Matematicko-fyzikální fakulta Univerzity Karlovy v Praze

Roman Mecl

Omezující podmínky v grafických uživatelských rozhraních

Diplomová práce

Vedoucí práce: Mgr. Roman Barták, Dr.

Praha 1999

Děkuji Mgr. Romanu Bartákovi, Dr. za trpělivé a soustavné vedení při zpracování diplomové práce.

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně a uvedl veškerou použitou literaturu

V Praze dne 7. května 1999

OBSAH

Úvod

Omezující podmínky jsou efektivním nástrojem, který umožňuje popisovat velké množství problémů. Výhodou tohoto popisu je jeho deklarativní charakter, který je podobný přesnému matematickému zadání problému. Spolu s existencí obecných efektivních algoritmů, které umějí tyto sady omezujících podmínek řešit, tzn. převádět řešení z implicitní podoby (množina podmínek) do podoby explicitní (určení hodnot proměnných), nabízejí omezující podmínky velmi široké spektrum využití v mnoha počítačových oblastech, jako je umělá inteligence, počítačová grafika, kombinatorika, robotika. Přestože programování s použitím omezujících podmínek zatím nepatří mezi hojně rozšířené postupy, dá se očekávat, že v dalších letech spolu s vývojem nových a výkonnějších algoritmů, které dovedou omezující podmínky řešit, dojde k jejich masovějšímu použití.

Tím, že programátor může ponechat zodpovědnost za udržování požadovaných vztahů, např. v uživatelském rozhraní, na systému, nemusí se starat o tyto vztahy vlastními prostředky, a tím také omezí případný výskyt chyb. Usnadní mu to vývoj a udržování rozsáhlých systémů. Navíc omezující podmínky nabízejí deklarativní způsob zápisu požadovaných vlastností. Tento zápis je pak jistě přehlednější než fragment kódu programu, který zajišťuje tutéž funkcionalitu. Také při změně některých vlastností stačí v případě použití omezujících podmínek jen upravit odpovídající popisy, a tak není nutné radikálním způsobem program měnit.

Cílem této práce je seznámit se s teorií omezujících podmínek, především se způsoby jejího použití v grafických uživatelských rozhraních a se známými algoritmy, které tyto omezující podmínky řeší. Na základě takto získaných znalostí pak navrhnout a implementovat grafický editor, který k popisu grafických objektů omezující podmínky využívá. Práce si neklade za cíl vytvořit plnohodnotný grafický editor srovnatelný s komerčními produkty, spíše je zaměřena na prezentaci možností omezujících podmínek v grafických uživatelských rozhraních. Součástí práce je také výběr vhodného řešiče, který se postará o vlastní řešení systémů podmínek vytvářených grafickým editorem.

Diplomová práce je rozdělena do tří částí. V první části je podán stručný výklad základů teorie omezujících podmínek se zvláštním zřetelem na hierarchie podmínek. Druhá část je věnována především popisu algoritmů řešících hierarchie omezujících podmínek a otázkám jejich použitelnosti v konkrétním případě grafických uživatelských rozhraní. Závěrečná třetí část je pak popisem návrhu i vlastní implementace grafického editoru.

Práce je zakončena shrnutím dosažených cílů a poznámkami o možnostech dalšího rozšíření. V příloze jsou pak návody, jak lze grafický editor obohatit o nové grafické objekty a jak vyměnit interní řešič omezujících podmínek. Součástí práce je rovněž stručný uživatelský návod popisující ovládání grafického editoru.

1 Část první – základní pojmy

Nejprve vysvětlíme základní pojmy, které se omezujících podmínek týkají, popíšeme rozšíření teorie o hierarchie podmínek a řekneme, jakými způsoby lze dosáhnout větší efektivity algoritmů, které tyto omezující podmínky řeší. Soustředíme se především na aplikaci omezujících podmínek v grafických uživatelských rozhraních (dále jen GUI).

Omezující podmínka je, obecně řečeno, relací zachycující vztah mezi jejími proměnnými. Příkladem omezujících podmínek jsou lineární rovnice a nerovnice. Domény těchto proměnných mohou být různé, nejčastěji jde o určitou podmnožinu reálných čísel. K vyřešení systému, který je popsán množinou omezujících podmínek, je tedy nutné najít pro jednotlivé proměnné vyskytující se v systému takové hodnoty z jejich domén, že všechny podmínky jsou splněny.

1.1 Omezující podmínky v GUI

Omezující podmínky v GUI obecně vyjadřují vztahy mezi grafickými objekty. Především jde o matematické rovnice a nerovnice nad reálnými proměnnými, které k popisu geometrických objektů nabízí analytická geometrie. Způsob řešení takových podmínek pak vychází z numerických metod vypočtení číselných hodnot proměnných. Příklady takových podmínek jsou: obecný čtyřúhelník je obdélníkem, vybrané objekty jsou zarovnány, určený objekt je uvnitř vymezené oblasti, nebo určené tři body jsou kolineární.

1.2 Hierarchie a komparátory

Tím, že na systém klademe omezující podmínky, automaticky očekáváme splnění těchto podmínek při řešení systému. Máme-li ale na systém příliš nároků, může se stát, že nelze všechna omezení splnit. O takovém systému říkáme, že je *příliš omezený*.

Teorie hierarchií podmínek byla navržena v souvislosti s řešením příliš omezených systémů jako způsob, jak tuto záležitost řešit. Umožňuje deklarativní popis nejen podmínek, které je nutné splnit, tzv. *pevných podmínek*, ale i podmínek, které nemusí být nutně splněny, tzv. *podmínek slabých*. Pokud je systém popsán pouze pomocí pevných podmínek, může nastat takový stav, že pro definovanou sadu podmínek nelze najít žádné řešení, systém je příliš omezený. Jestliže chceme i v takovém případě získat řešení alespoň části systému, máme dvě možnosti. Jednak můžeme postupně vyřazovat některé podmínky ze systému a zbylé opakovaně řešit až do doby, kdy už bude systém řešitelný. Druhou možností je použít hierarchii podmínek, tzn. každé podmínce přiřadíme její stupeň důležitosti. Podmínky jsou

pak splňovány postupně: nejdříve podmínky nejdůležitější a dále pak další podmínky sestupně dle důležitosti tak, aby se ve výsledném řešení neobjevila situace, že některá méně důležitá podmínka je uspokojena, ale její splnění brání možnosti uspokojit důležitější podmínku.

Vnoříme-li tedy standardní systém podmínek do hierarchické varianty, odpovídá to situaci, ve které všechny podmínky mají nejvyšší stupeň důležitosti, jsou tzv. vyžadované. Dále se musíme rozhodnout, kterým podmínkám snížíme stupeň důležitosti tak, že potom nebude nezbytně nutná jejich splnitelnost. Tím docílíme toho, že systém přestane být příliš omezený.

Hierarchii podmínek tedy lze zjednodušeně popsat jako množinu podmínek, které jsou ohodnoceny určitým stupněm důležitosti. Pro tyto stupně se často používájí tato označení: povinná (required), silná (strong), střední (medium) a slabá (weak) podmínka. Řešení dané hierarchie podmínek je potom to ohodnocení z množiny všech ohodnocení proměnných v povinných podmínkách, které je nejlepší při respektování hierarchie.

Pro formální zavedení je nutno zpřesnit některé pojmy. *Omezující podmínka* je relace nad doménou D. *Ohodnocení* množiny omezujících podmínek je funkce, která mapuje proměnné podmínek na prvky domény D.

Označená omezující podmínka (labeled constraint) pak je podmínka doplněná *preferencí*. O množině preferencí se předpokládá, že je lineárně uspořádaná. Je tedy možné jednotlivé stupně preference mapovat do množiny přirozených čísel, přičemž povinné podmínky odpovídají stupni 0.

Definice 1: *Hierarchie podmínek* je konečná množina označených omezujících podmínek.

Příklad hierarchie podmínek

c1	required	a + b = 15
c2	strong	b + c = 10
c3	weak	c = 10
c4	weak	b = 5
c5	weak	a = 5

Hierarchii podmínek lze rozdělit do vrstev takto:

H ... {všechny podmínky systému }

H_0 ...{ povinné podmínky systému (nejvyšší stupeň důležitosti)}

H_1 atd. { podmínky systému s 1. stupněm důležitosti (např. strong)}

V předchozím příkladu jsou pak rozděleny podmínky takto:

$H_0=\{c_1\}$, $H_1=\{c_2\}$, $H_2=\{\}$ a $H_3=\{c_3, c_4, c_5\}$

Definice 2: *Potenciální ohodnocení* hierarchie H je takové ohodnocení, které splňuje všechny povinné podmínky, tedy podmínky z H_0. Množinu všech potenciálních ohodnocení pro hierarchii H označme S_0.

S_0 = { U| $\forall c \epsilon$ H_0, cU platí}, přičemž cU značí výsledek aplikace ohodnocení U na podmínku c.

