezentaci (definice 17, kapitola 2.1). Předchůdce podmínek dostaneme tak, že z původních podmínek odstraníme pozitivně přispívající efekty a naopak přidáme předpoklady operátoru.

Příklad zpětné aplikace operátoru ilustruje obr. 6.27 – vstupní podmínka byla převedena do CNF formy a odebráním pozitivně přispívajících efektů se z původní podmínky odstranila celá první klauzule (byla splněna jedním z efektů) i druhý negovaný literál. K výsledné podmínce se do konjunkce přidaly předpoklady operátoru a máme konečný výsledek zpětné aplikace.



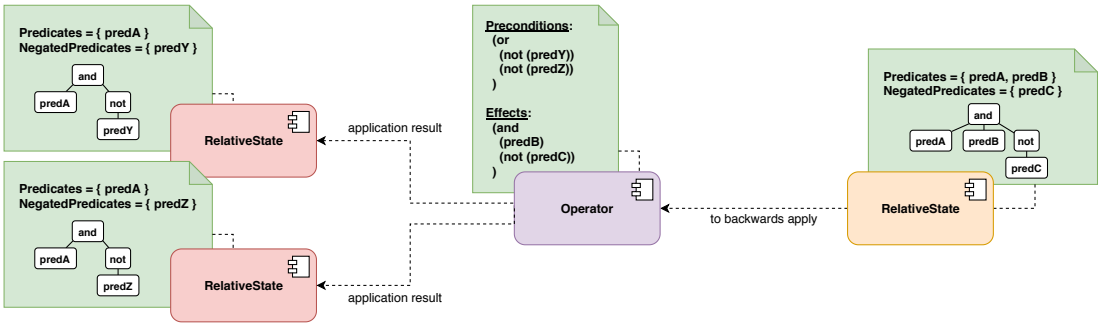
Obrázek 6.27: Příklad zpětné aplikace PDDL operátoru na podmínky

Proč potřebujeme mít vstupní podmínky v CNF? Za prvé, abychom se zbavili syntaktického cukru v podobě dalších logických operátorů (`imply`, `forall` apod.) a měli jasnou strukturu konjunkce disjunkcí. Taková struktura nám pak umožňuje snadné vyhodnocování relevantnosti i zpětnou aplikaci – např. v posledním kroku aplikace nám stačí pouze přidat do konjunkce předpoklady, které již máme rovněž připravené v CNF.

Zpětná aplikace operátoru na relevantní stav bude opět zcela analogická postupu s podmínkami. Jediný rozdíl je ten, že relativní stavy neumožňují vyjádřit disjunkci, čili pracujeme nativně s konjunkcí literálů a vyhodnocování je tak mnohem rychlejší.

Toto je jeden z důvodů existence dvou různých entit pro zpětné prohledávání. Zatímco podmínky mají obecně větší vyjadřovací sílu, zpracovávání podmínek může být náročné z důvodů převodu na CNF (který navíc může entitu až exponenciálně zvětšit). Na druhou stranu, u relativních stavů není třeba převádět nic – máme k dispozici množiny predikátů a negovaných predikátů a provedeme jednoduché množinové operace.

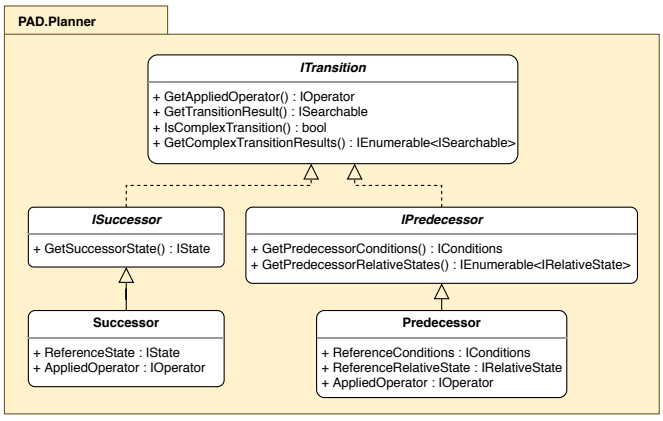
Výsledkem aplikace však nemusí být jeden relativní stav, jak ilustruje příklad zpětné aplikace na obr. 6.28, a větvící faktor se tak může za určitých okolností zvětšovat, v porovnání s podmínkami.



Obrázek 6.28: Příklad zpětné aplikace PDDL operátoru na relativní stav

6.4.5 Generování následníků a předchůdců

Nyní máme vše potřebné, abychom mohli generovat následníky stavů a předchůdce podmínek a relativních stavů. Tuto funkcionalitu pro plánovací problém zajišťuje komponenta `TransitionsGenerator`.



Obrázek 6.29: Následníci a předchůdci ve frameworku PAD

Vrátit všechny možné následníky pro předaný stav znamená pro každý liftovaný operátor vygenerovat všechny možné substituce parametrů operátoru a příslušné groundované varianty testovat na aplikovatelnost pro daný stav. Groundované operátory, které jsou aplikovatelné, jsou zapouzdřeně vráceny jako kolekce následníků `ISuccessor`.

Pro vygenerování všech možných předchůdců podmínek či relativních stavů se opět najdou všechny možné groundingy plánovacích operátorů a testují se

na relevantnost. Relevantní operátory pro dané podmínky či relativní stav jsou vráceny jako kolekce předchůdců `IPredecessor`.

Všechny dopředné/zpětné přechody jsou vráceny v rámci `IEnumerable`, tedy ve skutečnosti enumerátoru, který generuje dané přechody *líně*, až v momentě použití. Diagram na obr. 6.29 ilustruje hierarchii dopředných i zpětných přechodů v rámci frameworku PAD. Z daného dopředného či zpětného přechodu lze snadno získat odpovídající operátor a výsledek aplikace operátoru.

6.4.6 Podpora heuristik

Nyní krátce popíšeme principy heuristik implementačně závislých na vnitřní PDDL reprezentaci. Půjde o *STRIPS*[34] heuristiku a zejména pak heuristické funkce závislé na *relaxaci*.

STRIPS heuristika

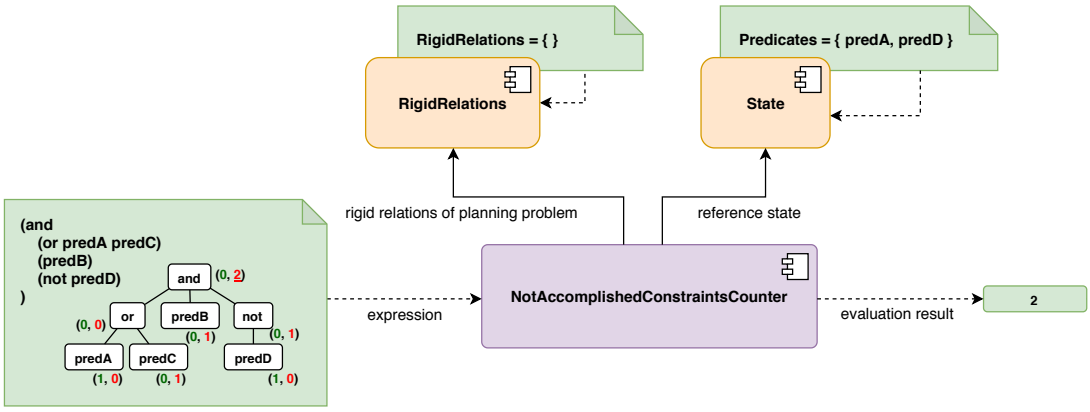
Tzv. *STRIPS heuristika* je jedna z prvních heuristik pro navádění ve stavovém prostoru. Je založena na myšlence, že čím více cílů (cílových podmínek) je v daném stavu splněno, tím je stav pravděpodobněji blíže cíli. Jakkoli se nejedná o ideální heuristiku, může dobře sloužit jako určitá reference pro sofistikovanější přístupy.

Původní STRIPS přístup hovoří o počtu nesplněných cílů v kontextu klasické reprezentace. Ve frameworku PAD však podporujeme komplexní podmínky ve formě logických výrazů. Adaptujme tedy původní myšlenku následujícím způsobem: budeme traverzovat datovou strukturu logického výrazu visitorem podobně jako při logickém vyhodnocování popsaném v části 6.4.2, ale pro jednotlivé podvýrazy si spočítáme počet splněných a nesplněných cílů vůči danému stavu. Nakonec vrátíme počet nesplněných cílů, což je výsledná hodnota heuristiky.

Výsledky jednotlivých podvýrazů rekurzivně složíme následujícím způsobem:

- operátor `and` – sečteme počty splněných cílů pro jednotlivé podvýrazy a to samé uděláme pro nesplněné cíle podvýrazů,
- operátor `or` – z počtů splněných cílů pro podvýrazy vybereme minimum a to samé uděláme pro nesplněné cíle podvýrazů,
- operátor `not` – počty splněných a nesplněných cílů pro podvýraz prohodíme,
- operátor `imply` – vyjádříme pomocí disjunkce a vyhodnotíme pomocí prostředků výše,
- primitivní výraz (predikát, rovnost, numerické porovnání) – vyhodnotíme standardně vůči danému stavu, zda jsou v daném stavu splněny, či nikoli,
- operátor `forall` – zjistíme všechny možné substituce parametrů a vyhodnotíme dohromady jako operátor `and`,
- operátor `exists` – zjistíme všechny možné substituce parametrů a vyhodnotíme dohromady jako operátor `or`,

Příklad vyhodnocování počtu nesplněných cílů ilustruje obr. 6.30. Vyhodnocování zajišťuje komponenta `NotAccomplishedConstraintsCounter` a analogicky pro CNF výrazy pak komponenta `NotAccomplishedConstraintsCounterCNF`. Oba tyto evaluátory jsou součástí globálního `EvaluationManager`.

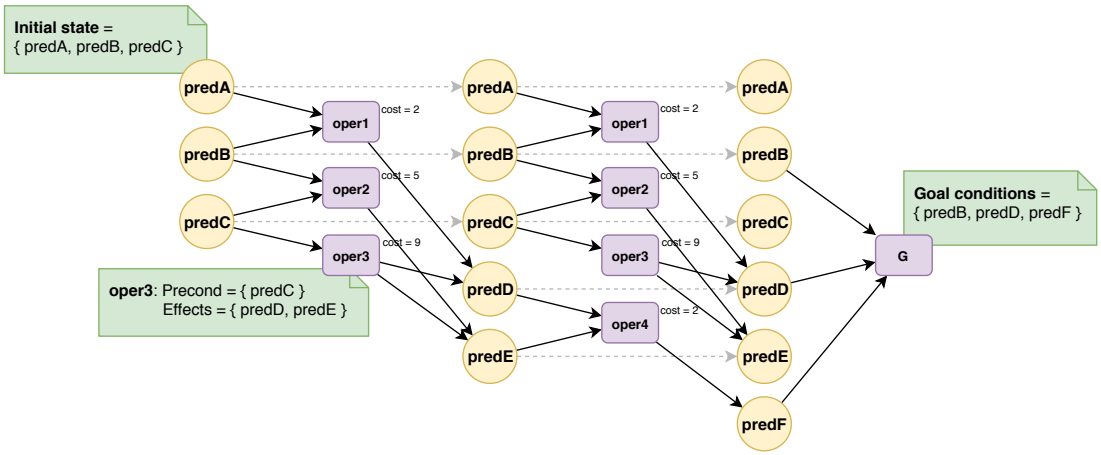


Obrázek 6.30: Příklad vyhodnocování počtu nesplněných cílů v PDDL

Heuristiky založené na relaxaci

Jednou z pokročilejších technik pro heuristiky je tzv. *relaxace*. Ta vychází z myšlenky, že v efektech operátorů zanedbáme negativní efekty, tj. takové, co ze stavů nějaké fakty odebírají. Plánovací problém s takto pozměněnými operátory se nazývá *relaxovaný problém* a jeho řešení nám může dát určitý odhad pro řešení problému původního.

Pro stav, pro který chceme spočítat heuristickou hodnotu, sestavíme relaxovaný problém, kde daný stav bude počátečním stavem. Cena řešení tohoto problému pak poslouží jako heuristická hodnota. Takto pojatou heuristikou, kdy počítáme optimální řešení relaxovaného problému, je ve frameworku PAD heuristika zvaná `PerfectRelaxationHeuristic`.



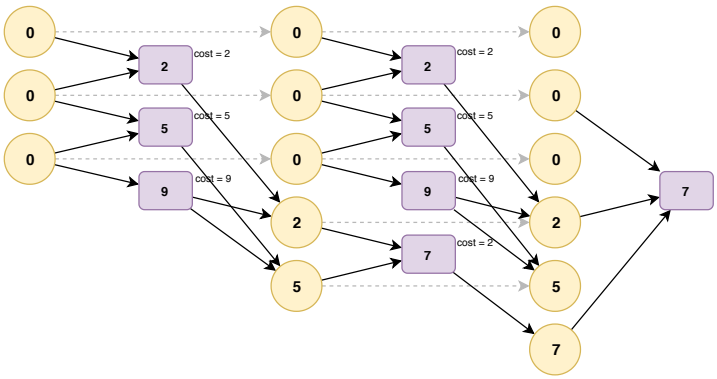
Obrázek 6.31: Příklad PDDL relaxovaného plánovacího grafu

Hledání optimálního řešení relaxovaného problému však bude pro mnoho případů pomalé, a proto jiné přístupy místo toho počítají odhad ceny relaxovaného problému pomocí tzv. *relaxovaného plánovacího grafu*.

Takový graf se skládá ze střídajících se prepozičních a akčních vrstev. První vrstva je prepoziční a obsahuje predikáty z počátečního stavu. Následující akční vrstva obsahuje takové akce, které jsou aplikovatelné na předcházející vrstvu, a sama svými efekty (vyjma negativních efektů) generuje následující prepoziční vrstvu. Takto se pokračuje, dokud nemáme prepoziční vrstvu splňující cílové pod-

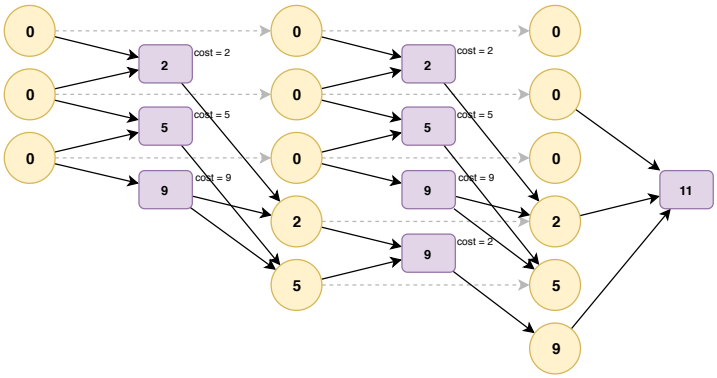
mínky. Příklad relaxovaného plánovacího grafu s naznačenými cenami operátorů ilustruje obr. 6.31.