Je zřejmé, že libovolné řešení hierarchie H, má-li hierarchie nějaké, musí být prvkem S_0.

Definice 3: *Množina řešení* hierarchie podmínek H je množina takových potenciálních ohodnocení, pro která neexistuje v množině S_0 lepší(better) potenciální řešení, vzhledem k relaci better. Množinu řešení značíme S.

S = { U| U$\epsilon$$S_0$ & $\forall$V$\epsilon$$S_0$ $\neg$better(V,U,H)}

Jak již bylo zmíněno, tím, že některá podmínka v hierarchii má stupeň důležitosti jiný než povinná, může při řešení dojít k situaci, že podmínka není splněna. Jednoduchý algoritmus řešící hierarchii potom takovou podmínku vyřadí a ta potom nemá na výsledné řešení žádný vliv. Toto chování však příliš našim představám neodpovídá. Intuitivním požadavkem pro chování chytřejšího algoritmu patrně je, aby se pokusil i tyto nesplnitelné podmínky co nejvíce uspokojit. V takovém případě je vhodné zavést tzv. *chybovou funkci*, e: cU $\rightarrow$ R. Ta pro každou podmínku *c* a každé ohodnocení *U* vrací nezáporné reálné číslo, které říká, jak velké chyby se při uspokojení podmínky daným ohodnocením dopouštíme, tzn. jak se liší reálné ohodnocení od požadovaného stavu, přičemž chybová funkce je rovna nule právě tehdy, když *cU* platí.

Množinu řešení S jsme zavedli jako množinu takových ohodnocení z S_0, že k nim neexistuje žádné lepší řešení. Nyní relaci better, tzv. *komparátor*, zavedeme formálně.

Definice 4: *Komparátor* je irreflexivní, tranzitivní relace nad jednotlivými ohodnoceními dané hierarchie H. Navíc musí *respektovat hierarchii*, tzn. jestliže nějaké ohodnocení v S_0 řeší všechny podmínky až do stupně důležitosti *k*, pak všechna ohodnocení v S musí splňovat podmínky až do stupně *k*.

Zapíšeme-li požadavky předchozí definice formálně, dostáváme tyto výroky.

$\forall H \forall u \neg$ better(u,u,H) (irreflexivita)

$\forall H \forall u \forall v \forall w$ (better(u,v,H) & better(v,w,H) $\Rightarrow$ better(u,w,H)) (tranzitivita)

($(\exists u \epsilon S_0 \exists k>0 \forall i \epsilon \{1,\ldots,k\} \forall c \epsilon H_i$ cu platí) $\Rightarrow$ ($\forall v \epsilon S_0 \exists j \epsilon \{1,\ldots,k\} \exists c' \epsilon H_j$ c´v neplatí))

$\Rightarrow \exists w \epsilon S_0$ better(w,v,H) (respektování hierarchie)

Komparátor tedy vytváří částečné uspořádání nad množinou S_0.

Blíže se seznámíme se dvěma skupinami používaných komparátorů.

První z nich jsou tzv. *lokální (locally-better) komparátory*. Ty posuzují jednotlivé podmínky hierachie nezávisle na ostatních podmínkách.

Definice 5: Ohodnocení U je *locally-better (LB)* než ohodnocení V, jestliže pro všechny podmínky až do stupně důležitosti *k-1* jsou chyby stejné pro obě ohodnocení, na stupni důležitosti *k* existuje alespoň jedna podmínka, pro kterou je chyba při U ostře menší než chyba při V, a chyby ostatních podmínek stupně *k* jsou menší nebo rovny pro ohodnocení U.

locally-better(u,v,H) $\equiv_{def}$

$\exists k>0\ \forall i \in\{1,\ldots,k-1\}\ \forall c\in H_i\ e(cu)=e(cv)\ \&\ \exists c'\in H_k\ e(c'u)<e(c'v)\ \&\ \forall c\in H_k\ e(cu)\leq e(cv)$

Nejjednodušším příkladem z této skupiny komparátorů je *locally-predicate-better* komparátor (LPB). Chybová funkce pro tento komparátor je triviální - nabývá pouze dvou hodnot: 0 (splněna), nebo 1 (nesplněna).

Jiným příkladem je již zmiňovaný přístup, při kterém určujeme nejenom splněnost/nesplněnost podmínky, ale i míru této splněnosti. Odpovídající komparátor se nazývá *locally-error-better(LEB)*. Chybová funkce potom odpovídá nějaké metrice nad proměnnými dané podmínky. Vezmeme-li například systém s *n* proměnnými nad doménou reálných čísel a jako metriku vzdálenost v prostoru R^n, pak chybová funkce odpovídá vzdálenosti skutečného ohodnocení od ohodnocení požadovaného.

Příklad nejlepšího řešení pro LEB komparátor
required $a \geq 20$
weak $a = 10$
Doménou proměnné *a* je R. Jediné řešení v případě použití LEB komparátoru je [a] = [20]. Druhá podmínka tedy splněna není a její chyba je 10, dle min |skutečná hodnota-požadovaná hodnota|. Kdybychom však použili komparátor LPB, byla by množina všech řešení nekonečná: $S=\{[x],x\in R\ \&\ x\geq 20\}$.

Druhou skupinou komparátorů jsou tzv. *globální (globally-better) komparátory*. Ty porovnávají ohodnocení vždy v rámci všech podmínek jednoho stupně důležitosti tím, že určují kombinovanou chybu všech těchto podmínek. K tomu je třeba zavést tzv. *kombinační funkci*, která každému ohodnocení na zvolené úrovni podmínek přiřazuje nějaké nezáporné reálné číslo. Navíc musí splňovat podmínku:

$g(u,H_k)=0 \Leftrightarrow \forall c\in H_k$ cu platí.

V závislosti na volbě kombinační funkce dostáváme různé globální komparátory, kombinační funkce je tedy u predikátu globally-better dalším parametrem.

Definice 6: Ohodnocení U je *globally-better (GB)* než ohodnocení V, pokud kombinované chyby až do stupně *k-1* jsou stejné pro obě ohodnocení a kombinovaná chyba stupně *k* je ostře menší pro ohodnocení U.

globally-better(u,v,H,g) $\equiv_{def}$

$\exists k>0\ \forall i \in\{1,\ldots,k-1\}\ g(u,H_i)=g(v,H_i)\ \&\ g(u,H_k)<g(v,H_k)$

Existuje několik komparátorů z kategorie globálních, které se běžně používají. Liší se jednak způsobem, jakým počítají výše zmíněnou kombinovanou chybu podmínek určitého stupně důležitosti, jednak použitou metrikou v chybové funkci. Příkladem je *weighted-sum-better (WSB)* komparátor, kde použitá metrika chybové funkce je stejná jako u LEB komparátoru a kombinovaná chyba se počítá jako vážený součet chyb jednotlivých podmínek. Každá podmínka může mít totiž kromě stupně důležitosti ještě svoji váhu. Tou lze mezi podmínkami stejného stupně důležitosti ještě docílit toho, že splnění některých podmínek je pro systém o něco přednější než splnění jiných. Hlavní rozdíl mezi váhou a preferencí podmínky spočívá v tom, že preference rozdělují podmínky do vrstev tak, že podmínka z vrstvy se slabší preferencí nemůže způsobit nesplnění libovolné podmínky se silnější preferencí. U vah ovšem může nastat situace, že je preferováno řešení, které vede ke splnění několika podmínek s menší váhou, před ohodnocením, které tyto podmínky nesplňuje, zato však splňuje jinou podmínku s větší váhou.

V některých implementacích algoritmů jsou pro zjednodušení preference a váhy podmínek řešeny jedním způsobem. Například preference je chápána jako váha podmínky. V takových případech je nutné zajistit, aby se implementace chovala korektně vzhledem k definici hierarchie.

Následující příklad ukazuje, jak pro jednu hierarchii podmínek může použití odlišných komparátorů dávat různá řešení.

Příklad použití různých komparátorů

required	a + b = 15	(-)
strong	b + c = 10	(1,0)
weak	c = 10	(0,3)
weak	b = 5	(0,8)
weak	a = 15	(0,4)

Váhy jednotlivých podmínek jsou uvedeny v závorce. U podmínek, které jsou povinné (required), je tento údaj zbytečný, protože musí být vždy splněny, není tedy třeba některé z nich preferovat.

S použitím komparátoru LPB existují dvě nejlepší řešení [a,b,c] = [15,0,10] a [10,5,5]. Pro komparátor *weighted-sum-predicate-better* (WSPB - speciální případ WSB, kde chybová funkce je triviální) už je řešení jediné, a to [10,5,5], protože má chybu na úrovni weak podmínek jen (0,7), zatímco řešení [15,0,10] má chybu (0,8). A nakonec s využitím komparátoru *unsatisfied-count-better* (speciální případ předchozího WSPB, pro váhy všech podmínek rovny jedné), který udává jako chybu počet nesplněných podmínek pro každou hierarchickou úroveň, je řešením pouze [15,0,10], protože není splněna jediná podmínka weak b = 5.

Formální definice zde zavedených pojmů lze nalézt v [2].Tato práce rovněž obsahuje důkazy, že jednotlivé predikáty locally-better a globally-better splňují vlastnosti komparátorů.

1.3 Edit a stay podmínky

V systémech podmínek zaujímají zvláštní postavení dva druhy podmínek. Mnohdy od systému požadujeme, aby byl parametrický, tzn. aby bylo možné sledovat jeho chování při změnách hodnot některých jeho předem určených proměnných. Potřebujeme tedy způsob, jak do systému tyto parametrické hodnoty vkládat. Zavedeme proto tzv. *edit podmínku*, což není nic jiného než rovnost x = k, kde *x* je proměnná, jejíž hodnotu chceme parametricky měnit, a *k* je hodnota, kterou do systému při každé žádosti o vyřešení dodáme z vnějšku. Použití této edit podmínky si lze snadno představit právě pomocí grafického editoru, ve kterém jsou objekty popsány systémem podmínek. Chceme-li libovolný objekt přesunout pomocí myši, je nutné změnit jeho souřadnice tak, aby odpovídaly novým souřadnicím myši. Právě tehdy lze použít edit podmínku, která nastaví proměnné odpovídající souřadnicím objektu patřičnými hodnotami.

Druhou zvláštní omezující podmínkou je tzv. *stay podmínka*. Ta zařizuje jistou stabilitu systému při jeho drobných změnách. Váže totiž proměnnou s její poslední hodnotou. Význam této podmínky spočívá v tom, že zajišťuje, aby se hodnota proměnné, kterou podmínka svazuje, neměnila, dokud to není nezbytně nutné kvůli jiné důležitější podmínce. Některé algoritmy dokonce přítomnost stay podmínek na všech proměnných vyžadují, aby pracovaly korektně. Dalším důvodem, proč je vhodné použít stay podmínky, je, že zadaný systém podmínek nedává jediné řešení. Použitím stay podmínek můžeme systém dodatečně omezit tak, aby řešič nabízel vždy jen jedno řešení.

Tím, že stay podmínky požadují neměnnost hodnot svých proměnných, je nutné jim přiřadit nejnižší možnou preferenci, aby splnění ostatních podmínek bylo preferováno a stay podmínky byly použity jen v případech, kdy předchozí podmínky hodnotu proměnné neurčí jednoznačně.

Příklad systému podmínek, který nedává jednoznačné řešení
required a+b = c
medium c = 10

Příklad systému podmínek dodatečně omezeného stay podmínkami
required a+b = c
medium c = 10
weak stay a
weak stay b

Existuje jisté množství algoritmů, které jsou více či méně vhodné k řešení omezujících podmínek. Využívají různé postupy a techniky výpočtu. Ty efektivnější se však shodují v některých základních principech. Proto je dobré se věnovat těmto obecným principům před bližším seznámením se s algoritmy.

1.4 Inkrementalita a kompilace

V GUI je často nutné řešit velmi podobné soustavy podmínek opakovaně. Proto je třeba zabývat se způsobem, jak při řešení následujících soustav využít řešení soustav předchozích.V GUI dochází často ke dvěma typům situací, v nichž se soustava podmínek v závislosti na akci uživatele změní pouze nepatrně.

Za prvé: pokud uživatel mění vlastnosti objektů nebo jejich částí. To se většinou projeví přidáním nebo odebráním jedné nebo více podmínek. V takovém případě je sice někdy nutné přepočítat celý systém podmínek, ale ve většině případů se požadované změny projeví jen na jeho části. Je tedy žádoucí využít znalosti předchozího stavu systému a k novému řešení využít jeho největší možnou část, které se tyto poslední změny nedotýkají. Vlastnost algoritmu, která výše uvedenou myšlenku realizuje, se nazývá *inkrementalita*.

Za druhé: pokud uživatel chce tažením jednoho z objektů pomocí myši docílit jeho přemístění. Pohyb myši je realizován dvojicí edit podmínek, které vytvářejí relaci mezi pozicí myši a souřadnicemi objektu, který je tažen. V takovém případě je nutné znovu řešit při každém překreslení obrazovky téměř stejnou soustavu podmínek, a ta se liší pouze v hodnotách proměnných u edit podmínek. V takovém případě není systému přidávána ani ubírána žádná podmínka, pouze je třeba přepočítat podmínky, kterých se změna dotkla, aby podmínky opět nabyly platnosti.

Výše uvedený výpočet je nutné provádět často, a je tedy zapotřebí, aby byl co možná nejrychlejší. Tím, že jde o speciální situaci, není v tomto případě standardní použití algoritmu efektivní, protože ten se zabývá i stavy, které v těchto případech nemohou nastat. Lze provést jakousi *kompilaci* neboli převedení obecného stavu podmínek do efektivnějšího tvaru, který

dovoluje velmi rychle přepočítat hodnoty proměnných v předem připraveném pořadí tak, aby znovu byly všechny podmínky systému splněny. Pokud tedy začneme měnit hodnotu určitých proměnných, lze provést kompilaci systému podmínek a vytvořit si *plán* pro efektivní vypočtení nových hodnot proměnných. Tohoto plánu lze pak využívat až do doby, kdy budeme chtít měnit hodnoty jiných proměnných nebo když dojde ke změně systému podmínek přidáním nové podmínky, popř. odebráním některé podmínky ze systému.

1.5 Lokální propagace

Speciálním druhem kompilace, kterému se nyní budeme věnovat, je *lokální propagace*. Zjednodušeně řečeno, lokální propagace je kompilací podmínek do tvaru, ve kterém je zohledněno, jak se podmínky navzájem ovlivňují.

Lokální propagace je jedním z nejjednodušších a nejobecnějších principů, jaké se při řešení omezujících podmínek využívají. Předpokladem je, že všechny podmínky jsou tzv. *funkcionální*, tedy takové, že každou podmínku lze vyjádřit množinou *metod*- procedur, které na základě znalostí hodnot některých proměnných podmínky, tzv. vstupních proměnných, jednoznačně vypočítají zbylé proměnné podmínky, tzv. výstupní proměnné, tak, aby byla podmínka splněna. Má-li být podmínka v systému splněna, musí pro ni být vybrána některá z jejích metod.

> Příklad: Pro podmínku A+B=C existují 3 různé metody C←A+B, A←C-B, B←C-A, které se liší ve vstupních i výstupních proměnných.

> Příklad: Podmínka $A \leq B$ sice také umožňuje ze vstupních proměnných vypočítat proměnné výstupní, přesto není funkcionální. Neurčuje totiž hodnoty výstupních proměnných jednoznačně.

Algoritmus pracující nad funkcionálními omezujícími podmínkami může ve svém výpočtu dospět do stavu, kdy pro nějakou podmínku *c* zná hodnoty všech vstupních proměnných některé metody *m* dané podmínky *c*. V tom případě lze metodu *m* použít k výpočtu hodnot jejích výstupních proměnných, a tím zajistit splnění podmínky *c*. Při snaze splnit další podmínky již lze ze znalosti hodnot těchto proměnných vycházet a pro výpočet si vybere opět tu metodu podmínky, u které jsou již známy všechny její vstupní proměnné. Tímto způsobem algoritmus pokračuje do té doby, dokud nejsou známy hodnoty všech proměnných.

Příklad: Nechť máme podmínky A+B=C a C+D=E. Předpokládejme, že algoritmus zná hodnoty proměnných B,D,E. Použitím metody C← E-D druhé podmínky lze určit proměnnou C a následně promítneme-li tuto znalost do první podmínky, lze použít metodu A← C-B k určení proměnné A.

Techniky lokální propagace lze tedy úspěšně používat v situacích, kdy je systém v konzistentním stavu, jsou známy hodnoty všech jeho proměnných a přijde požadavek na změnu hodnot některých proměnných. V takovém případě není vždy nutné přepočítávat všechny podmínky, ale pouze promítnout požadované změny k podmínkám, které to ovlivní, a pouze ty postupně přehodnotit. Zbytek systému lze ponechat v původním stavu.

Obecně nemusí být metodami funkcionálních podmínek jen numerické funkce. Jde vlastně o akci, která vede k určení hodnot výstupních proměnných, může tedy například spojit jméno objektu s jeho fyzickým umístěním na obrazovce.

Výhodou systémů využívajících lokální propagaci je efektivnost a použitím metod také jistá obecnost. Omezení jsou, že neumějí vyřešit například soustavu rovnic a že mohou používat pouze funkcionální podmínky. Důvodem k takovému omezení je fakt, že algoritmy založené na lokální propagaci se vlastními silami nemohou vyrovnat se situací, kdy se v podmínkách objeví cyklus. K takovému stavu například dojde tehdy, pokud k určení hodnoty jedné proměnné je zapotřebí znalost druhé proměnné, k jejímuž určení je opět třeba znalosti hodnoty první proměnné. V další kapitole je význam pojmu cyklus v omezujích podmínkách šířeji rozveden.