Nyní zkonstruovaný plánovací graf zepředu vyhodnotíme (viz obr. 6.32): predikáty v první vrstvě mají nulovou cenu; cenu akce v následující vrstvě určíme jako maximum z cen předcházejících predikátů + ceny příslušného operátoru; ceny v další vrstvě predikátů se pak spočítají naopak z minim cen předcházejících uzlů atd., dokud nedorazíme do cílových podmínek. Výsledek u cílových podmínek udává odhad ceny řešení relaxovaného problému. Popsané heuristice ve frameworku PAD říkáme `MaxRelaxationHeuristic`.



Obrázek 6.32: Vyhodnocení plánovacího grafu *maxovou* strategií

Alternativně můžeme relaxovaný plánovací graf vyhodnotit tak, že u výpočtu cen akcí nevybíráme maximální cenu z předcházejících uzlů grafu, nýbrž vezmeme součet. Takový výpočet ilustruje obr. 6.33 a ve frameworku PAD se nazývá `AdditiveRelaxationHeuristic`.



Obrázek 6.33: Vyhodnocení plánovacího grafu *aditivní* strategií

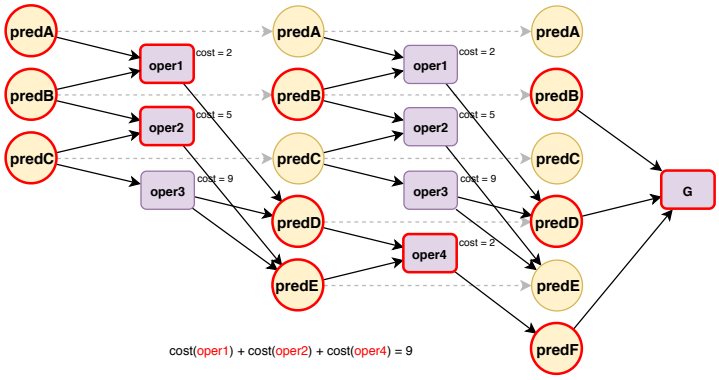
Nejpokročilejším typem heuristiky založené na relaxaci je tzv. *FF-heristika* (ve frameworku PAD pod názvem `FFHeuristic`). Opět sestrojíme relaxovaný plánovací graf, ale vyhodnocení tentokrát provádíme odzadu od cílových podmínek.

Jednotlivé uzly grafu si můžeme *označovat* (na počátku jsou všechny neoznačené, vyjma cílového uzlu). O akčních uzlech v grafu dále řekneme, že jsou *naplněné*, pokud jsou všechny přímo předcházející uzly v grafu označené, zatímco o predikátových uzlech řekneme, že jsou naplněné, pokud alespoň jeden z předchůdců v grafu je označený.

Následně opakovaně aplikujeme následující pravidla, dokud všechny označené uzly v grafu nejsou naplněné (pokud jich je aplikovatelných více, má prioritu pravidlo s nižším číslem):

1. Označ všechny přímé předchůdce označeného nenaplněného akčního uzlu.
2. Označ přímého předchůdce označeného nenaplněného predikátového uzlu, který je predikátovým uzlem (neměnné přechody, na diagramu naznačeny šedivou čarou).
3. Označ nejlepšího možného předchůdce označeného nenaplněného predikátového uzlu, tj. operátor s nejnižší cenou (nazývá se *support action*).

Příklad vyhodnocení dříve sestrojeného grafu na obr. 6.31 pro výpočet FF heuristiky ilustruje obr. 6.34. Výslednou heuristickou hodnotu udává součet cen označených akčních uzlů v grafu.



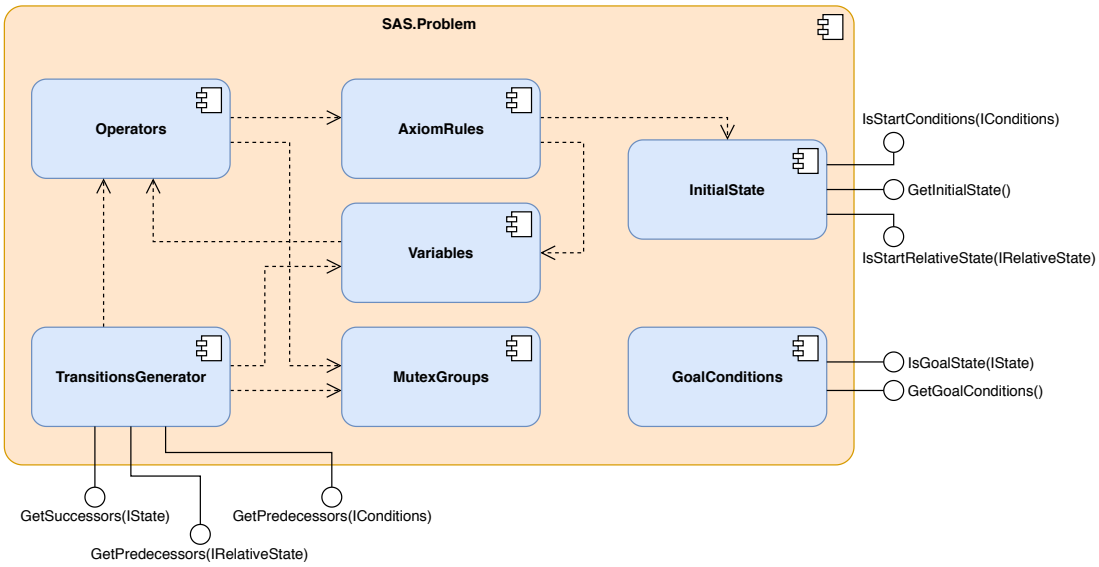
Obrázek 6.34: Vyhodnocení plánovacího grafu *FF* strategií

6.5 Plánovací problém SAS$^+$

V následující části popíšeme strukturu a fungování plánovacího problému v reprezentaci SAS$^+$. Spousta aspektů plánovacího problému bude zcela analogická k PDDL reprezentaci a budeme na ní zpětně odkazovat. Ve spoustě částí zas půjde o značné zjednodušení z povahy reprezentace SAS$^+$.

Diagram na obr. 6.35 popisuje klíčové vnitřní komponenty `SAS.Problem`, spolu s naznačenými závislostmi a poskytovaným rozhraním. Opět si stručně popišme funkce a kontext jednotlivých komponent a v následujících podkapitolách pak rozebereme detailněji specifika plánovacího problému v reprezentaci SAS$^+$.

- `Operators` – komponenta spravující kolekci (groundovaných) operátorů daného problému; podrobněji o SAS$^+$ operátorech dále v části 6.5.3 níže,
- `AxiomRules` – komponenta spravující kolekci *axiomatických pravidel* daného problému, včetně metod pro aplikaci; více o axiomech dále v části 6.5.4,
- `Variables` – komponenta spravující informace o proměnných používaných v rámci daného problému, např. jméno, ID, význam jednotlivých hodnot, příp. zařazení v rámci axiomatických vrstev, rigidita proměnné apod., viz podkapitola 6.5.2,



Obrázek 6.35: Komponenty SAS$^+$ plánovacího problému

- `MutexGroups` – komponenta spravující informace o tzv. *mutexových skupinách*, které dodávají další omezení na vzájemně se vylučující vztahy v jednotlivých stavech systému; podrobnosti v části 6.5.5,
- `TransitionsGenerator` – komponenta zajišťující generování nových přechodů v rámci plánovacího problému, tj. následníky pro stavy či předchůdce pro podmínky a relativní stavy; více o generování v části 6.5.6,
- `InitialState` – počáteční stav plánovacího problému; rovněž je používán při aplikaci axiomatických pravidel, neboť zároveň definuje defaultní hodnoty axiomatických proměnných (více v části 6.5.4 o axiomech); počáteční stav je možné manuálně měnit, ale ve většině případů zůstane stejný po celou dobu životnosti plánovacího problému, více o stavech v části 6.5.1,
- `GoalConditions` – cílové podmínky plánovacího problému; opět je možné je v případě potřeby změnit, ale v typickém případě se tak dít nebude a cílové podmínky zůstávají po celou dobu stejné, více v části 6.5.1.

Plánovací problém musí podporovat práci se základními prohledávacími entitami (stavy, podmínky, relativní stavy) – tj. umět generovat následníky stavů, předchůdce podmínek a relativních stavů, ověřování vůči cílovým podmínkám a počátečnímu stavu atd. Vnitřní implementace prohledávacích entit však bude výrazně odlišná od PDDL reprezentace, jak ukáže následující sekce.

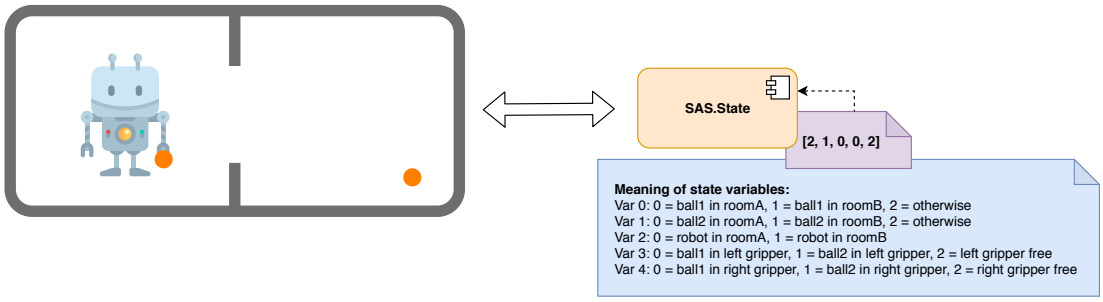
6.5.1 Základní entity problému

Jak už víme z popisu reprezentace se stavovými proměnnými z kapitoly 2.3, skutečnosti v SAS$^+$ reprezentaci budou vyjádřeny jiným způsobem, a to stavovými proměnnými.

Stavy

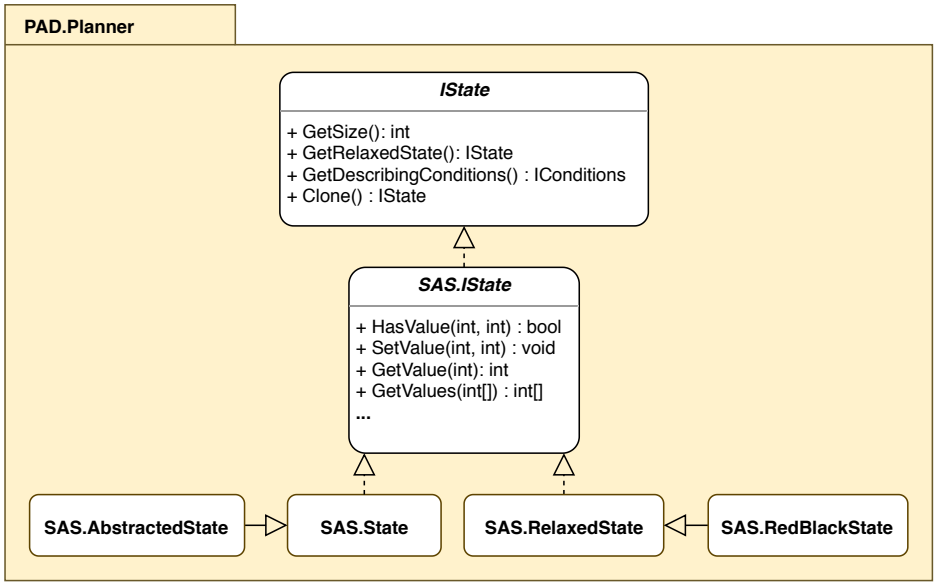
Stavových proměnných pro daný plánovací problém bude vždy pevný počet a mohou nabývat přesně specifikovaných hodnot dle definice vstupního problému.

Stav jako takový bude pouze kolekce stavových proměnných, podporující určité operace. Příklad jednoduchého SAS$^+$ stavu ukazuje obr. 6.36 níže.



Obrázek 6.36: Jednoduchý SAS$^+$ stav v doméně Gripper

Stav tedy musí podporovat práci se stavovými proměnnými, tj. zejména test na hodnotu proměnné, modifikace hodnoty proměnné, či vrácení hodnot specifikovaných proměnných. Rozhraní pro SAS$^+$ stavy ilustruje diagram na obr. 6.37.



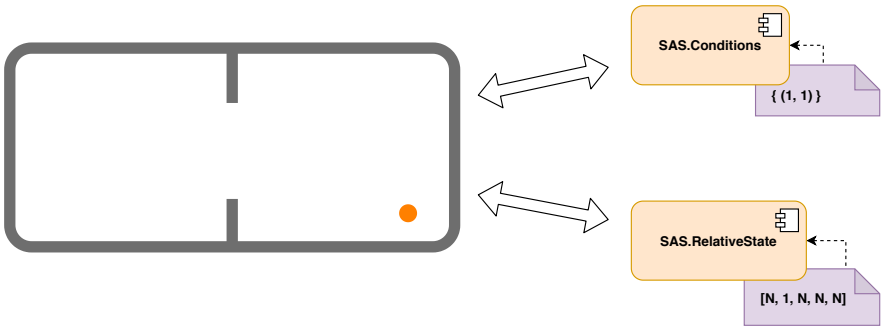
Obrázek 6.37: Hierarchie SAS$^+$ stavů

K dispozici jsou kromě standardní implementace stavu `SAS.State` i další pokročilé implementace používané pro speciální účely relaxace/abstrakce, které budou dále rozebrány.

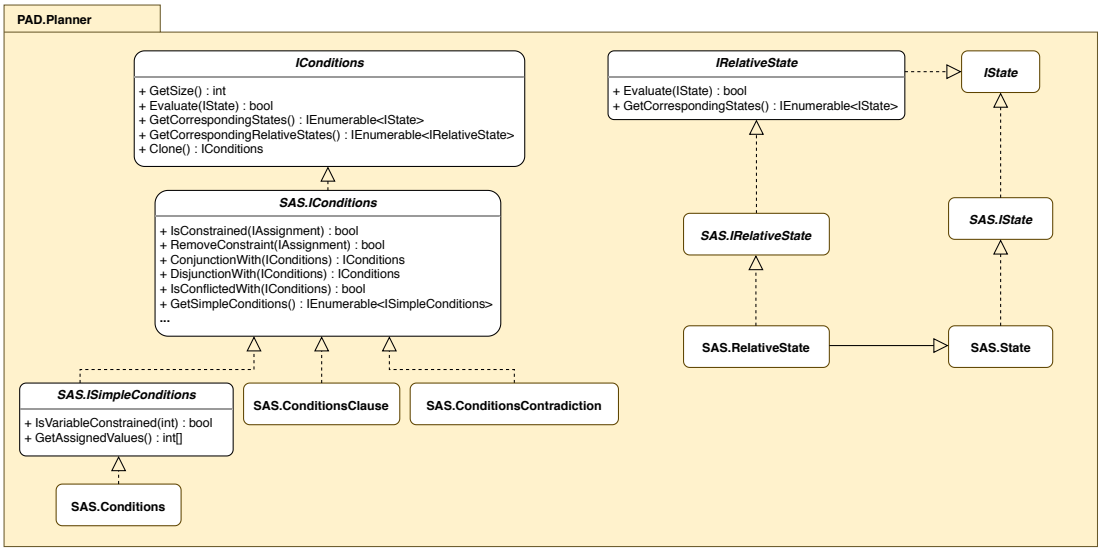
Podmínky a relativní stavy

Analogicky k PDDL reprezentaci, i v SAS$^+$ rozlišujeme k vyjádření podmínek dvě příbuzné reprezentace, tj. podmínky a relativní stavy. Obr. 6.38 níže ukazuje příklad podmínek a ekvivalentního relativního stavu.