Příklad sady podmínek, ve které je cyklus
Podmínky A+B = 5 a A –B = 3 tvoří cyklus. Chceme-li totiž z první podmínky určit proměnnou A, je nutná znalost proměnné B. Tu sice můžeme určit z druhé podmínky, ale k tomu opět potřebuje znát proměnnou A.

Systémy založené na lokální propagaci lze rozdělit do několika skupin podle druhu metod. Rozlišujeme je jednak podle toho, zda mají podmínky pouze jednu nebo více metod, jednak mají-li metody jednu nebo více výstupních proměnných.

1.6 Graf podmínek a řešící graf

Při popisu myšlenek jednotlivých algoritmů budeme také často využívat pojmy graf systému a řešící graf. Ty umožňují přehlednější reprezentaci systému podmínek pomocí teorie grafů, proto nyní tyto pojmy objasníme.

Definice 7: *Graf systému* je bipartitní graf G=(V,C,E), kde V a E jsou množiny proměnných systému, resp. podmínek, a E je množina neorientovaných hran taková, že pro lib. proměnnou *v* podmínky *c* obsahuje množina E hranu spojující *v* a *c.*

Aplikujeme-li na jednotlivé podmínky některou z jejích metod, určujeme tak „průtok informací“ podmínkou. Zvolení metody lze naznačit orientací jednotlivých hran grafu spojených s podmínkou. Orientace jednotlivých hran určuje, které proměnné jsou vstupní a které výstupní, čímž také naznačují, která metoda dané podmínky byla vybrána. Vede-li šipka z *v* do *c*, je *v* vstupní proměnnou, pokud šipka směřuje z *c* do *v*, pak jde o proměnnou výstupní.

Definice 8: *Orientovaný graf systému* je graf systému, u kterého jsou jednotlivé hrany orientovány dle použitých metod jednotlivých podmínek.

Pomocí pojmu orientovaný graf systému lze zpřesnit význam pojmu cyklus v systému omezujících podmínkách. Je třeba si také uvědomit, že pro danou množinu podmínek H může existovat více orientových grafů systému H v závislosti na množství alternativních metod pro jednotlivé omezující podmínky.

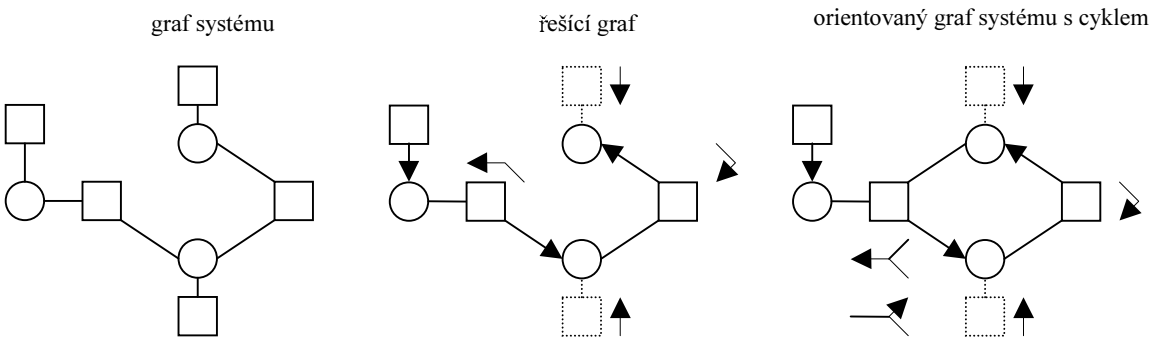
Definice 9: Množina omezujících podmínek H obsahuje *cyklus*, jestliže všechny orientované grafy systému H obsahují orientovaný cyklus.

K zavedení pojmu *řešící graf*, který popisuje řešení systému podmínek vhodným podgrafem orientovaného grafu systému, je třeba si uvědomit, že nalezneme-li nějaké řešení, neznamená to, že všechny podmínky musejí být splněny. Pro tyto nesplněné podmínky nelze vybrat žádnou metodu, takže není určena orientace jejich hran v grafu. Proto nesplněné podmínky nemohou být zahrnuty v řešícím grafu, který má všechny hrany orientované dle zvolených metod jednotlivých podmínek.

Definice 10: *Řešící graf* $G_1(V,E_1)$ je maximální podgraf orientovaného grafu systému G(V,E) ($E_1 \subseteq$ E) takový, že do každé proměnné vede nejvýše jedna hrana (1) a neobsahuje cyklus (2). Maximálním podgrafem se rozumí takový podgraf, ke kterému již nelze přidat žádnou hranu, aniž by zůstaly zachovány vlastnosti (1) a (2).

Při grafickém znázornění řešícího grafu často pro přehlednost používáme šipky pouze k zvýraznění výstupní proměnné. Je rovněž vhodné u vrcholu každé podmínky naznačit piktogramem její alternativní metody. Jak již bylo řečeno, nesplněné podmínky nejsou součástí řešícího grafu. Přesto je vhodné je zobrazovat, a proto pro jejich odlišení používáme například tečkování.

Následující znázornění jsou příklady grafu systému, korektního řešícího grafu a grafu s orientovaným cyklem. Proměnné jsou znázorněny kolečky a podmínky čtverečky.



2 Část druhá – algoritmy řešení hierarchií podmínek

V druhé části se již budeme zabývat jen algoritmy pro řešení hierarchií omezujících podmínek se zřetelem na interaktivní grafiku. Nejdříve stanovíme požadavky, které je nutné na tyto algoritmy klást, dále se blíže seznámíme s některými algoritmy, které mohou být vhodné k použití v interaktivní grafice, a klasifikujeme je podle jejich vlastností. Na závěr, po vyslovení svých požadavků, vybereme vhodný algoritmus, který bude nejlépe vyhovovat pro naši další práci.

2.1 Omezující podmínky v GUI

Jak již bylo řečeno, v grafických uživatelských rozhraních se setkáme především s omezujícími podmínkami, které vycházejí z analytické geometrie. Proměnné tedy mají většinou jako doménu nějaký interval reálných čísel, protože pro znázornění jejich hodnot využíváme nějaké grafické výstupní zařízení. Takže, chceme-li kreslit grafické objekty na obrazovku, je třeba se omezit například na hodnoty v intervalu <0;600>. Pro kreslení bychom sice vystačili s celými čísly, ale mohlo by to komplikovat některé výpočty, takže využíváme reálná čísla a pro zobrazování pak hodnoty zaokrouhlujeme. Tím se sice dopouštíme jisté nepřesnosti, ta je ale při zobrazení zanedbatelná.

Kromě běžných podmínek, které odpovídají matematickým rovnicím a nerovnicím, se v GUI neobejdeme také bez edit a stay podmínek.

Příklad použití omezujících podmínek v GUI
Pokusme se nyní pomocí omezujících podmínek vyjádřit tento vztah: dané tři body A,B,C leží na jedné přímce tak, že bod C je středem úsečky AB.
Body vyjádřeme pomocí dvojic proměnných, které budou reprezentovat souřadnice x a y. Tedy A = $[x_1,y_1]$, B = $[x_2,y_2]$, C = $[x_3,y_3]$. Omezíme doménu proměnných na reálná čísla v rozsahu <0, 600>.
V prvé řadě je třeba vytvořit podmínky, které se týkají bodů izolovaně: omezení domény všech proměnných a stay podmínky na všechny proměnné. Pro jednoduchost jsou tyto podmínky zapsány vždy pro všechny proměnné společně. Ve skutečnosti každý řádek odpovídá 6 reálným podmínkám.

required	$x_1,x_2,x_3,y_1,y_2,y_3 \geq 0$
required	$x_1,x_2,x_3,y_1,y_2,y_3 \leq 600$
weak	stay x_1,x_2,x_3,y_1,y_2,y_3

Podmínky zachycující vztah, že bod C je středem úsečky AB pak vypadají takto:

required	$(x_1+x_2)/2 = x_3$
required	$(y_1+y_2)/2 = y_3$.

Chceme-li navíc připravit systém na situaci, kdy měníme polohu prostředního bodu, přidáme k těmto podmínkám ještě

medium	edit x_3
medium	edit y_3.

2.2 Kritéria výběru vhodného algoritmu v GUI

K tomu, aby byl daný algoritmus vhodný k použití, je při interaktivní grafice potřebné, aby prováděné výpočty vždy proběhly ve velmi krátkém čase a interakce s uživatelem mohla být plynulá. Efektivity algoritmů lze dosahovat technikami zmíněnými v první části, které mohou v mnoha situacích výrazně zkrátit dobu výpočtu. Dále ke zvýšení efektivity algoritmu přispívá jeho specializovanost na určitou podmnožinu omezujících podmínek. Proto je třeba před aplikací algoritmu v daném případě zvážit vhodnost jeho použití.

Pokud vybíráme vhodný algoritmus pro výpočty pro grafická rozhraní, je třeba udělat několik rozhodnutí. V mnoha případech je nutno se rozhodnout mezi efektivitou a možnostmi algoritmu. Vyžadujeme-li univerzální algoritmus, musíme očekávat delší odezvy, a naopak, jestliže potřebujeme velmi rychlý algoritmus, je nutné omezit požadavky, které na něj klademe.

- funkcionální podmínky vs obecné podmínky

U funkcionálních podmínek lze předpokládat výrazně rychlejší výpočty, neboť algoritmy mohou využívat techniku lokální propagace. Naproti tomu u obecných systémů je nutné při každé změně přepočítávat celý systém.

- povinné podmínky vs hierarchie podmínek

Pokud se rozhodneme pouze pro povinné podmínky, je třeba věnovat zvýšenou pozornost při přidávání nových podmínek do systému, neboť může dojít k jeho přílišnému omezení. Na druhou stranu však systém plně odpovídá požadovaným omezením.

- cykly v podmínkách

Toto rozhodnutí vyžaduje poněkud hlubší analýzu problému, který chceme pomocí omezujících podmínek řešit. Ukazuje se však, že v mnoha problémech je výskyt cyklu nevyhnutelný. Někdy lze sice popsat určitý problém sadou podmínek, jejichž graf je acyklický. K tomu je ale mnohdy zapotřebí detailnějšího rozboru problému. Častým případem totiž bývá, že v systému může vzniknout cyklus, pokud podmínka přidaná do systému je nadbytečná - je již implikována předchozími podmínkami. Bylo by tedy nutné stále kontrolovat, zda podmínka, o kterou chceme systém rozšířit, není zbytečná. To je však zcela

proti duchu použití omezujících podmínek, jejichž výhodou je právě deklarativní charakter a ponechání analýzy sady podmínek až na algoritmu.

- rovnice vs nerovnice

Jak jsme již podotkli, v grafických uživatelských rozhraních se využívají především numerické omezující podmínky nad proměnnými s číselným oborem hodnot. Je třeba tedy rozhodnout, zda k popisu problému vystačíme pouze s rovnostmi, nebo budeme potřebovat také nerovnosti. Vzhledem k tomu, že rovnice jsou speciálním případem funkcionálních podmínek, překrývá se toto rozhodnutí s volbou funkcionální podmínky vs obecné podmínky.

- lineární výrazy vs obecné výrazy

K zodpovězení této otázky nám pomůže analytická geometrie. Lineárními výrazy lze popsat jen určitou část objektů, protože nelze měřit vzdálenosti ani úhly. Obecnější výrazy jsou ale mnohem náročnější na řešení, a proto algoritmy, které dovedou tyto výrazy řešit, nejsou příliš rychlé.

2.3 Příklady algoritmů

Popisem vybraných algoritmů, které řeší omezující podmínky, se budeme zabývat na následujících stránkách. Omezíme se však jen na algoritmy, které řeší hierarchie podmínek, neboť ty přirozeně mohou řešit i systémy výhradně s povinnými podmínkami. Navíc v GUI je použití hierarchií naprosto přirozené, a to nejen kvůli použití edit a stay podmínek.

Nejdříve se seznámíme s algoritmy pracujícími nad funkcionálními podmínkami (DeltaBlue, SkyBlue, QuickPlan a Houria III), dále ukážeme jeden algoritmus, který se vypořádá s řešením numerických nerovností (Indigo), a nakonec předvedeme dva algoritmy, které dovedou řešit obecné lineární výrazy a navíc si poradí s cykly v grafu (Ultraviolet, Cassowary).

2.3.1 DeltaBlue

Algoritmus DeltaBlue [15] patří do kategorie algoritmů řešících hierarchické funkční podmínky, které mají vždy jen jednu výstupní proměnnou. Je založený na lokální propagaci, podmínky řeší použitím locally-graph-better (LGB) grafu, který je podobný locally-predicate-better komparátoru.

Definice 11: Řešící graf je LGB, pokud v něm neexistuje nesplněná podmínka, kterou by bylo možné splnit odstraněním nějakých slabších podmínek.

V práci [17] je ukázáno, jaký je vztah mezi LGB grafem a LPB komparátorem. Lze nalézt příklad, kdy řešení dle LGB grafu není řešením dle LPB komparátoru.

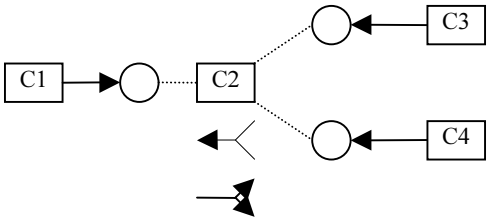
Pro ilustraci uvedeme jeden příklad. Neodpovídá sice omezením na typ podmínek pro algoritmus DeltaBlue (obsahuje totiž podmínku s více výstupními proměnnými), přesto je vhodné se s ním seznámit.

Příklad korektního LGB grafu, který neodpovídá LPB řešení

c1:	strong	PT=(2,3)
c2:	medium	PT = (X,Y)
c3:	weak	X=4
c4:	strong	Y=3

Proměnná PT představuje složenou datovou strukturu(dvojice čísel) a podmínka c2 realizuje převod mezi jednotlivými jednoduchými proměnnými (X aY) a složenou proměnnou PT.

LGB graf

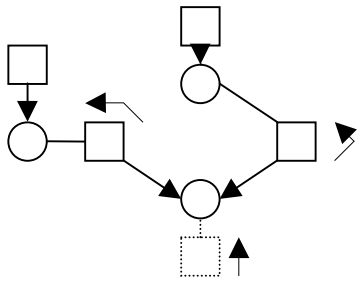


Graf splňuje vlastnost LGB, protože podmínku c2 není možné splnit. Použijeme-li jednu metodu, je v konfliktu s podmínkou c1, použití druhé metody vede ke konfliktu s podmínkou c4. Přesto odpovídající řešení [PT=(2,3), X = 4, Y = 3] není LPB. Správné LPB řešení totiž splňuje medium podmínku c2 a nesplňuje weak podmínku c3 [PT=(2,3), X = 2, Y = 3].

Algoritmus interně reprezentuje sadu podmínek jako graf systému, kde podmínky jsou označeny preferencemi. Z takového grafu se pak snaží vytvořit řešící graf, tj. pro každou splněnou podmínku vybrat některou její metodu, a tím určit proměnnou, která bude touto podmínkou určena. Řešící graf popisuje způsob, jakým je třeba hodnoty proměnných přepočítat, aby byly všechny splnitelné podmínky podle LGB komparátoru splněny. Řešení systému algoritmus dosáhne, pokud najde řešící graf, neboli orientovaný graf, ve kterém není cyklus, a do každé proměnné vede nejvýše jedna šipka, tedy každou proměnnou počítá nejvýše jedna metoda.

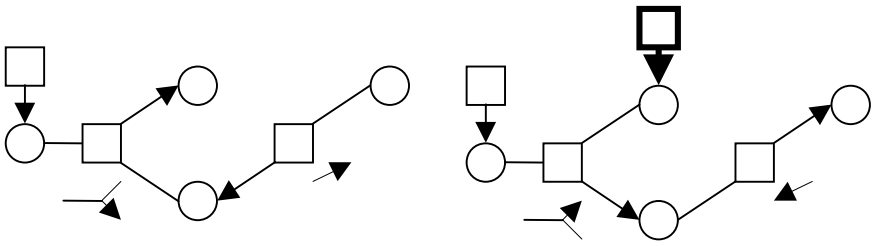
Ilustrace ukazuje příklad grafu, kde je možné najít konflikt metod, takže není řešícím grafem.

graf s konfliktem metod



Další dva obrázky jsou korektní řešící grafy a ukazují, jak se může řešící graf změnit, přidáme-li do systému novou podmínku. Vlevo je původní řešící graf a vpravo je nově vzniklý řešící graf. Přidávaná podmínka je na grafu vpravo označena tučně.

řešící grafy



DeltaBlue je inkrementální algoritmus umožňující rychlé přidání resp. ubrání podmínky.