Standardní podmínky `SAS.Conditions` jsou koncipovány jako kolekce omezení ve tvaru dvojice *(proměnná, požadovaná hodnota)*. Pro pokročilejší účely pak existují ještě komplexní podmínky `SAS.ConditionsClause` tvořící disjunkci standardních podmínek, či speciální typ podmínky `SAS.ConditionsContraditions` reprezentující spor. Hierarchii podmínek v reprezentaci SAS$^+$ ilustruje obr. 6.39.



Obrázek 6.38: Příklad podmínek a relativního stavu v SAS$^+$



Obrázek 6.39: Hierarchie podmínek a relativních stavů v SAS$^+$

Relativní stavy opět vycházejí z implementace standardních stavů, nicméně jednotlivé proměnné mohou navíc nabývat speciální nedefinované hodnoty (viz obr. 6.38), která značí, že na dané hodnotě nezáleží.

Na rozdíl od PDDL reprezentace zde platí, že standardní SAS$^+$ podmínky a relativní stavy jsou si ve vyjadřovací síle ekvivalentní. Klauzuli podmínek pak pochopitelně bude odpovídat více relativních stavů, podobně jako v případě PDDL. Sporné podmínce pak nebude odpovídat nic, jelikož jde pouze o signalizační entitu slepé větve výpočtu.

6.5.2 Proměnné

Jak jsme již zaregistrovali, SAS$^+$ pracuje už na vstupu s poměrně primitivními entitami. Počáteční stav je zadán pouze výpisem číselných hodnot, cílové podmínky pak seznamem požadovaných dvojic (proměnná, hodnota).

Program však na vstupu dostává i speciální informace o povaze jednotlivých proměnných. Lze tedy v případě potřeby (zj. při extrakci řešení do *lidsky čitelné* formy) zjistit, název dané proměnné i význam jednotlivých hodnot.

Stejně jako v PDDL narazíme na tzv. *rigidní vztahy*, tj. vztahy neměnné po dobu existence problému. Můžeme tedy o jednotlivých proměnný říct, zda jsou nebo nejsou rigidní (technicky: nefigurují v žádném z efektů operátorů, ani axi-

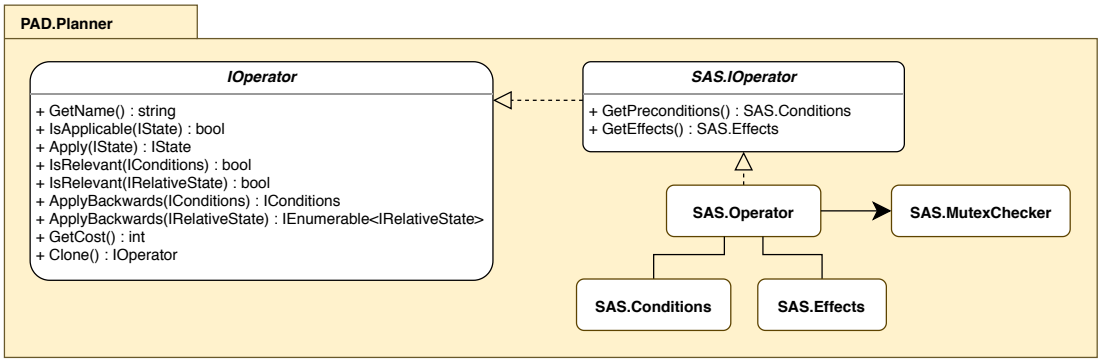
omů).

Dále se proměnné rozlišují na standardní a axiomatické. Pokud je nějaká proměnná na vstupu definovaná jako axiomatická, pak nemůže být modifikována žádným operátorem, nýbrž pouze axiomy (pro neaxiomatické přesně naopak). Pro axiomatické proměnné se rozlišují navíc tzv. axiomatické vrstvy, které ovlivňují pořadí vyhodnocování axiomy. Více viz podkapitola 6.5.4.

Všechny atributy proměnných popsané výše zprostředkovává globálně v kontextu plánovacího problému SAS$^+$ komponenta `Variables`.

6.5.3 Operátory

V reprezentaci SAS$^+$ narozdíl od PDDL neexistují liftované operátory, na vstupu dostaneme již plně groundované varianty (a místo *groundované operátory* říkáme pouze *operátory*). Konverze původního PDDL problému do SAS$^+$ typicky způsobí vytvoření velkého množství SAS$^+$ operátorů, které ale již zůstávají neměnné a v životním cyklu plánovacího problému se na ně pouze odkazuje (referencí nebo jejich číselným identifikátorem).



Obrázek 6.40: Hierarchie SAS$^+$ operátorů

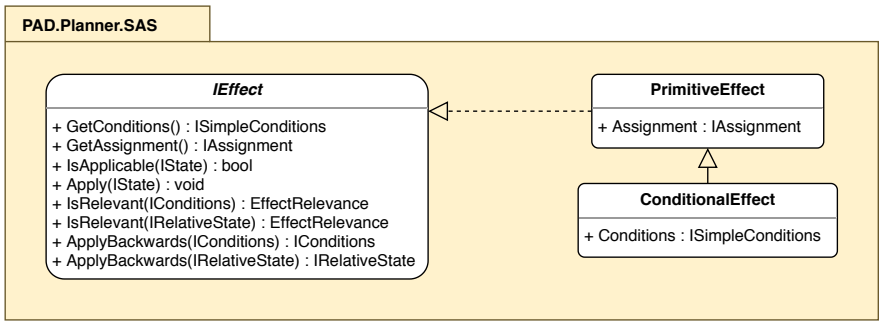
Diagram na obr. 6.40 ukazuje hierarchii SAS$^+$ operátorů – vidíme, že jednotlivé operátory si drží svoje komponenty předpokladů a efektů, na které delegují příslušné služby operátoru. Komponenta `MutexChecker` zajišťuje ověřování mutexových podmínek při aplikaci operátoru (více v části 6.5.5).

Aplikace operátoru

Aby byl operátor aplikovatelný na daný stav, musí pro něj být splněny podmínky aplikovatelnosti, tj. musí být v souladu s předpoklady operátoru.

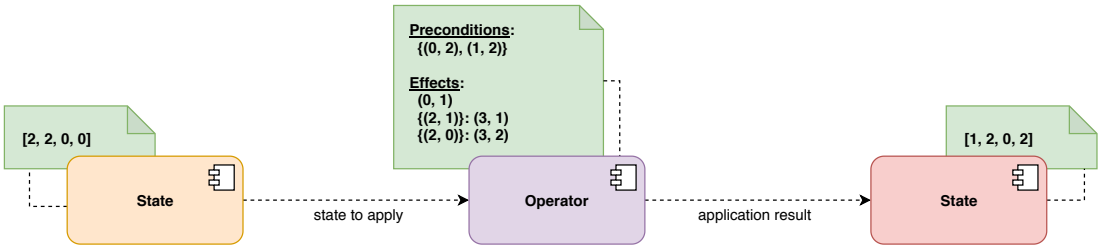
SAS$^+$ podmínky mají k dispozici funkci pro ověření toho, zda pro ně stav platí – jednoduše se lineárně prochází seznam vyžadovaných podmínek ve formě dvojic *(proměnná, požadovaná hodnota)*, vyskytující se též jako `IAssignment`, a kontroluje se, zda je tento vztah ve stavu splněn. Pokud všechna omezení v podmínce platí, pak jsou dané podmínky pro stav splněny.

Samotná aplikace operátoru je pak postupné aplikování jednotlivých efektů v operátoru. Tyto služby zajišťuje komponenta `SAS.Effects`, která je zároveň kolekcí efektů pro daný operátor. Jednotlivé efekty jsou na rozdíl od PDDL velmi jednoduché – strukturu SAS$^+$ efektů ukazuje diagram na obr. 6.41.



Obrázek 6.41: Hierarchie SAS$^+$ efektů

Základním typem efektu je tedy primitivní efekt, který definuje proměnnou, která se má v daném stavu přepsat, a novou hodnotu této proměnné. Druhým typem efektů je podmíněný efekt, který navíc definuje dodatečné podmínky pro aplikovatelnost tohoto efektu (pokud aplikovatelný není, pak se aplikace tohoto konkrétního efektu neprovede).



Obrázek 6.42: Příklad aplikace SAS$^+$ operátoru na stav

Obr. 6.42 ilustruje provedení aplikace jednoduchého SAS$^+$ operátoru na stav. Efekty daného operátoru obsahují i dva podmíněné efekty, kde pouze jeden z nich je aplikovatelný na daný stav.

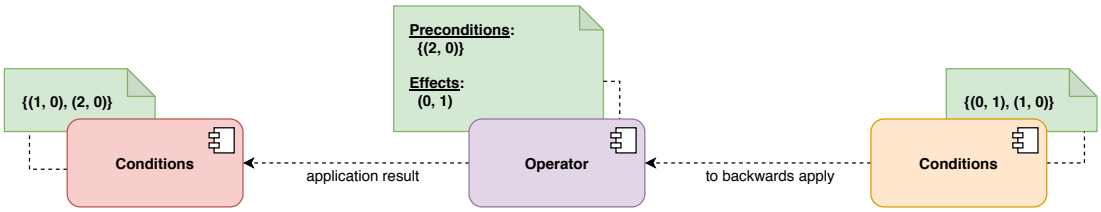
Reverzní aplikace operátoru

Aby mohl být operátor zpětně aplikovatelný na nějakou podmínku či relativní stav, je stejně jako u PDDL potřeba, aby splňoval podmínku relevantnosti.

Test na relevantnost operátoru je zcela analogický PDDL: musí platit, že operátor obsahuje efekt, který pozitivně přispívá ke splnění dané podmínky či relativního stavu, a žádný efekt nesmí být v konfliktu s nějakou hodnotou podmínek či relativního stavu. Nakonec se zkontroluje, zda operátorem neovlivněné hodnoty nejsou v konfliktu s předpoklady operátoru (tj. zpětnou aplikací by došlo ke sporu).

Zpětná aplikace operátoru na podmínky proběhne opět tak, že se z podmínky odstraní pozitivně přispívající efekty, tj. dvojice (proměnná, hodnota), které se vyskytují v efektech, a k výsledku se přidají omezení z předpokladu operátoru. Příklad zpětné aplikace na podmínky ilustruje obr. 6.43. Analogicky, pro relativní stavy odebrání nějakého omezení znamená nastavit pro odpovídající proměnnou speciální nedefinovanou hodnotu.

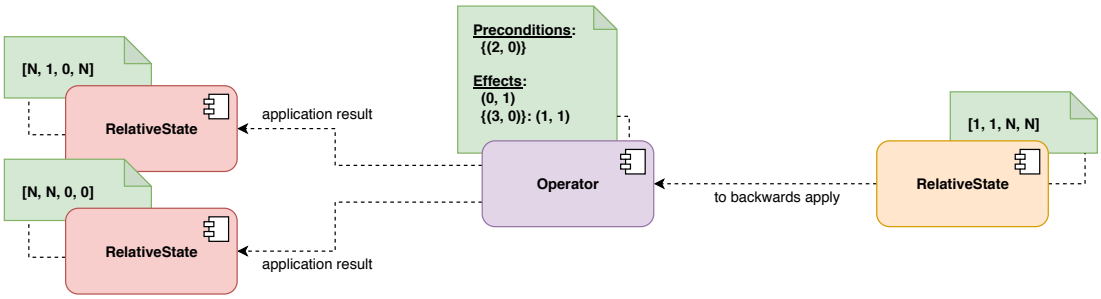
Jediným aspektem, který celou zpětnou aplikaci zkomplikuje, je přítomnost podmíněných efektů. Zpětná aplikace podmíněného efektu bude probíhat totožně



Obrázek 6.43: Příklad zpětné aplikace SAS$^+$ operátoru na podmínky

jako u efektu standardního, ale navíc se do výsledku musí přidat předpoklady tohoto podmíněného efektu.

Potíž je v tom, že každý podmíněný efekt pro daný operátor mohl a nemusel být aplikovaný. Zpětnou aplikací musíme pokrýt oba případy (a obecně, při více podmíněných efektech, všechny kombinace). Vznikne nám tak klauzule podmínek (disjunkce několika případů, v závislosti na tom, zda nějaký podmíněný efekt použit byl, anebo ne), resp. stejnou logikou vznikne několik možných relativních stavů. Příklad zpětné aplikace na relativní stav za přítomnosti podmíněných efektů ilustruje obr. 6.44.



Obrázek 6.44: Příklad zpětné aplikace SAS$^+$ operátoru na relativní stav

Z důvodů výše popsaných je zřejmé, že u problémů s podmíněnými efekty může při zpětném prohledávání docházet velmi rychle k neúměrnému zesložiťování podmínek, resp. zvyšování větvícího faktoru u relativních stavů. Proto na takové problémy nemusí být algoritmus zpětného prohledávání vhodný.

Třída klauzule podmínek `SAS.ConditionsClause` nicméně obsahuje funkce pro operace konjunkce a disjunkce s dalšími podmínkami a vnitřně se výraz automaticky zjednodušuje. Při případném zachycení logického sporu se vrátí podmínka typu `SAS.ConditionsContradiction` značící slepou větev výpočtu.