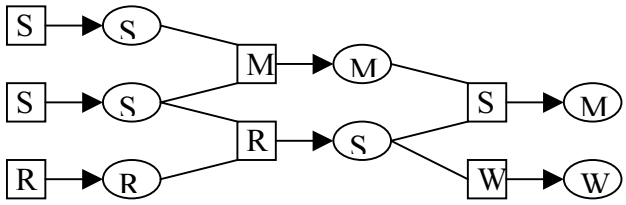
V případě přidání podmínky do systému mohou nastat dva případy. Buď lze vybrat některou metodu této podmínky tak, aby nebyla v konfliktu s žádnou jinou metodou řešícího grafu před přidáním podmínky, a přidání podmínky je pak triviální, nebo je nutné nejdříve provést určité změny v řešícím grafu, které mohou nakonec vést k tomu, že některá podmínka musí být ze systému odstraněna, aby bylo možno novou podmínku do systému zahrnout. Nemá-li být porušena locally-graph-better vlastnost systému podmínek, je možné ze systému odebrat pouze podmínku se slabší preferencí. Pokud toto není možné, novou podmínku nelze do systému zařadit. K nalezení vhodných změn řešícího grafu, které by umožnily přidání podmínky, je nutné vyzkoušet všechny možné varianty a vybrat z nich tu nejvýhodnější. To může být časově velmi náročné, proto je pro tento účel vhodné zavést pro proměnné tzv. *průchozí preference* (walkabout strength). Ty pro každou proměnnou indikují, jaká je preference nejslabší splněné podmínky, kterou by bylo nutné odebrat ze systému, aby se

proměnná stala volnou, neboli mohla být vypočítána metodou nové podmínky. Ohodnotíme-li takto všechny proměnné grafu, stačí pro přidání podmínky vybrat takovou její metodu, která má výstupní proměnnou s nejslabší průchozí preferencí, nicméně nižší, než je preference nové podmínky. Tuto výstupní proměnnou bude nyní určovat nová podmínka, je tedy nutné odebrat podmínku, která výstupní proměnnou určovala do této chvíle. V dalším kroku se pak pokusíme odebranou podmínku přidat zpět do systému. Toto rekurzivně provádíme do té doby, dokud pro přidání podmínky nebude nutno odebrat nějakou slabší podmínku. Pokud algoritmus dospěje do situace, kdy přidání nové podmínky je podmíněno odebráním podmínky stejně silné nebo silnější, není možné novou podmínku do systému zařadit.

Jestliže průchozí preference průběžně přepočítáváme v závislosti na změnách řešícího grafu, lze při operaci přidání resp. ubrání podmínky docílit výrazné redukce vyzkoušených možností řešícího grafu, než je nalezena jeho nová podoba.

Na obrázku je znázorněno ohodnocení řešícího grafu průchozími preferencemi. Písmeno v omezující podmínce odpovídá preferenci podmínky, zatímco v proměnné označuje průchozí preferenci této proměnné. Jednotlivá písmena R, S, M, W odpovídají po řadě preferencím required, strong, medium a weak.

graf s průchozími preferencemi



Další klíčovou vlastností tohoto algoritmu je schopnost kompilace výpočtu. V těch případech, kdy nedochází ke změnám v systému podmínek, ale je stanoven požadavek jen na změnu hodnot některých proměnných, umožňuje algoritmus generovat tzv. plány. Jde o posloupnost metod, jejichž postupné provedení zajistí, po změně daných proměnných, opětné splnění podmínek.

Algoritmus DeltaBlue díky své jednoduchosti a rychlosti při operacích, které jsou klíčové v interaktivní grafice, patří mezi efektivní algoritmy. Jeho hlavním omezením však je, že neumí řešit cykly podmínek. Pokud při tvorbě řešícího grafu narazí na orientovaný cyklus, ohlásí chybu.

Úplný popis algoritmu včetně pseudokódu a příkladů využití lze najít v práci [15].

2.3.2 SkyBlue

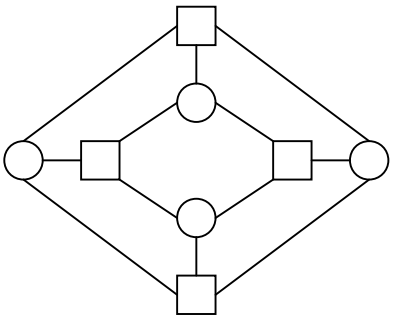
Posoudíme-li algoritmus DeltaBlue s jeho hlavními omezeními, lze o algoritmu SkyBlue [14] prohlásit, že byl vytvořen proto, aby odstranil tato omezení. Podporuje používání metod s více výstupními proměnnými a umožňuje cykly podmínek. Vychází plně z filozofie algoritmu DeltaBlue, jen s některými úpravami, které jsou nezbytné s ohledem na jeho širší použití.

Zdá se, že podpora podmínek s více výstupními proměnnými pro obecné použití algoritmu není nezbytná, protože ve většině případů lze podmínku s více výstupními podmínkami rozdělit na několik podmínek s jednou výstupní proměnnou. Avšak jde o rozšíření, které může jednak zjednodušit deklarativní zápis systému, jednak v některých situacích zabránit vzniku cyklu v grafu.

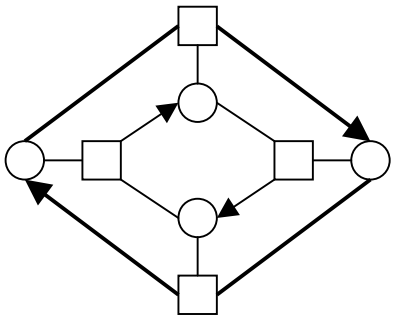
Jako příklad si uvedeme převod mezi kartézským a polárním systémem souřadnic. Chceme-li v systému bod reprezentovat jak v kartézských tak v polárních souřadnicích, je nutné zavést do systému podmínky, které mezi těmito souřadnicemi udržují konzistenci. Použijeme-li však čtveřici podmínek (každá podmínka má tři možné metody vždy s jednou výstupní proměnnou): $X=R \cos \alpha$, $Y=R \sin \alpha$, $R=\sqrt{(X^2+Y^2)}$, $\alpha=\arctan(Y/X)$, jde o soustavu čtyř rovnic se čtyřmi proměnnými. Taková soustava je typickým příkladem sady podmínek s cyklem. Tomuto problému se lze snadno vyhnout, využijeme-li podmínku se dvěma metodami, které mají dvě výstupní proměnné: $(X,Y) \leftarrow (R\cos \alpha, R\sin \alpha)$ a $(R,\alpha) \leftarrow (\sqrt{(X^2+Y^2)}, \arctan(Y/X))$.

Na obrázku vlevo je k nahlédnutí graf výše uvedeného systému čtyř omezujících podmínek, jejichž metody mají jednu výstupní proměnnou, a vpravo je pak jeden orientovaný graf systému. Tučně je v něm zvýrazněn jeden z cyklů.

graf systému



orientovaný graf systému s cyklem



Tím, že algoritmus podporuje metody s více výstupními proměnnými a umožňuje orientované cykly podmínek, bylo nutné kvůli zachování efektivnosti algoritmu při přidávání

a ubírání podmínek provést změnu v definici průchozích preferencí. V algoritmu SkyBlue je tedy průchozí preference definována jako dolní hranice preference nejslabší podmínky v řešícím grafu, kterou by bylo nutné vyřadit, aby proměnná mohla být vypočtena novou podmínkou. Tato definice umožňuje algoritmu při vyhledávání nejvhodnějšího tvaru řešícího grafu po přidání nové podmínky vynechat metody, které mají alespoň jednu výstupní proměnnou se stejnou nebo vyšší preferencí, než jakou má nová podmínka. Průchozí preference zde tedy nemohou použití backtrackingu při vyhledávání vhodného řešení zabránit, ale mohou jeho použití alespoň výrazně omezit.

Jak již bylo řečeno, algoritmus SkyBlue je schopen řešit případy, kdy se v řešícím grafu objeví orientovaný cyklus podmínek. To znamená: pokud na takový cyklus narazí, neskončí svoji činnost, nýbrž korektně zpracuje všechny podmínky vně tohoto cyklu. Samotné podmínky cyklu ovšem přímo řešit neumí. Algoritmus je ale otevřený, takže je možné řešení těchto podmínek předat nějakému jinému algoritmu, který si s nimi již dovede poradit.

Podrobnější informace o algoritmu SkyBlue včetně jeho pseudokódu lze nalézt v práci [14].

2.3.3 QuickPlan

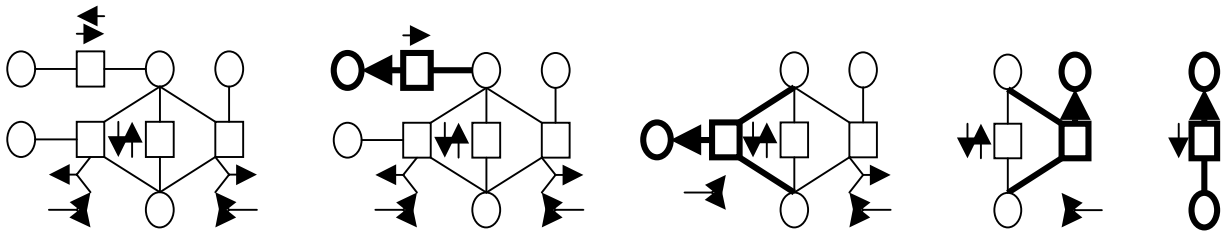
Algoritmus QuickPlan [16] patří do stejné kategorie algoritmů jako SkyBlue, podporuje tedy funkcionální podmínky s více výstupy. Stejně jako SkyBlue vychází z grafové reprezentace systému omezujících podmínek. Oba algoritmy se shodují i v tom, že se skládají ze dvou fází. V první etapě, tzv. *plánování*, algoritmus pro jednotlivé podmínky vybírá metody tak, aby byla hierarchie podmínek co nejlépe splněna podle LGB komparátoru. Ve druhé fázi, tzv. *exekuce*, dochází k výpočtu hodnot proměnných realizací jednotlivých vybraných metod ve zvoleném pořadí.