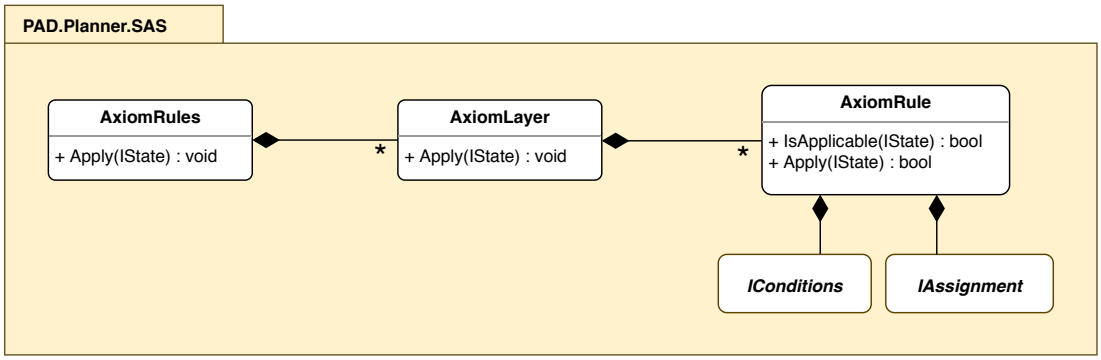
6.5.4 Axiomatická pravidla

Axiomatická pravidla (nebo zkráceně jen *axiomy*) jsou definována na vstupu a jednoduše řečeno představují automaticky odvozované vztahy v systému. Každé pravidlo v podstatě odpovídá automaticky vyhodnocovanému a aplikovanému efektu – každé má své předpoklady aplikovatelnosti a jednotlivý efekt. Stejně jako podmíněný efekt u operátorů, se axiomatické pravidlo provede pouze tehdy, když je splněn příslušný předpoklad.

Jak již bylo řečeno v části 6.5.2 o proměnných, v SAS$^+$ rozlišujeme tzv. axiomatické a neaxiomatické proměnné. První z nich mohou být modifikovány pouze

axiomatickými pravidly, druhé jmenované pak pouze operátory – oba typy proměnných však mohou vystupovat např. v předpokladech operátorů.

Strukturu axiomatických pravidel a souvisejících tříd zachycuje diagram na obr. 6.45. Axiomatická pravidla jsou sdružována do tzv. *axiomatických vrstev*, které udávají pořadí vyhodnocování a aplikací jednotlivých axiomatických pravidel. Axiomatické vrstvy jsou číslovány od nuly a každé axiomatické pravidlo je právě v jedné vrstvě. To, do které konkrétní vrstvy pravidlo patří, určuje trochu krkolomně přímo definice proměnné, kterou dané pravidlo modifikuje.



Obrázek 6.45: Struktura SAS$^+$ axiomatických pravidel

Způsob vyhodnocení axiomatických pravidel

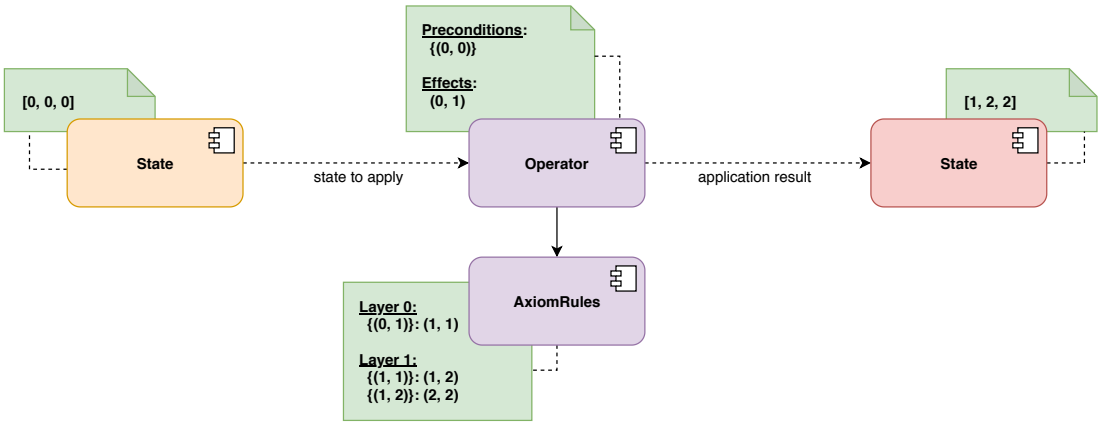
Axiomatická pravidla (pokud jsou nějaká definována) jsou v SAS$^+$ aplikována pokaždé, když se vytváří nový stav systému aplikací operátoru. Postup aplikací pravidel na daný stav je vždy následující:

1. Nastav všem axiomatickým proměnným defaultní hodnotu, která je dána hodnotou v počátečním stavu problému.
2. V axiomatické vrstvě 0 procházíme jednotlivá axiomatická pravidla – pokud je pravidlo aplikovatelné, aplikujeme ho na stav.
3. Opakujeme bod 2 tak dlouho, dokud žádné pravidlo není aplikovatelné, nebo již nijak nemění daný stav.
4. Vrátíme se na bod 2 pro axiomatickou vrstvu 1, 2, atd., dokud neprojdeme všechny přítomné vrstvy.

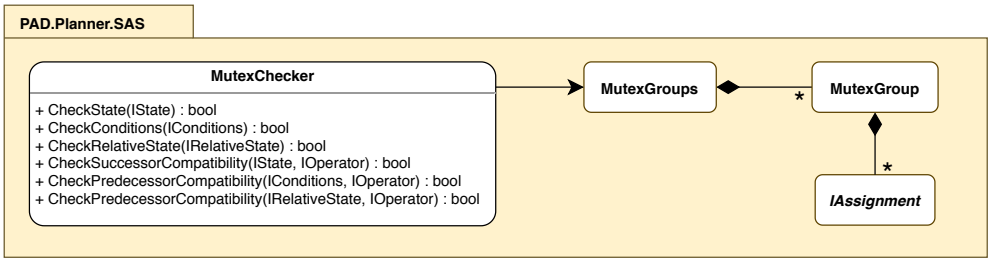
Příklad bezprostřední aplikace axiomatických pravidel po aplikaci SAS$^+$ operátoru na stav ilustruje obr. 6.46, kde má plánovací problém jednu neaxiomatickou a dvě axiomatické proměnné.

6.5.5 Mutexové skupiny

Mutexové skupiny jsou další entitou definovanou na vstupu SAS$^+$ problému a udávají dodatečné vzájemně se vylučující podmínky v SAS$^+$ problému – tj. kombinace hodnot na proměnných, které nemohou ve stejnou chvíli v SAS$^+$ stavu nastat. Implicitními mutexovými skupinami jsou hodnoty v rámci jedné proměnné (neboť jedna proměnná stavu nemůže mít zároveň více hodnot).

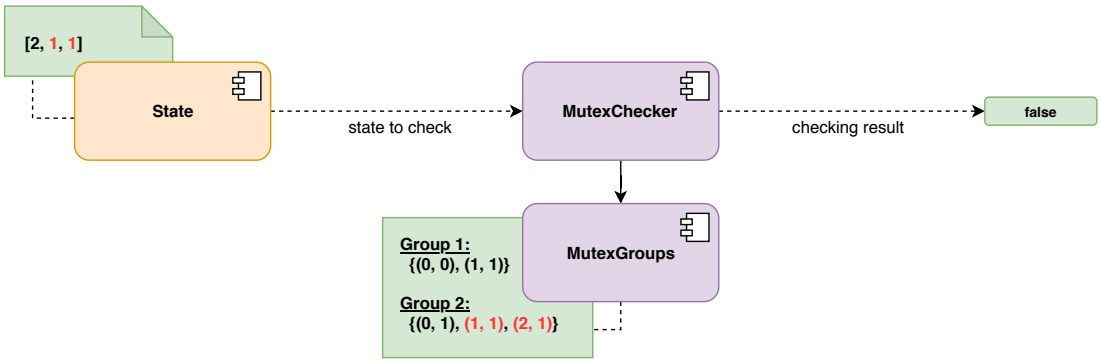


Obrázek 6.46: Příklad aplikace axiomatických pravidel po aplikaci operátoru



Obrázek 6.47: Struktura mutexových skupin SAS$^+$ a checkeru

Jedna mutexová skupina je zadána jako množina dvojic (proměnná, hodnota). Vyhodnocování mutexových skupin pak probíhá při testování aplikovatelnosti a relevantnosti operátorů jako dodatečná kontrola – pokud aplikace operátoru způsobí, že dojde k porušení nějaké mutexové skupiny, pak daný operátor nemůže být aplikovatelný, resp. relevantní.



Obrázek 6.48: Příklad vyhodnocení SAS$^+$ stavu vůči mutexovým skupinám

Popsané vyhodnocování má na starosti komponenta `MutexChecker` a probíhá postupným *zamykáním* položek v daných mutexových skupinách – pokud se pokusíme zamknout položku z již zamknuté skupiny, nahlásí se problém. Strukturu mutexových skupin a ověřovací komponenty ukazuje diagram na obr. 6.47, obr. 6.48 pak ilustruje vyhodnocení mutexových skupin vůči stavu.

6.5.6 Generování následníků a předchůdců

I v reprezentaci SAS$^+$ zajišťuje generování následníků stavů a předchůdců podmínek a relativních stavů komponenta `TransitionsGenerator`. V reprezentaci SAS$^+$ však máme všechny operátory přímo groundované a tedy rovnou připravené na testování aplikovatelnosti a relevantnosti.

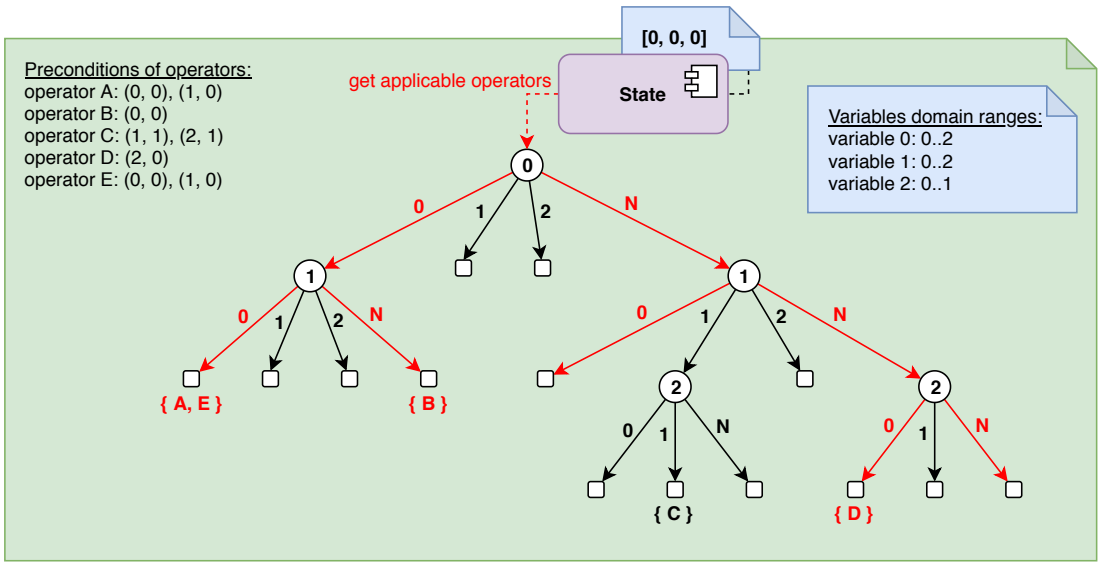
Operátory, které splňují podmínky aplikovatelnosti, jsou vráceny jako možné dopředné přechody v rámci instance `PAD.Planner.Successor`, a operátory, které splňují podmínky relevantnosti jsou vráceny jako možné zpětné přechody v rámci instance `PAD.Planner.Predecessor`. Viz struktura přechodů na obr. 6.29.

Kolekce přechodů jsou opět vráceny jako `IEnumerable`, tedy ve skutečnosti enumerátor, který generuje jednotlivé instance líně až v momentě použití.

Optimalizace výběru aplikovatelných operátorů

Vzhledem k jednoduché povaze předpokladů operátorů v reprezentaci SAS$^+$ provedeme s operátory určitý preprocessing a uložíme je do struktury tzv. *rozhodovacího stromu*, který nám výrazně zoptimalizuje následné testování aplikovatelnosti na jednotlivé stavy.

Každému vnitřnímu uzlu stromu bude odpovídat jedna rozhodovací proměnná a daný uzel bude mít takový počet podstromů, který odpovídá jednotlivým hodnotám, kterých proměnná může nabývat (a jeden navíc, pokud na hodnotě dané proměnné nezáleží). Dané podstromy budou strukturované analogicky pro další rozhodovací proměnné, až k listům stromu, které budou obsahovat jednotlivé aplikovatelné operátory. Příklad struktury stromu ilustruje obr. 6.49



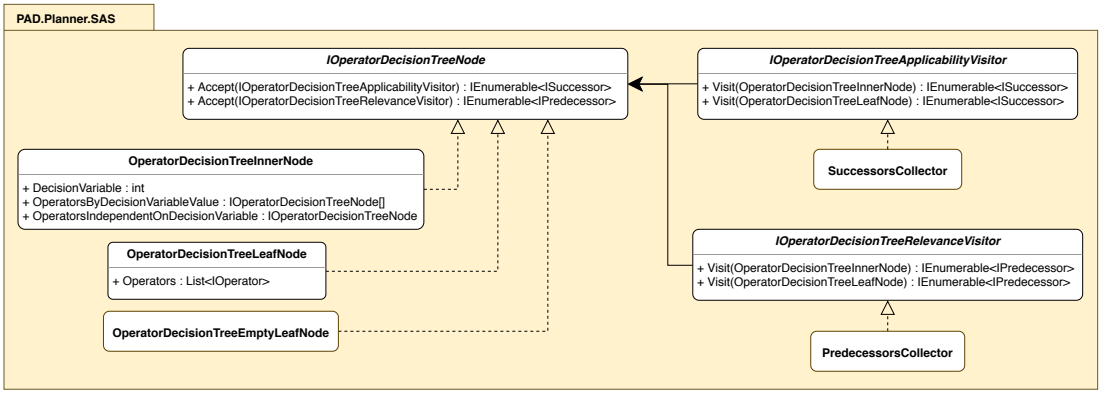
Obrázek 6.49: Příklad rozhodovacího stromu pro aplikabilitu operátorů

Při hledání aplikovatelných operátorů na daný stav tak budeme postupovat následovně: kořen stromu má určitou rozhodovací proměnnou, podíváme se na její konkrétní hodnotu v daném stavu – ta nám určí, ve kterém podstromu budeme dále hledat aplikovatelné operátory. V kořenu podstromu máme další rozhodovací proměnnou a pokračujeme stejným způsobem až do listů stromu, kde už pouze vrátíme seznam aplikovatelných operátorů, odpovídající dané cestě z kořene.

Tento postup by byl korektní, pokud by předpoklady operátorů vždy omezovaly všechny proměnné, což samozřejmě nemusí být pravda. Proto v každém uzlu stromu máme speciální podstrom pro operátory, které hodnotu rozhodovací proměnné neomezují. Najít všechny chtěné operátory tak znamená v každém uzlu rozdělit hledání do svou podstromů (toho odpovídající hodnotě rozhodovací proměnné a toho, kde na hodnotě nezáleží).

Postup získání všech aplikovatelných operátorů pro daný stav je naznačen na obr. 6.49. Pro nalezené operátory se ještě musí otestovat kompatibilita vůči mutexovým skupinám.

Strukturu tříd rozhodovacího stromu pro operátory ukazuje diagram na obr. 6.50. Postavení stromu zajišťuje komponenta `OperatorDecisionTreeBuilder` a o posbírání operátorů se stará `SuccessorsCollector`, což je jednoduchý visitor traverzující uzly rozhodovacího stromu.



Obrázek 6.50: Hierarchie uzlů rozhodovacího stromu

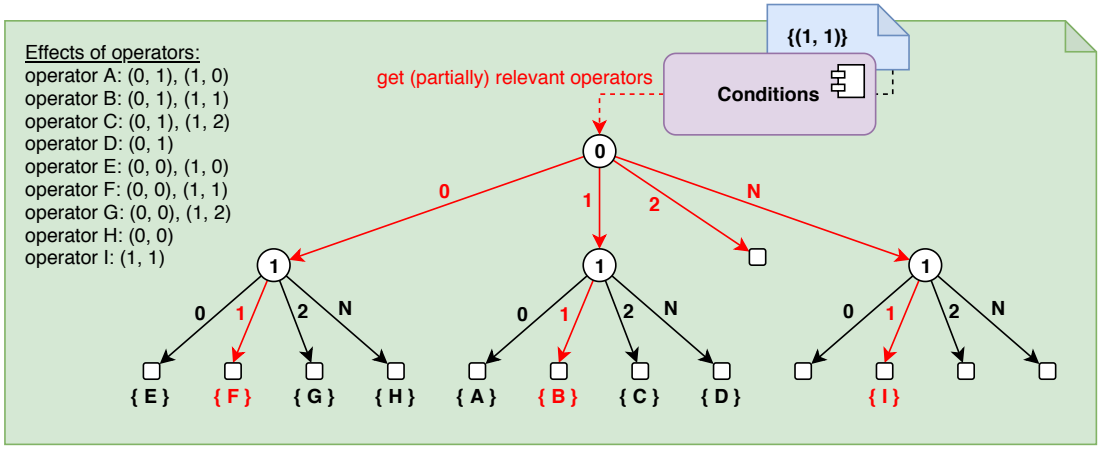
Optimalizace výběru relevantních operátorů

Analogický přístup lze použít i pro hledání relevantních operátorů pro předané podmínky (příp. relevantní stav). Rozhodovací strom bude zkonstruován na základě efektů operátorů – jednotlivé podstromy vnitřních uzlů budou odpovídat hodnotám, které pro danou rozhodovací proměnnou mohou být modifikovány v rámci efektů operátorů.

Jednoduše řečeno, rozhodovací strom bude *předvybírat* ty operátory, které pozitivně přispívají k předaným podmínkám (první podmínka relevantnosti operátoru; říkejme jim *částečně relevantní*). Další nutné podmínky (bezkonfliktnost s dalšími efekty, bezkonfliktnost s předpoklady operátoru, nebo kompatibilita s mutexovými skupinami) je pak nutné dodatečně zkontrolovat standardní metodou operátoru pro kontrolu relevantnosti. Rozhodovací strom nám však pomůže výrazně snížit počet nutných testů, protože odřeže v první fázi mnoho neperspektivních operátorů.

Příklad rozhodovacího stromu pro hledání relevantních operátorů pro dané podmínky ilustruje obr. 6.51. Znovu máme i naznačený průběh prohledávání stromu, který zabezpečuje komponenta `PredecessorsCollector`.

Pro cílové podmínky se prochází strom a pokud pro danou rozhodovací proměnnou je hodnota vázána, pak v dalším postupu hledáme opět v podstromu

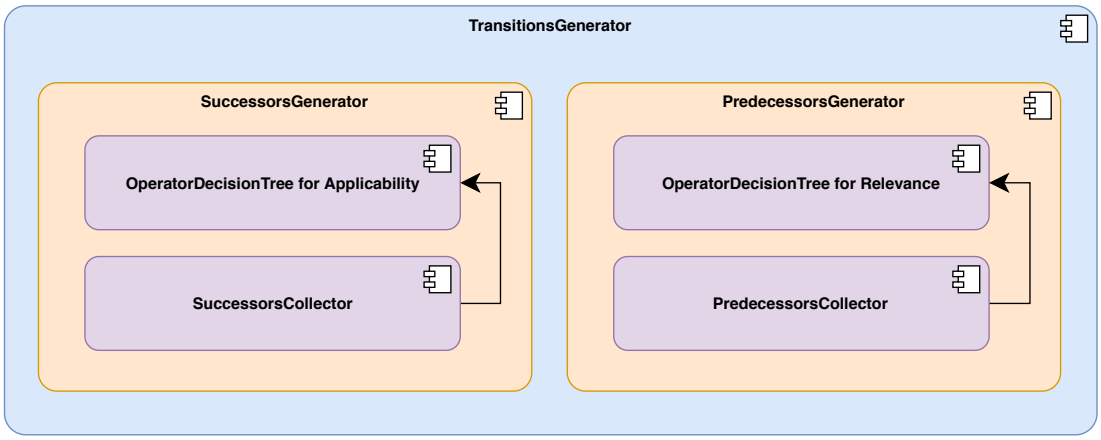


Obrázek 6.51: Příklad rozhodovacího stromu pro relevantní operátory

příslušné hodnoty. Pokud však daná rozhodovací proměnná není v podmínkách vázána, pak v dalším postupu projdeme všechny podstromy daného uzlu.

V případě komplexní podmínky ve tvaru klauzule by se stejné vyhodnocení provedlo pro všechny jednotlivé disjunkty v klauzuli a vrátilo se sjednocení těchto parciálních výsledků.

Digram na obr. 6.52 ukazuje strukturu komponenty `TransitionsGenerator`, včetně vnitřních komponent zmíněných v této podkapitole.



Obrázek 6.52: Celková struktura generátoru přechodů

6.5.7 Podpora heuristik

V části 6.4.6 jsme popsali principy heuristik implementačně závislých na reprezentaci PDDL. Spousta z těchto principů je analogicky použita i pro implementaci heuristik v reprezentaci SAS$^+$.

Například ve STRIPS heuristice si, vzhledem k výrazně jednodušším podmínkám, vystačíme s operátory konjunkce a disjunkce (u `ConditionsClause`) a na základě stejných pravidel vyhodnotíme počty nesplněných cílů, což nám určí výslednou heuristickou hodnotu.

Také heuristiky založené na relaxaci jsou téměř stejné, pouze fakty v prepozičních vrstvách relaxovaného plánovacího grafu nejsou vyjádřeny pomocí predikátů,

nýbrž pomocí přiřazení ve tvaru (proměnná, hodnota). Způsob vyhodnocení na základě popsaných strategií v části 6.4.6 zůstává stejný.

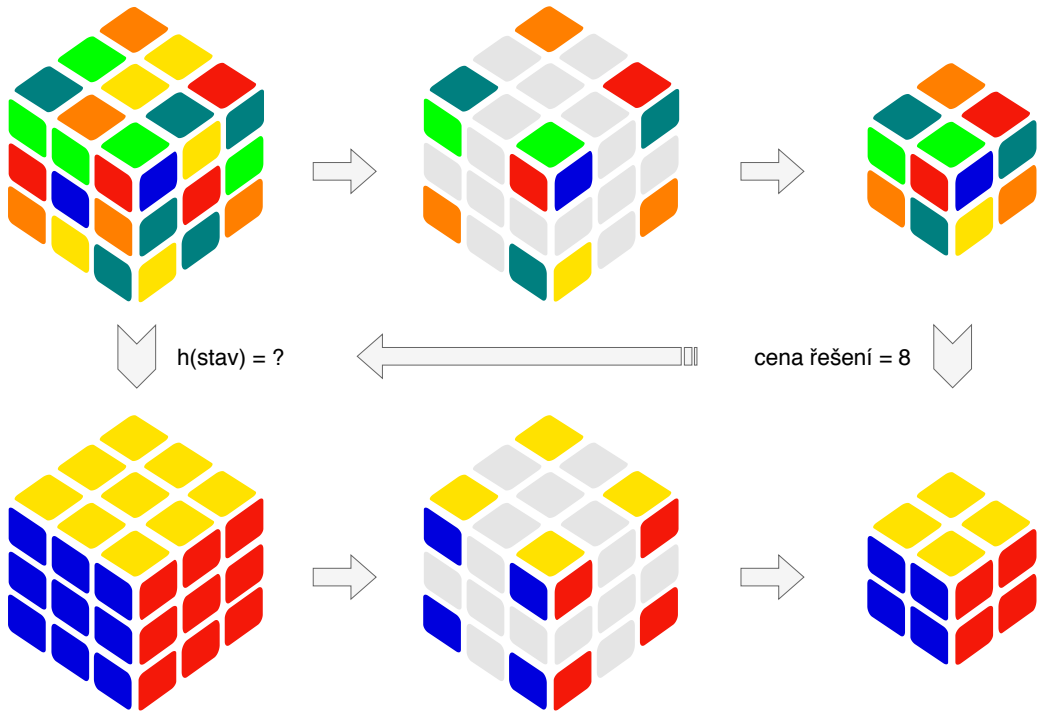
PDB Heuristika

V heuristickém plánování slaví úspěch ještě jiný pokročilý přístup, založený na tzv. *abstrakci*. Známým zástupcem heuristiky založené na abstrakci je tzv. *databáze vzorů*[35][36] (angl. *Pattern Database*, a odsud *PDB heuristika*).

I abstrakce zjednodušuje původní plánovací problém pro získání odhadu vzdálenosti v problému původním. *Vzor* v kontextu PDB znamená vybranou podmnožinu proměnných, které v pozměněném plánovacím problému bereme jako relevantní – od zbylých proměnných se oprostíme a vyřešíme zjednodušený problém, který nám udá hodnotu heuristiky.

Příklad tohoto postupu ilustruje obr. 6.53 pro problém Rubikovy kostky – odhadneme vzdálenost do cíle (tj. složené kostky) tím, že budeme řešit stejný problém pouze pro rohy kostky.

Přitom je obecně možné najednou použít více různých vzorů, vypočítat na základě nich heuristické hodnoty, které se následně sečtou (hovoříme o aditivních vzorech) a vrátí se jedna výsledná hodnota heuristiky.

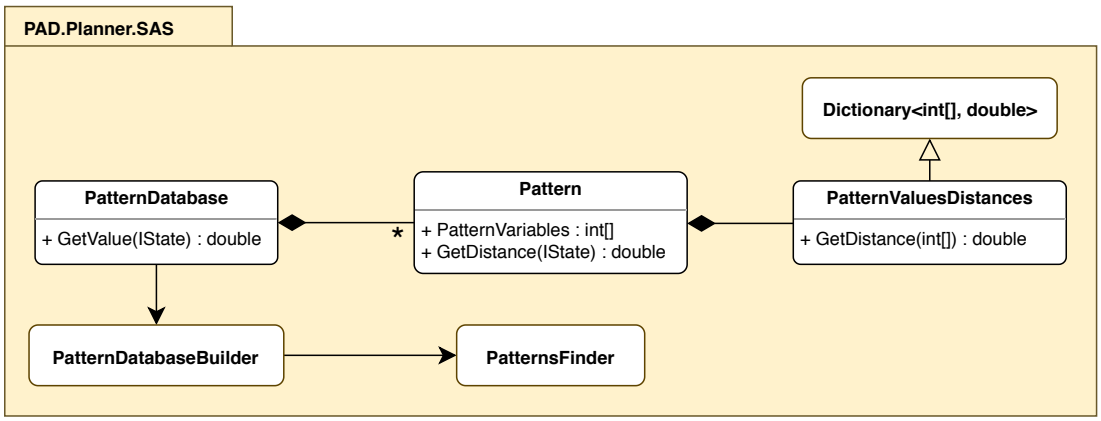


Obrázek 6.53: Příklad využití abstrakce pro dolní odhad vzdálenosti k cíli pro problém Rubikovy kostky (heuristická hodnota původního stavu bude 8)

Databáze vzorů pak funguje tak, že si předpočítá vzdálenosti do cíle pro jednotlivé vzory a jejich příslušné hodnoty. Vytvoření databáze sice může zabrat určitý čas (a paměť), ale získávání heuristických hodnot pak již bude probíhat v konstantním čase.

Strukturu databáze vzorů ve frameworku PAD ukazuje diagram na obr. 6.54. Třída `PatternDatabaseBuilder` vytváří databázi na základě zvolených vzorů.

Postupuje přitom zpětným prohledáváním od cílových podmínek na abstrahovaném problému a zaznamenáváním minimálních vzdáleností pro jednotlivé hodnoty vzoru.



Obrázek 6.54: Struktura databáze patternů ve frameworku PAD

Důležitým problémem je nalézt vhodné vzory tak, aby se příslušná databáze vešla do paměti a heuristika byla co nejinformativnější. Databáze vzorů v PAD umožňuje jak manuální specifikaci vzorů, tak automatizované nalezení aditivních vzorů (komponenta `PatternsFinder`) na základě *blízkosti* proměnných.

Pro automatické nalezení takových vzorů se provede analýza grafu, kde vrcholy odpovídají jednotlivým proměnným a zároveň jsou vždy dva vrcholy spojeny hranou, pokud existuje v plánovacím problému operátor, který mění obě odpovídající proměnné najednou. Nalezenými vzory jsou pak jednotlivé komponenty souvislosti v grafu, okleštěné o vzory, jejichž proměnné nemají průnik s cílovými podmínkami.