V následujícím odstavci si předvedeme základní myšlenku jednodušší neinkrementální varianty plánovací fáze.

K výběru metod slouží postup propagace stupňů volnosti. V grafu systému jsou vyhledány nejdříve všechny proměnné *v*, které jsou v grafu spojeny pouze s jedinou podmínkou *c*, a zároveň jsou výstupem některé metody *m* podmínky *c*. Tyto proměnné se nazývají volné. Pro podmínku *c* potom vybereme metodu *m*. Tím je určena orientace hran vedoucí z podmínky *c*. Následně podmínku *c* spolu s proměnnými *v* vyřadíme z grafu. Na nově vzniklý graf rekurzivně aplikujeme stejný postup.

Na obrázku je zleva doprava znázorněn výše popsaný postup. Eliminované podmínky a proměnné jsou označeny tučně.

postupná eliminace podmínek

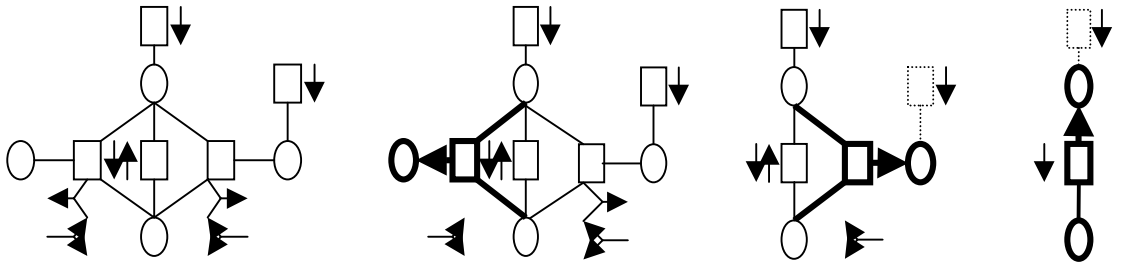


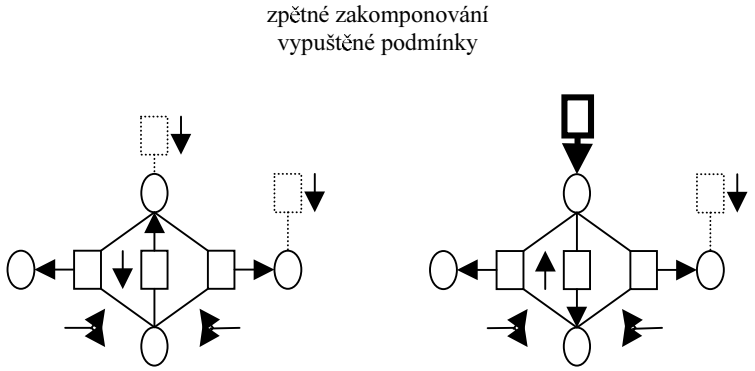
Zmíněný postup lze opakovat do té doby, dokud buď graf nebude obsahovat žádnou podmínku, nebo nastane situace, kdy nelze vybrat žádnou volnou proměnnou. První případ odpovídá úspěšnému nalezení metod pro všechny podmínky. Ve druhém případě je nutné nějakou podmínku z grafu vypustit, tj. nebude splněna. Tak snížíme stupeň volnosti proměnných svázaných s touto podmínkou a je možné se znovu pokusit nalézt volné proměnné. Při vypouštění podmínky přirozeně respektujeme hierarchii podmínek, a proto v grafu vybíráme podmínku s nejslabší preferencí.

Uvedený postup sice zaručuje, že algoritmus dospěje při eliminaci až ke grafu, který neobsahuje žádnou podmínku. Nicméně k tomu, abychom mohli zkonstruovat řešící graf, je ještě nutné pokusit se o zpětné zakomponování dříve vypuštěných podmínek. Toho lze docílit tím, že u některých podmínek buď změníme jejich vybrané metody, nebo vypustíme některou podmínku se slabší preferencí.

Následující ilustrace ukazuje proces eliminace podmínek včetně vypuštění podmínky, aby v grafu vznikly nové volné proměnné. V posledním grafu je znázorněno také zpětné zakomponování dříve odebrané podmínky.

eliminace podmínek
a vypuštění podmínky





To, v čem se QuickPlan liší od algoritmu SkyBlue, a proč je tedy dobré se o něj podrobněji zajímat, je záruka, že algoritmus nalezne vždy acyklické řešení, pokud soustava podmínek alespoň jedno takové acyklické řešení má. Je tedy schopen sám řešit širší spektrum systémů podmínek než algoritmus SkyBlue. Ovšem v případě, kdy jediné řešení systému bude cyklické, i on potřebuje na tyto podmínky použít jiný, specializovaný řešič.

Podrobnější popis celého algoritmu včetně jeho inkrementální verze najdeme v práci [16].

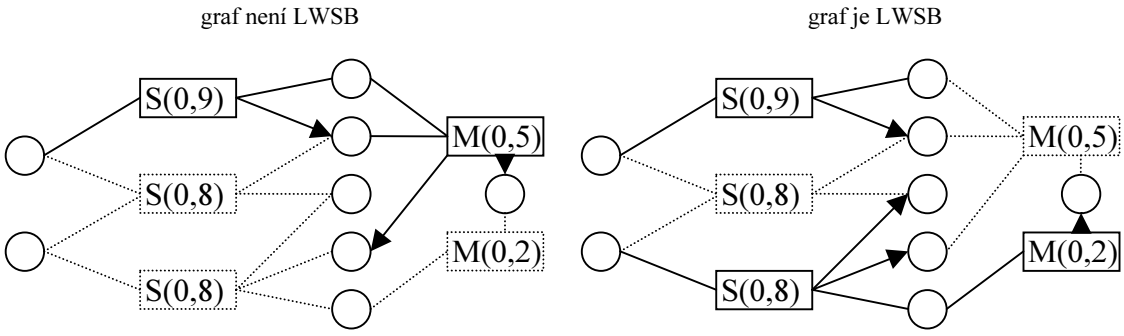
2.3.4 Houria III

Algoritmus Houria [8] má stejnou specifikaci jako předchozí algoritmy, podporuje tedy pouze funkcionální podmínky s více výstupními proměnnými. Hlavními rozdíly tohoto algoritmu jsou: 1. nepoužívá locally-graph-better komparátor, jako tomu je u algoritmů SkyBlue i QuickPlan, ale využívá weighted-sum-predicate-better (WSPB), 2. vrací všechna nejlepší řešení zadané hierarchie. Všechny podmínky mají kromě své preference také svoji váhu, která umožňuje ještě na úrovni podmínek se stejnou preferencí rozlišovat jejich důležitost. I zde je vnitřní reprezentace podmínek realizována pomocí grafu, tentokrát však je řešící graf definován jako tzv. *lexicographic-weighted-sum-better* (LWSB) graf. Jde vlastně o variantu standardního řešícího grafu s tím, že použitým komparátorem je WSPB.

Význam slova *lexikografický* nám dává návod, jak porovnat dva grafy G1 a G_2. Spočteme součty vah nesplněných podmínek na jednotlivých úrovních preference pro oba grafy. Tyto součty je nutno počítat do té doby, dokud se pro oba grafy vypočtené hodnoty shodují. V okamžiku, kdy součet vah nesplněných podmínek na určité úrovni pro jeden graf, např. G_1 je ostře menší, lze prohlásit, že je tento graf G_1 lepší dle WSPB než G_2.

Definice 12: Řešící graf G_1 systému H je *lexicographic-weighted-sum-better* (LSWB) než jiný řešící graf G_2 systému H, jestliže pro každý stupeň důležitosti *i* až do úrovně *k-1* je součet vah nesplněných podmínek úrovně *i* v grafu G_1 roven součtu vah nesplněných podmínek úrovně *i* v grafu G_2 a pro stupeň důležitosti *k* je součet v grafu G_1 ostře menší.

Na obrázku je příklad dvojice grafů, které reprezentují stejnou hierarchii. Graf vpravo je příkladem LWSB grafu, zatímco graf vlevo není LWSB. Nesplněné podmínky jsou označeny tečkovaně. Písmeno S udává preferenci strong a písmeno M preferenci medium. Váhy podmínek jsou uvedeny u každé podmínky v závorce.