7. Validace a verifikace PADu

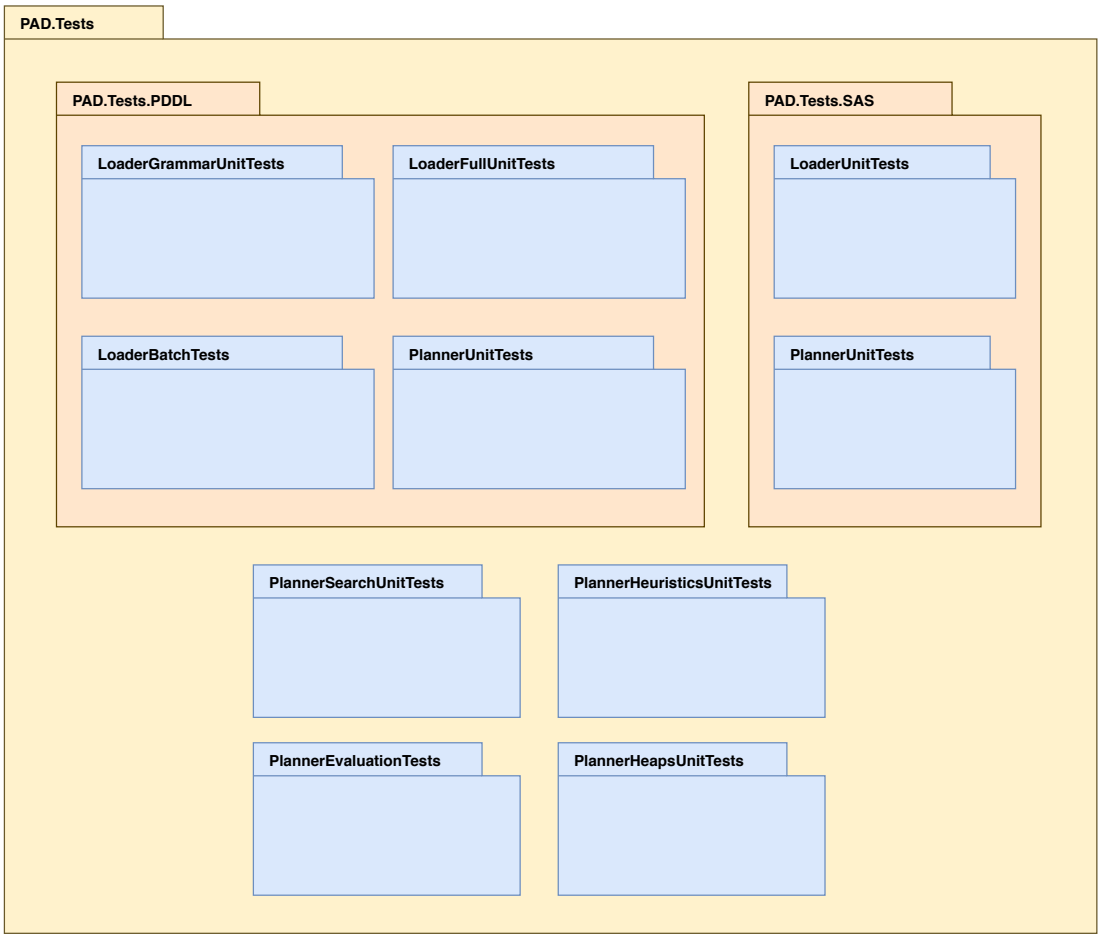
Nezbytnou součástí vývoje jakéhokoli významnějšího softwarového produktu je zajištění *kvality*, tj. dosažení určitých vlastností související s jeho schopností plnit požadované/předpokládané potřeby. V rámci softwarového inženýrství se zajišťováním kvality v rámci vývojového cyklu zabývá oblast tzv. *Quality assurance*.

Významnou součástí zajištění kvality je validace a verifikace, což jsou obecně řečeno aktivity cílené na ověření, zda určité části programu splňují nároky na ně kladené. Jednou z forem validace a verifikace je pak testování.

Testování může mít obecně mnoho forem a přístupů. Pro projekt typu PAD, který může být používán mnoha různými lidmi a mnoha různými způsoby, se jeví nejvýhodnější přístupem mít přiloženou sadu unit testů, která by ověřovala korektnost jednotlivých částí programu.

7.1 Projekt `PAD.Tests`

Již v během konstrukce frameworku PAD probíhal simultánní vývoj unit testů v rámci testovacího projektu `PAD.Tests`. Ten je integrální součástí celkového solutionu a ověřuje funkčnost a korektnost všech klíčových komponent programu.



Obrázek 7.1: Struktura testů v projektu `PAD.Tests`

Unit testy byly konstruovány za pomocí vývojového nástroje *Visual Studio Unit Testing Framework*, který je integrovanou součástí vývojového prostředí *Microsoft Visual Studio 2019*.

Obr. 7.1 ukazuje jednotlivé balíčky testů v rámci projektu `PAD.Tests`. Jak jde vidět, balíčky unit testů by se daly rozdělit do tří hlavních kategorií: PDDL specifické, SAS$^+$ specifické a testy společných komponent.

Některé z testovacích případů jsou koncipovány z tzv. *black-box* perspektivy, což znamená, že testujeme komponentu zvenku bez znalosti vnitřní implementace, jiné zase z *white-box* perspektivy, kde testujeme komponenty se znalostí vnitřní implementace. Celkem bylo implementováno 227 testovacích případů.

Popišme si stručně obsah jednotlivých balíčků unit-testů:

- `PDDL.LoaderGrammarUnitTests` – obsahuje white-box testy interních součástí PDDL loaderu vstupních dat, konkrétně testy na jednotlivé větve gramatiky a ověřování korektního naparsování a převedení do AST stromu,
- `PDDL.LoaderFullUnitTests` – obsahuje black-box testy procesu načítání vstupních dat loaderem a následného procesu validace; testy jsou zacílené na ověření načítání jednotlivých funkčních bloků ze vstupních PDDL souborů,
- `PDDL.LoaderBatchTests` – obsahuje dávkové testy z black-box prespektivy na PDDL loader vstupních dat; jako testovací vstupy jsou použity oficiální problémy z Mezinárodní plánovací soutěže (IPC),
- `PDDL.PlannerUnitTests` – obsahuje white-box i black-box testy všech součástí plánovače, které jsou specifické pro PDDL (zj. jde o komponenty plánovacího problému),
- `SAS.LoaderUnitTests` – obsahuje black-box testy celkového načítání vstupních SAS$^+$ dat loaderem a procesu validace; testy jsou cílené na ověření načítání jednotlivých sekcí ze vstupních SAS$^+$ souborů,
- `SAS.PlannerUnitTests` – obsahuje testy z white-box i black-box perspektivy na všechny součásti plánovače, které jsou specifické pro formát SAS$^+$ (zj. komponenty plánovacího problému),
- `PlannerSearchUnitTests` – obsahuje testy jednotlivých implementací algoritmů heuristického prohledávání a pomocných struktur z black-box perspektivy,
- `PlannerHeuristicsUnitTests` – obsahuje black-box testy všech implementovaných heuristik používaných pro heuristické prohledávání ve frameworku PAD a také testy statistik využití,
- `PlannerHeapsUnitTests` – obsahuje black-box testy všech implementací hald, coby datových kolekcí pro prohledávací uzly, které jsou dostupné pro algoritmy heuristického prohledávání,
- `PlannerEvaluationTests` – obsahuje souhrnné evaluační testy využívající jednotlivé plánovací úlohy z Mezinárodní plánovací soutěže, zadané v obou vstupních formalismech.

Testy se pouští přímo ve Visual Studiu a celková doba běhu všech testů obvykle nepřekročí 1 minutu na běžné sestavě. Unit testy je dobré pouštět zejména

při větších změnách či provádění refactoringu. Při implementaci nových komponent je nutné doplnit unit testy těchto nových komponent v projektu `PAD.Tests`.

7.2 Zvyšování kvality a budoucí vývoj

Vzhledem k charakteru vyvíjeného softwaru se očekává další (potenciálně rozsáhlý) vývoj a rozšiřování funkcionality. S tímto faktorem se při vývoji počítalo, a i proto obsahuje framework PAD rozsáhlou kolekci unit testů, která jakékoli rozšiřování a optimalizování softwaru značně usnadňuje a zvyšuje míru důvěry v konečný produkt. Nevýhodou je nutnost udržovat tyto testy a doplňovat testy na nově vzniklé komponenty, v opačném případě testy postupně pozbudou významu.

Kvalitní návrh a kód

K dosažení kvalitního návrhu se při vývoji frameworku PAD široce využívaly návrhové vzory a snaha byla o maximální naplnění myšlenky *programovat vůči rozhraní a nikoli implementaci* (viz *Liskovové pricip zastoupení* zmíněný v části 4.3), čímž se snižuje míra provázanosti a dosahuje vysoké míry modulárnosti a rozšiřitelnosti softwaru.

Pro kvalitní a konzistentní kód, který bude pochopitelný pro všechny budoucí uživatele frameworku, je nutné se držet určitých konvencí – zde dává smysl použít oficiální kódové konvence od Microsoftu pro doménu .NET[15]. Již zmíněným, dobrým zdrojem informací pro dosažení kvalitního kódu je kniha *Dokonalý kód*[16]. Občas může být nutné provést na starším kódu refactoring, což je bezpečná operace pouze tehdy, když máme jednotlivé komponenty dobře pokryty unit testy (což pro framework PAD jistě platí).

Pro odchytávání častých chyb se používá ještě tzv. *statická analýza kódu*, která umí v době překladu automaticky nalézt různé podezřelé konstrukce a špatné programové návyky. Na rozdíl od překladače, který kontroluje syntax, se statická analýza kódu zaměřuje na sémantiku programu; obecně umí nalézt spíše jednoduché chyby, ale i tak je silně doporučována. Reprezentanty takových analyzátorů v doméně .NET jsou například *ReSharper*[40] nebo *Microsoft Code Analysis 2019*[41] – oba byly ve frameworku PAD použity.

Výkonnost programu

Pro zvyšování výkonnosti programu se používají tzv. *profilery*, což jsou analyzátory umožňující nalézt *úzká hrdla* výkonnosti v daném programu a zlepšovat efektivitu jednotlivých komponent. Opět je možné použít integrovaný profiler Microsoft Visual Studia, který umožňuje diagnostikovat využití CPU, GPU nebo paměti a přitáhnout pozornost vývojáře k problémovým místům.

Jsou však výkonnostní hlediska, kde ani profiler nepomůže a může jít jednoduše o nedokonalý návrh. Framework PAD si nedělá ambice být od začátku dokonalým plánovačem, vývoj se primárně soustředil na vytvoření kvalitního a robustního základu, na který by navazoval další intenzivní vývoj. Implementace mnoha komponent je ovlivněna snahou o maximální obecnost (např. nabídnout

uživateli vše, co PDDL teoreticky umožňuje), což se občas může projevit horší efektivností takových komponent.

Idea bezprostředně navazujícího vývoje je doimplementovat do systému určité adaptabilní prvky: např. PDDL stavy nebo podmínky v sobě mohou nést informace o predikátech, objektových a numerických fluentech, ve většině případů však budeme pracovat pouze s predikáty, a proto by mohl být dobrý nápad v takových případech použít speciální implementaci stavů optimalizovanou pouze pro práci s predikáty.

Dále lze zkonstruovat nové efektivnější implementace některých entit v programu na základě toho, že jich bude omezené množství nebo budou mít omezenou doménu, a výrazně tak ovlivnit paměťovou stopu těchto entit. Myšlenka zmíněné adaptability je tedy ta, že se zanalyzuje vstupní problém a na základě toho se použijí maximálně vhodné implementace některých primitiv pro optimální průběh výpočtu.

8. Použití PADu

V následující kapitole se podíváme na to, jakými způsoby se dá framework PAD používat. Popíšeme projekt `PAD.Launcher`, umožňující automatizované provádění hromadných paralelních výpočtů, nebo si ukážeme, jakým způsobem lze vytvořit zcela novou reprezentaci plánovacího problému ve frameworku PAD.

8.1 Přímé využití komponent frameworku

Jak ukazoval diagram na obr. 4.1, celé řešení frameworku PAD obsahuje několik logicky oddělených projektů. Vyobrazené závislosti naznačují, že některé projekty lze využít izolovaně, nezávisle na ostatních komponentách frameworku.

Pokud by např. existoval externí uživatel, který potřebuje pouze nástroj pro načítání a zpracování vstupních dat reprezentací PDDL nebo SAS$^+$, pak si může bez problémů zahrnout do svého řešení pouze projekt `PAD.InputData`, zapouzdřující veškerou potřebnou funkcionalitu.

Příklad použití ilustruje ukázka zdrojového kódu 8.1 níže. Volitelným parametrem konstruktoru lze navíc vypnout validaci vstupních dat (standardně zapnuto).

```
var dataPDDL = new InputData.PDDLInputData("d.pddl", "p.pddl");
var dataSAS = new InputData.SASInputData("problem.sas");
```

Zdrojový kód 8.1: Příklad samostatného načtení vstupních dat

Pokud by chtěl jiný uživatel přímo využít služeb plánovače, musel by si do svého řešení zahrnou jak projekt `PAD.Planner`, tak i `PAD.InputData`, protože zde již existuje jistá závislost plánovacích komponent na vstupních datech.

Samotné spuštění výpočtu plánovače nad zvoleným plánovacím problémem je velmi přímočaré, jak ilustruje zdrojový kód 8.2 níže. Prohledávacím rutinám lze dle potřeby nastavit i časový a paměťový limit, nebo možnosti logování výpočtu.

Samotná funkce `Start()` spustí výpočet, ale následně vrací pouze informaci, jak výpočet proběhl. Uživatel si pak dle své potřeby může vyzvednout např. pouze cenu řešení, pouze plán řešení, stručné či kompletní statistiky výpočtu.

```
var problem = new Planner.SAS.Problem("problem.sas");
var heur = new Planner.Heuristics.FFHeuristic(problem);
var heap = new Planner.Heaps.BinomialHeap();

var search = new Planner.Search.AStarSearch(problem, heur, heap);
if (search.Start() == Planner.Search.ResultStatus.SolutionFound)
{
    Console.WriteLine(search.GetSolutionPlan().ToString());
}
```

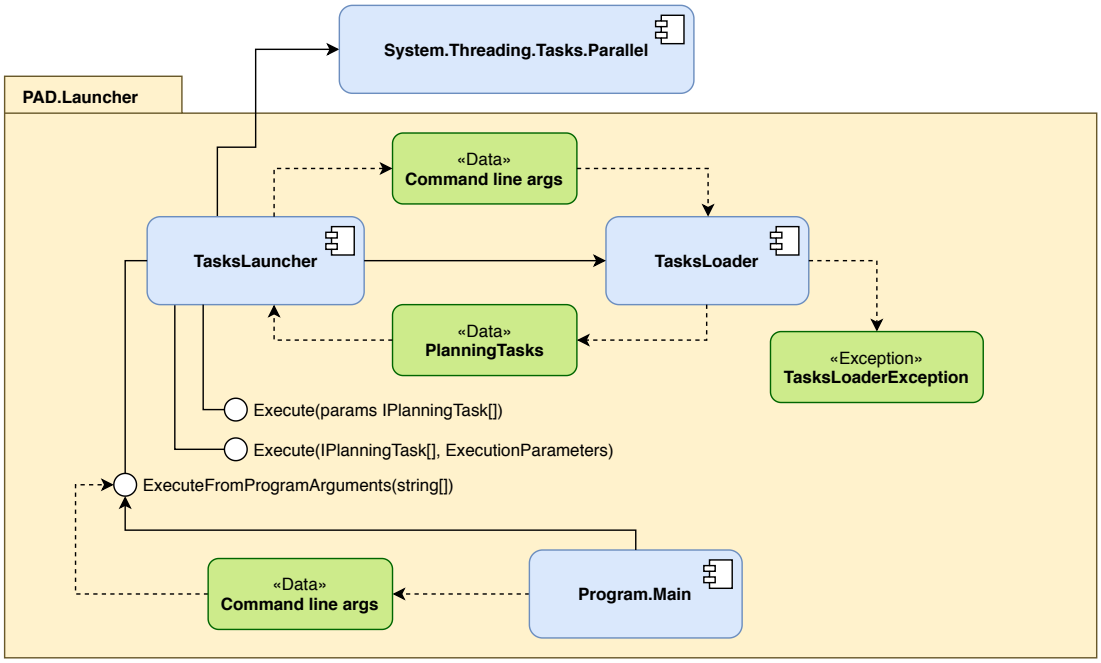
Zdrojový kód 8.2: Příklad nastavení a spuštění plánovače

8.2 Dávkové výpočty – `PAD.Launcher`

Jedním z požadavků zadavatele je umožnit realizaci experimentálních hromadných výpočtů nad mnoha plánovacími problémy, včetně zajištění možnosti pro automatické zpracování či vyhodnocení výsledků prohledávání.

K tomuto účelu byl vytvořen projekt `PAD.Launcher`, poskytující nástroje pro definici jednotlivých i hromadných plánovacích úloh (*tasků*), které následně umožňuje nechat automatizovaně zpracovat. Podporováno je paralelní zpracování těchto úloh, které umožňuje efektivní vytížení prostředků počítače a zrychluje celý proces dávkového výpočtu.

Diagram na obr. 8.1 ukazuje vnitřní strukturu projektu `PAD.Launcher`. Standardně je tento projekt v rámci frameworku PAD nastaven jako *start-up* projekt, tudíž po přeložení je volána jeho metoda `Program.Main`, která přímo deleguje zpracování vstupních parametrů programu na komponentu `TasksLauncher`.



Obrázek 8.1: Komponenty projektu `PAD.Launcher`

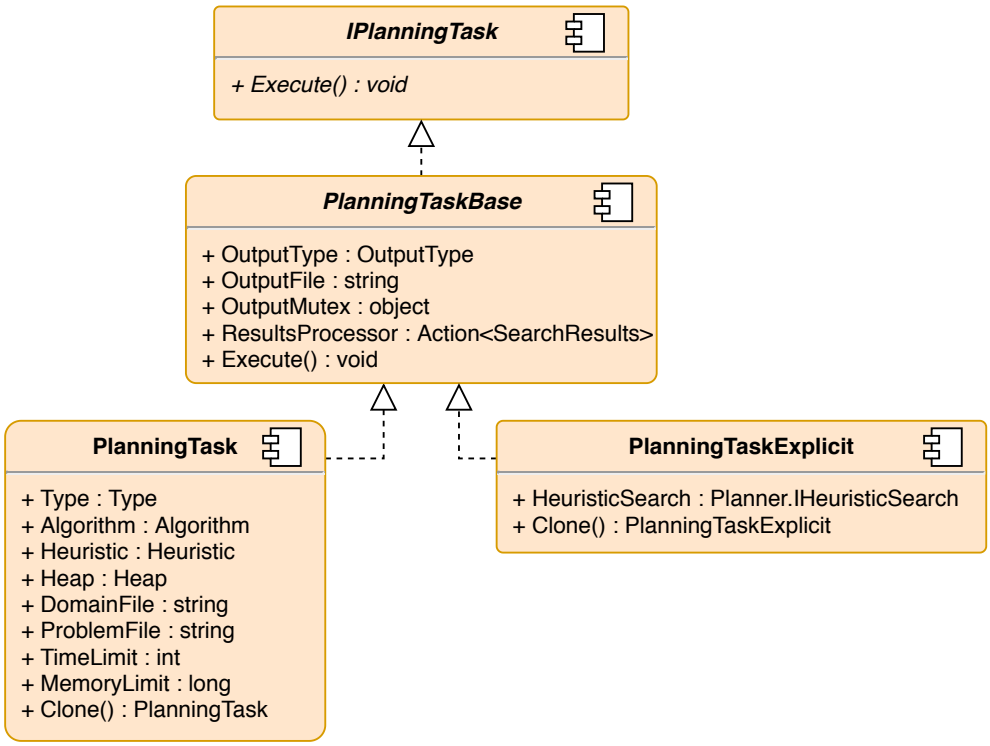
Komponenta `TasksLauncher` si následně nechá na základě vstupních parametrů připravit jednotlivé úlohy pro výpočet od komponenty `TasksLoader` a nechá je paralelně spočítat. Alternativně si lze jednotlivé úlohy připravit programově a přímo je poslat komponentě `TasksLauncher`.

`TasksLauncher` k paralelní exekuci úloh využívá .NET knihovny TPL (*Task Parallel Library*), které si samy řídí spouštění jednotlivých úloh a zajišťují *load-balancing* výpočetních prostředků, na základě nastavené maximální míry paralelismu.

Plánovací úlohy

Strukturu plánovacích úloh pro výpočet ukazuje diagram na obr. 8.2. Objekt `PlanningTask` parametricky specifikuje typ vstupní reprezentace, použitý algoritmus, použitou heuristiku, použitou haldu, vstupní soubory, typ výstupu (konzole,

soubor nebo zpracování uživatelskou funkcí), příp. výstupní soubor, mutexový objekt pro ošetření souběžného zápisu více úloh a také časový a paměťový limit výpočtu úlohy. Pokud nejsou vytvořené úloze nastaveny všechny parametry, jsou použity defaultní hodnoty. Nezbytně nutné údaje pro vyplnění jsou typ vstupní reprezentace a cesty ke vstupním souborům.



Obrázek 8.2: Hierarchie plánovacích úloh

Pro spuštění úlohy na základě nastavených parametrů pak již jen stačí zavolat její metodu `Execute()`. Úloha si sama načte data, vytvoří odpovídající objekty a spustí prohledávací rutinu.

Alternativně si lze úlohu vytvořit prostřednictvím `PlanningTaskExplicit`, které stačí předat již připravenou prohledávací rutinu (včetně nastaveného problému, heuristiky apod.) a příp. výstupní parametry. Toto dává uživateli frameworku větší míru flexibility při vytváření úloh.

Specifikace úloh pomocí konfiguračního souboru

Standardní objekty `PlanningTask` je možné vytvářet na základě vstupních parametrů programu. Pokud je specifikován pouze jeden vstupní parametr, předpokládáme, že jde o konfigurační soubor výpočtu. Příklad konfiguračního souboru ukazuje kód 8.3.

Vidíme, že konfigurační soubor umožňuje nastavit globálně hodnotu maximálního počtu paralelně zpracovávaných úloh (standardně zvoleno automaticky dle počtu jader uživatelského systému) a následně parametry jednotlivých úloh. Definice každé úlohy musí začínat řádkem ve tvaru `#jméno_úlohy` a následuje seznam nastavení parametrů, které odpovídají atributům plánovací úlohy na obr. 8.2.

V příkladu konfiguračního souboru 8.3 nastavuje úloha 1 explicitně všechny parametry plánovací úlohy, definice dalších dvou úloh pak nechávají většinu pa-

rametrů na defaultních hodnotách. Zatímco první úloha je definována pro jeden problém, druhá už se spustí na dvou různých problémech a třetí na celé složce problémů (komponenta `TasksLoader` vytvoří jednotlivé plánovací úlohy pro každý problém z této složky).

```
MaxNumberOfParallelTasks=8

# Task 1
Type=PDDL
Algorithm=AStarSearch
Heuristic=FFHeuristic
Heap=BinomialHeap
Domain=domain.pddl
Problems=problem.pddl
OutputType=ToFile
OutputFile=results.txt
TimeLimit=60
MemoryLimit=50000

# Task 2
Type=SAS+
Algorithm=HillClimbingSearch
Problems=problem1.sas;problem2.sas

# Task 3
Type=SAS+
Problems=Problems/SAS/
```

Zdrojový kód 8.3: Příklad konfiguračního souboru pro výpočet

Alternativně lze spustit program bez konfiguračního souboru tak, že jednotlivá nastavení jsou přímo vstupními argumenty programu.

Generování jednotlivých plánovacích úloh na základě vstupních parametrů nebo konfiguračního souboru zajišťuje již zmíněná komponenta `TasksLoader`. Pokud se v rámci načítání objeví chyba, program se ukončí a zahlásí se chybová hláška. Dokumentace obsahuje plný popis akceptovatelného formátu konfiguračních souborů pro výpočet.

Definování úloh pro výpočet pomocí konfiguračního souboru usnadňuje použití plánovače např. v rámci externích skriptů. Nevýhodou ovšem je, že pro použití nově vytvořené reprezentace problému, heuristiky, algoritmu či haldy je nutné rozšířit vstupní typy a napojit je na nové reprezentace (což sice není operace nijak složitá ani zdlouhavá, avšak lehce nepohodlná).

Specifikace úloh programově

Vytvoření a spuštění plánovacích úloh programově je velmi přímočaré, jak ukazuje příklad kódu 8.4 – standardní i explicitní úlohu připravíme a pouze předáme komponentě `TasksLauncher`, která provede výpočet.

```
var task = new PlanningTask
{
    Type = Type.SAS,
    Heuristic = Heuristic.FFHeuristic,
    ProblemFile = "elevators.sas"
};

var task2 = new PlanningTaskExplicit
{
    HeuristicSearch = GetMyHeuristicSearch()
};

TasksLauncher.Execute(task, task2);
```

Zdrojový kód 8.4: Příklad programového vytvoření a spuštění úloh

Při programovém vytváření lze úloze navíc přiřadit libovolnou funkci pro zpracování výsledků úlohy – stačí parametru `ResultsProcessor` nastavit akci, která má za argument objekt `SearchResults`, tj. kompletní výsledky výpočtu. Výsledky pak nejsou vypsány do souboru, ani konzole, ale pouze se provede specifikovaná funkce. Tento mechanismus dává uživateli maximální flexibilitu pro post-processing výsledků.

Jednoduché využití této možnosti ilustruje příklad 8.5 níže – do konzole je po skončení výpočtu vypsán čas výpočtu. Lze však velmi snadno realizovat agregování výsledků apod. Stačí použít některou z thread-safe kolekcí a vybrané výsledky sbírat, filtrovat či nad nimi počítat pokročilejší statistiky.

```
Action<SearchResults> processor = (SearchResults results) =>
{
    Console.WriteLine(results.SearchTime);
};

PlanningTask task = new PlanningTask
{
    Type = Type.SAS,
    ProblemFile = "elevators.sas",
    ResultsProcessor = processor
};

TasksLauncher.Execute(task);
```

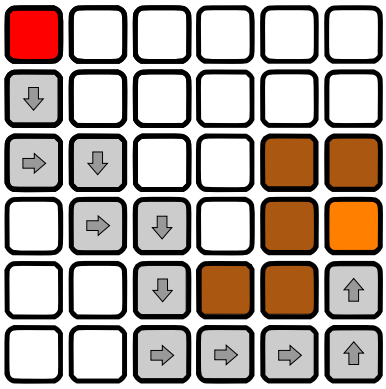
Zdrojový kód 8.5: Příklad úlohy s definovaným procesorem výsledků

Definici funkce nelze v tuto chvíli použít při spouštění pomocí konfiguračního souboru. V případě potřeby se však velmi snadno dá rozšířit `TasksLauncher` o kolekci často používaných funkcí a ty pak pomocí konfiguračního souboru specifikovat typovým parametrem (podobně jako je např. realizována volba algoritmu).

8.3 Vytvoření nové reprezentace problému

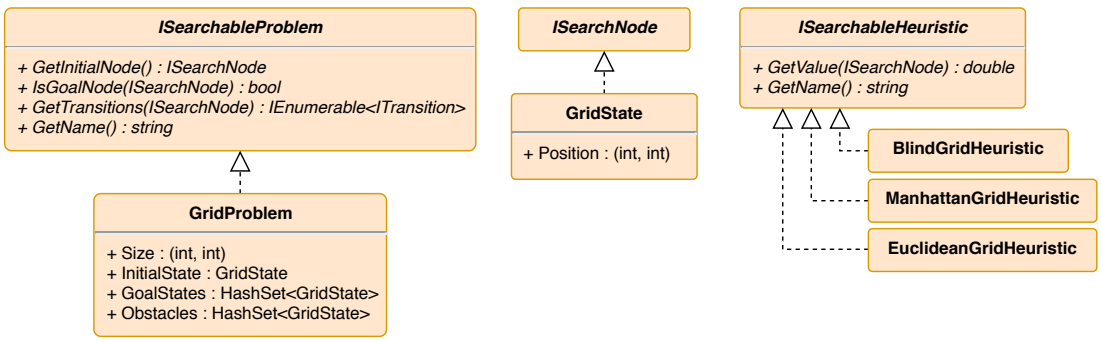
Jedním z klíčových požadavků pro framework PAD je podpora rozšiřitelnosti. V této části si předvedeme konkrétní případ vytvoření minimalistické, nové reprezentace, která efektivně využije jádrové prvky frameworku.

Předpokládejme, že před námi stojí následující problém: máme zadanou čtvercovou mřížku konečné velikosti (na které se můžeme pohybovat pouze horizontálně a vertikálně), dále máme startovní bod na této mřížce, množinu cílových bodů a množinu bodů, které jsou neprostupnými překážkami. Cílem je nalézt nejkratší cestu ze startovního bodu do některého z cílových bodů. Příklad popsaného problému ilustruje obr. 8.3.



Obrázek 8.3: Příklad problému hledání na čtvercové mřížce

Problém se dá pojmout a realizovat mnoha způsoby, my však předvedeme nejjednodušší funkční variantu, dokonce bez použití operátorů. Z kapitoly 6.2 již víme, že prohledávací algoritmy frameworku pracují s abstrakcí plánovacího problému a abstraktními prohledávacími uzly. Od těchto entit tedy vytvoříme nové implementace.



Obrázek 8.4: Kompletní reprezentace mřížkového problému

Vytvořme si nový problém `GridProblem`, který bude implementovat interface `ISearchableProblem`. Rozhraní diktuje, že problém musí umět vrátit startovní bod, ověřovat cílové body a generovat přechody do nových bodů. Vše z toho jsme schopni realizovat, neboť problém má všechny potřebné informace. Při generování následníků pro bod se vezmou čtyři sousední čtverce, vyjma těch, co jsou mimo hranice pole nebo jsou překážkami.

Stav v popsaném systému implementuje rozhraní `ISearchNode` a bude obsahovat globální souřadnice X a Y, určující naši polohu na mřížce, nazveme ho `GridState`.

Pro efektivnější prohledávání ještě zbývá implementovat specifické heuristiky od `ISearchableHeuristic`, které musí vracet pro libovolný prohledávací uzel heuristickou hodnotu. Snadno implementujeme např. heuristiku založenou na Manhattan vzdálenosti nebo Euklidovské vzdálenosti a pro porovnání též slepou heuristiku. Implementace všech komponent shrnuje diagram na obr. 8.4.

Nyní už pouze stačí spustit výpočet pomocí standardní `AStarSearch` rutiny, která nám vrátí výsledek prohledávání, jak ilustruje ukázka 8.6 (při nespecifikované haldě se použije defaultní halda). Výsledný plán je sekvence stavů z počátečního stavu do cílového.

```
var problem = new GridProblem(size, start, goals, obstacles);
var heuristic = new ManhattanGridHeuristic(problem);
var search = new AStarSearch(problem, heuristic);

if (search.Start() == ResultStatus.SolutionFound)
{
    Console.WriteLine(search.GetSolutionPlan().ToString());
}
```

Zdrojový kód 8.6: Příklad definice a spuštění úlohy s mřížkou

8.4 Proč používat PAD

Framework PAD samozřejmě není první softwarový produkt svého druhu. Podívejme se na některé existující, známé implementace plánovačů a knihoven podporující plánování v PDDL a SAS$^+$ a srovnejme je s frameworkem PAD.

- **Fast Downward**[25] je známý plánovač zabývající se klasickým plánováním za pomocí heuristického prohledávání. Podporuje propoziční fragment jazyka PDDL verze 2.2, který interně převádí na SAS$^+$. Při řešení používá tzv. *kauzální graf*, pokročilou techniku využívající hierarchické dekompozice pro výpočet heuristické funkce. Plánovač je psán v jazyce C/C++ a optimalizován je primárně na výkon, a mimofunkční požadavky (dobrý OOP návrh, rozšiřitelnost, čitelnost, dokumentovatelnost apod.) jsou tím pádem lehce v ústraní. Není použitelný jako obecný framework pro další vývoj, ale jako výkonný, referenční plánovač.
- **HSP**[42] a **Fast-Forward**[39] jsou dva v minulosti úspěšné plánovače, podporující dopředné hledání v prostoru stavů za použití relaxačních heuristik. Daly základ pro vznik plánovače Fast Downward, či mnoha dnes známým heuristikám. Jako obecný framework jsou nepoužitelné. Zdrojové kódy těchto plánovačů jsou relativně nízkoúrovňové C/C++.
- **PDDL4J**[43] je poměrně nová knihovna pro PDDL plánování, napsaná v Javě. Obsahuje parser jazyka PDDL verze 3.1, nástroje pro pre-processing načteného PDDL vstupu a základní prohledávání. Implementováno je též několik základních relaxačních heuristik.

K dispozici jsou další spíše menší plánovače či parsery PDDL, nicméně žádný z nich nesplňuje požadavky specifikované pro framework PAD. Zatím žádné plánovače či knihovny zabývající se plánováním, zdá se, nebyly konstruovány jako framework a žádné nepodporují unifikované zpracování jak reprezentace PDDL, tak SAS$^+$. Pravděpodobně doposud neexistuje žádná významnější implementace plánovače v jazyku C#.

Odvrácenou stranou dobrého architektonického návrhu a dosti širokého záběru frameworku PAD však jistě bude o něco nižší výkonnost oproti plánovačům úzce zaměřeným na určitý fragment plánování či jednu určitou reprezentaci. Nároky na optimalizace jednotlivých komponent takto koncipovaného projektu jsou větší, avšak potenciál celého projektu je v delším horizontu také o mnoho větší.

Závěr

Cílem práce bylo navrhnout a vytvořit softwarový framework coby podpůrný prostředek pro výzkum a vývoj v oblasti plánování. Dle požadavků zadavatele má podporovat dva nejvýznamnější vstupní formáty PDDL a SAS$^+$ a implementovat základní prohledávací algoritmy a heuristiky klasického plánování, s důrazem na kvalitní objektový návrh a budoucí rozšiřitelnost produktu.

Práce je rozdělena na dvě hlavní části. V té první jsme čtenáři představili oblast plánování, vysvětlili základní koncepty a postupy při hledání řešení plánovacího problému. Koncepty byly ilustrovány na konkrétních modelových příkladech.

Druhá část práce se zabývala samotnou implementací frameworku PAD (celým názvem *Planning Algorithms Development Framework*). Byl načrtnut přístup k návrhu, analýze a konstrukci SW produktu z hlediska softwarového inženýrství a popsány jednotlivé fáze a postupy přímo v kontextu frameworku PAD.

Významnou část implementace frameworku představuje podpora pro zpracování vstupních dat. Pro tento účel byl vytvořen projekt *načítače* vstupních dat `PAD.Loader`. V případě jazyka PDDL bylo nutné vytvořit robustní parser dle definované vstupní gramatiky jazyka PDDL, zatímco pro formalismus SAS$^+$ jsme si vystačili s relativně jednoduchou lineární čtečkou. Výsledkem načítání dat je zapouzdřená, nezávislá komponenta vstupních dat.

Framework PAD podporuje načítání nejnovější PDDL verze 3.1, jakkoli ne všechny podporované aspekty jsou dále použity v rámci plánovače. Učiněno tak bylo proto, aby byl loader vstupních dat použitelný i pro ostatní kolegy z příbuzných oblastí vývoje (např. rozvrhování). Součástí loaderu je i komplexní validátor, ověřující konzistentnost vstupních dat a jejich validitu vůči specifikaci.

Další významnou částí frameworku je projekt plánovače `PAD.Planner`. Implementována byla řada používaných algoritmů heuristického prohledávání pro hledání řešení plánovacího problému. Podporováno je dopředné i zpětné prohledávaní, součástí plánovače je též implementace mnoha variant hald coby efektivních datových struktur používaných při prohledávání a nechybí implementace základních i komplexnějších heuristik.

Plánovač obsahuje společnou část pracující nad abstrakcí plánovacího problému (nezávislé na použité reprezentaci) a dále PDDL a SAS$^+$ specifické části realizující implementace v kontextu daného formalismu.

Velké úsilí bylo vloženo do vytvoření kvalitního testovacího balíku. V rámci frameworku byl implementován projekt `PAD.Tests`, obsahující bezmála 230 unit testů, ověřující funkcionalitu a korektnost jednotlivých komponent programu. Součástí testovacích případů jsou i reálné příklady z Mezinárodní plánovací soutěže (IPC). Dobré pokrytí testy podporuje budoucí rozšiřitelnost i údržbu produktu a obecně zvyšuje důvěru v používaný software.

V závěrečné části jsme se podívali na to, jakými způsoby lze používat framework PAD, či jak jednoduše lze vytvořit zcela novou reprezentaci problému. Popsali jsme též komponentu `PAD.Launcher`, realizující hromadné paralelní výpočty plánovacích problémů.

Framework byl umístěn na projekt GitHub pod volnou licencí GNU GPL. Veřejný repozitář, společně s popisem jednotlivých částí, příklady a dokumentací lze nalézt na adrese `https://github.com/PlanningFramework/PAD`.

Nakonec je třeba zmínit, že ne všechny součásti frameworku fungují naprosto optimálně. Mnohé komponenty jistě půjdou dále optimalizovat, nebo zcela nahradit efektivnějšími implementacemi. Cílem této práce však nebylo vytvořit soutěžní plánovač, ale vybudovat kvalitní a robustní základnu pro budoucí vývoj, která má potenciál přispět k budoucímu rozvoji v oboru plánování.

Další možná rozšíření

Zmiňme krátce oblasti a komponenty, které potenciálně zaslouží pozornost při dalším budoucím rozšiřování a optimalizaci frameworku PAD.

Jak již bylo nastíněno v části 7.2, efektivitu prohledávání lze úspěšně zvyšovat vytvářením specializovaných reprezentací na základě analýzy vstupního plánovacího problému či zohledňováním vstupních *requirementů* v případě PDDL. Experimentálně byla např. vytvořena specifická STRIPS implementace PDDL s explicitně groundovanými operátory, která na určitém souboru úloh byla až desetkrát rychlejší než bázová implementace. Pro spoustu jiných úloh však nebyla použitelná, protože explicitní groundování zahltilo paměť počítače, a zbytek úloh nebylo možné počítat kvůli úzké specializaci reprezentace.

Vytvářením specializovaných implementací a automatickým rozpoznáváním vhodné reprezentace pro jednotlivé sorty problémů lze v budoucnu dále zlepšovat výkonnost plánovače.

Některé detaily existující implementace plánovacích problémů zaslouží dále optimalizovat a rozšiřovat – např. PDDL v současnosti nepodporuje plánování s odvozenými predikáty, SAS^+ reprezentace by zase potenciálně zasloužila např. lepší zohlednění axiomatických pravidel při výběru operátorů.

Framework v současnosti obsahuje hlavně základní, *učebnicové* heuristiky a několik pokročilých heuristik. Sada heuristik by v budoucnu jistě zasloužila rozšířit o další populární přístupy z posledních let – zmiňme například heuristiky na principu *Merge and Shrink*, *Potential* heuristiky, *Landmark* heuristiky, *Cost Partitioning* heuristiky a další.

Zdroje a použitá literatura

[1] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[2] J. Matoušek, J. Nešetřil. *Kapitoly z diskrétní matematiky*. Karolinum, 2007.

[3] J. Koehler. Gripper domain (for the First International Planning Competition). `http://ipc98.icaps-conference.org/`, 1998.

[4] V. Švejdar. *Logika: neúplnost, složitost a nutnost*. Academia, Praha, 2002.

[5] T. H. Cormen, Ch. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (3rd Edition)*. MIT Press and McGraw-Hill, 2009.

[6] GNU GPL License. `https://www.gnu.org/licenses/gpl-3.0.en.html`.

[7] Project MONO. `https://www.mono-project.com/`.

[8] Irony Project. `https://github.com/IronyProject/`.

[9] Wintellect Power Collections. `https://nuget.org/packages/SoftUni.Wintellect.PowerCollections/`.

[10] P. Bourque and R. E. Fairley. *Guide to the Software Engineering Body of Knowledge*. IEEE Computer Society, 2014.

[11] Unified Modeling Language. `https://www.omg.org/spec/UML/`.

[12] E. Freeman, E. Robson, B. Bates, and K. Sierra. *Head First Design Patterns*. O'Reilly Media, United Kingdom, 2004.

[13] Robert C. Martin. *Principles Of OOD*. butunclebob.com, 2003.

[14] Inversion of Control Containers and the Dependency Injection Pattern. `https://martinfowler.com/articles/injection.html`.

[15] Microsoft Developer Network. C# Coding Conventions. `https://msdn.microsoft.com/en-us/library/ff926074.aspx`, 2015.

[16] S. McConnell. *Code Complete, 2nd Edition*. Microsoft Press, Redmond, WA, USA, 2004.

[17] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins. *PDDL - The Planning Domain Definition Language*. Yale Center for Computational Vision and Control, 1998.

[18] M. Fox and D. Long. *PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains*. Journal of Artificial Intelligence Research, 2003.

[19] S. Edelkamp and J. Hoffmann. *PDDL2.2: The Language for the Classical Part of the 4th International planning Competition*. Institut für Informatik, 2003.

[20] A. Gerevini and D. Long. *Preferences and Soft Constraints in PDDL3.* Proceedings of the ICAPS-2006 Workshop on Preferences and Soft Constraints in Planning, 2006.

[21] D. L. Kovacs. *BNF Definition of PDDL3.1: partially corrected, with comments/explanations.* IPC-2011, 2011.

[22] C. Backström and B. Nebel. *Complexity results for SAS$^+$ planning.* Computational Intelligence, 1995.

[23] P. Jonsson and C. Backström. *State-variable planning under structural restrictions: Algorithms and complexity.* Artificial Intelligence, 1998.

[24] Fast Downward Project – SAS$^+$ Format. `http://www.fast-downward.org/TranslatorOutputFormat`.

[25] M. Helmert. *The Fast Downward Planning System.* Institut für Informatik, 2006.

[26] I. Neamtiu, J. S. Foster, and M. Hicks. *Understanding Source Code Evolution Using Abstract Syntax Tree Matching.* ACM, 2005.

[27] D. E. Knuth. *Backus Normal Form vs. Backus Naur Form.* Communications of the ACM, 1964.

[28] DeRemer F. *Practical Translators for LR(k) Languages.* MIT, 1969.

[29] Planning Domains. `http://planning.domains/`.

[30] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach (3rd Edition).* Prentice Hall, Upper Saddle River, NJ, USA, 2009.

[31] R. Korf. *Depth-first Iterative-Deepening: An Optimal Admissible Tree Search.* Artificial Intelligence, 1985.

[32] S. Aine, S. Swaminathan, V. Narayanan, V. Hwang, and M. Likhachev. *Multi-Heuristic A*.* Conference Paper, Proceedings of Robotics: Science and Systems, 2014.

[33] D. Reddy. *Speech Understanding Systems: A Summary of Results of the Five-Year Research Effort.* Carnegie Mellon University, 1977.

[34] R. E. Fikes and N. J. Nilsson. *STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving.* Stanford Research Institute, Menlo Park, CA, USA, 1971.

[35] S. Sievers, M. Ortlieb, and M. Helmert. *Efficient Implementation of Pattern Database Heuristics for Classical Planning.* SOCS, 2012.

[36] P. Haslum, A. Botea, M. Helmert, B. Bonet, and S. Koenig. *Domain-Independent Construction of Pattern Database Heuristics for Cost-Optimal Planning.* Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, 2007.

[37] B. Bonet and H. Geffner. *Planning as Heuristic Search.* Artificial Intelligence 129, 2001.

[38] Ch. Betz and M. Helmert. *Planning with h^+ in Theory and Practice.* Institut für Informatik, 2009.

[39] J. Hoffmann and B. Nebel. *The FF Planning System: Fast Plan Generation Through Heuristic Search.* Journal of Artificial Intelligence Research, 2001.

[40] ReSharper. `https://www.jetbrains.com/resharper/`.

[41] Microsoft Code Analysis 2019. `https://docs.microsoft.com/cs-cz/visualstudio/code-quality/roslyn-analyzers-overview`.

[42] B. Bonnet and H. Geffner. *HSP: Heuristic Search Planner.* AIPS-98 Planning Competition, 1998.

[43] D. Pellier and H. Fiorino. *PDDL4J: A Planning Domain Description Library for Java.* Journal of Experimental and Theoretical Artificial Intelligence, 2018.

Seznam obrázků

A. Elektronická příloha

Elektronická příloha této práce obsahuje:

- kompletní zdrojové kódy frameworku PAD,
- vygenerovanou statickou dokumentaci frameworku PAD,
- sadu testovacích příkladů v rámci projektu `PAD.Tests`,
- specifikaci gramatiky jazyka PDDL 3.1,
- specifikaci vstupního formalismu SAS$^+$,
- URL link na GitHub stránku projektu PAD.