Důvodem, proč graf vpravo je lepší dle komparátoru WSPB než graf vlevo, je to, že součet vah nesplněných podmínek na úrovni strong u grafu vlevo je 1,6 (druhá a třetí podmínka) zatímco u grafu vpravo má jediná nesplněná podmínka váhu 0,8. Protože preference medium je nižší než preference strong, není už třeba u preference medium hodnotit váhy nesplněných podmínek.

Při hledání řešení algoritmus postupně přidává jednotlivé podmínky do systému a udržuje si všechny možné řešící grafy a váhy těchto grafů podle WSPB komparátoru. Po přidání všech podmínek jsou všechny řešící grafy setříděny podle svých vah, aby se zjistilo, který z nich je nejlepší vzhledem ke komparátoru WSPB.

Protože pro přidání podmínky využívá algoritmus dosud nalezeného řešení, lze ho považovat za inkrementální. Na rozdíl od algoritmu SkyBlue a QuickPlan však nepodporuje cykly v grafech podmínek.

Podrobnější informace o algoritmu Houria III lze najít v [8].

2.3.5 Indigo

Všechny předchozí algoritmy měly jednu společnou vlastnost: byly určeny k řešení systémů funkcionálních podmínek. V grafických uživatelských rozhraních však mnohdy s těmito

podmínkami nevystačíme. Příkladem mohou být podmínky pro vyjádření vztahu mezi objekty „nad“ , „vlevo“, ale i „uvnitř“. Převedeme-li tyto výrazy do formálního tvaru, výsledkem jsou lineární nerovnice. Existují sice numerické postupy, které umí soustavy lineárních nerovnic řešit, ale jejich aplikování v hierarchiích vede k časově náročným výpočtům. Bylo tedy nutné hledat vhodný algoritmus, který by dovedl tento druh podmínek řešit efektivně.

Indigo [3,4] je právě takový algoritmus. Vychází z myšlenky lokální propagace, kterou však bylo nutné modifikovat pro nerovnosti. Existují dvě verze algoritmu. U první verze je místo konkrétní hodnoty proměnné (viz 1.5) jako tomu bylo u algoritmů zpracovávajících funkcionální podmínky (viz 2.3.1-2.3.4) propagována horní a dolní mez hodnoty proměnné, neboli interval možných hodnot. Jde o slabší a jednodušší variantu algoritmu. Druhá, silnější varianta propaguje pro každou proměnnou množinu nepřekrývajících se intervalů.

Podmínky algoritmus zpracovává v pořadí od nejsilnějších po nejslabší, přičemž zpracování každé podmínky vede k omezení možného rozsahu hodnot jejích proměnných. Tyto změny rozsahu hodnot proměnných je pak třeba promítnout i do ostatních podmínek, ve kterých se změněné proměnné nacházejí a které ještě nemají své proměnné jednoznačně určené (tzv. *aktivní podmínky*). Díky tomu, že systém vyžaduje pro každou proměnnou implicitně stay podmínku, která proměnnou svazuje s její předchozí hodnotou, lze říci, že v závěru zpracování má každá proměnná jednoznačně určenou hodnotu.

Dalším rozdílem oproti algoritmům jako je SkyBlue nebo QuickPlan je použití jiného komparátoru. U podmínek typu $x \leq a$ (obecně nerovností) se totiž zdá přirozené sledovat nejenom, zda ji algoritmus splnil či nikoliv, ale i míru této splnitelnosti. Například máme-li podmínku $x \leq 30$ jako součást většího systému podmínek a algoritmus v jednom řešení nabízí za *x* hodnotu 40, je jistě takové řešení přijatelnější, než kdyby navrhoval za *x* hodnotu 50, byť obě hodnoty podmínku nesplňují. Algoritmus Indigo využívá tedy komparátoru locally-error-better a chybovou funkcí pak je vzdálenost dvou reálných čísel.

V práci [3] je dokázáno, že obě varianty algoritmu pracují korektně, tedy nalezne-li algoritmus řešení, je vždy správné. Slabá varianta však v některých případech není schopná řešení nalézt. Následující příklad ukazuje takovou množinu podmínek, o které slabá varianta algoritmu prohlásí, že ji nedovede vyřešit. Důvodem, proč slabá varianta některé množiny nevyřeší, je to, že předpokládá: pokud není podmínka splněna, určí všem svým proměnným jednoznačné hodnoty. Nesplněnou podmínku proto není pak třeba zařazovat mezi aktivní podmínky, a slabá varianta tedy předpokládá, že všechny aktivní podmínky jsou zároveň splnitelné.

Příklad příliš obtížné množiny podmínek pro slabou variantu algoritmu Indigo

required	$-1 \leq a \leq 1$
required	$-1 \leq b \leq 1$
strong	$a*b = 2$
weak	$a = 0{,}5$

Algoritmus nejdříve omezí domény proměnných *a* a *b* na intervaly [-1;1]. Při zpracování podmínky a*b=2 není možné podmínku splnit, hodnoty proměnných však nelze ani omezit (interval [-1,1] bychom museli rozdělit na dva intervaly [-1;-1] a [1;1]). Protože hodnoty proměnných podmínky nejsou určeny jednoznačně, je podmínka zařazena mezi aktivní podmínky. Tím, že není splněná, způsobí tím v dalším běhu algoritmu chybu. Při zpracování podmínky a = 0,5 omezíme hodnotu proměnné *a* na interval [0,5;0,5] a propagací přes podmínku a*b=2 dostáváme pro *b* interval [1,1]. Obě proměnné jsou tedy omezeny na konkrétní hodnotu, přesto aktivní podmínka a*b=2 není splněna, a chyba jejího nesplnění není nejnižší možná (pokud bychom zvolili a=b=1, byla by chyba menší). Důvod, proč v tomto příkladě slabá varianta zklamala, je ten, že při zpracování podmínky a*b=2 nemohla omezit možné hodnoty obou proměnných, a tak při zpracování weak podmínky a=0,5 mohla být tato podmínka splněna, ačkoli by tomu tak být nemělo.

Uveďme si ještě jeden příklad. V něm se pokusíme demonstrovat postup, jak danou množinu podmínek zpracuje slabší varianta algoritmu.

Příklad: Je dána tato hierarchie pro reálné proměnné

c1:	required	$x \leq 100$
c2:	required	$y \leq 75$
c3:	required	$x + y = 25 + u$
c4:	required	$u - v = 75$
c5:	strong	$v \geq 50$
c6:	medium	$y = 60$
c7:	weak	$x = 125$
c8:	weak	$y = 100$
c9	weak	$u = 150$
c10:	weak	$v = 75$

v tabulce č. 1 je pak znázorněn postup, jak algoritmus Indigo hierarchii vyřeší:

tabulka č. 1

operace s podmínkou	x	y	u	v	Vysvětlení
	(-∞,+∞)	(-∞,+∞)	(-∞,+∞)	(-∞,+∞)	počáteční meze proměnných
c1	(-∞,100]	(-∞,+∞)	(-∞,+∞)	(-∞,+∞)	
c2	(-∞,100]	(-∞,75]	(-∞,+∞)	(-∞,+∞)	
c3	(-∞,100]	(-∞,75]	(-∞,150]	(-∞,+∞)	
c4	(-∞,100]	(-∞,75]	(-∞,150]	(-∞,75]	
c5	(-∞,100]	(-∞,75]	(-∞,150]	[50,75]	
c4	(-∞,100]	(-∞,75]	[125,150]	[50,75]	propagace z podmínky c5

c3	[75,100]	[50,75]	[125,150]	[50,75]	propagace z podmínky c5
c6	[75,100]	[60,60]	[125,150]	[50,75]	
c3	[90,100]	[60,60]	[125,135]	[50,75]	propagace z podmínky c6
c4	[90,100]	[60,60]	[125,135]	[50,60]	propagace z podmínky c6
c7	[100,100]	[60,60]	[125,135]	[50,60]	nelze splnit, je pouze minimalizována chyba
c3	[100,100]	[60,60]	[135,135]	[50,60]	propagace z podmínky c7
c4	[100,100]	[60,60]	[135,135]	[60,60]	propagace z podmínky c7
c8,c9,c10	[100,100]	[60,60]	[135,135]	[60,60]	nelze splnit

Algoritmus Indigo má dvě slabá místa: 1. vyžaduje acyklické zadání podmínek, 2. nepodporuje inkrementalitu. Je tedy nutné při jakékoliv změně systému podmínek, včetně změny pouhé konstanty, celý systém přepočítat. Nepřehlédnutelnou okolností je také výrazně vyšší obtížnost implementace silnější verze algoritmu, než jak je tomu u slabší varianty.

Tato omezení však, zdá se, není možné jednoduchým způsobem obejít. Algoritmus tak sám o sobě není prakticky vhodný k používání, ale může být využit jako subsystém do jiného mocnějšího řešiče.

V pracích [3] a [4] je algoritmus podrobně popsán.Práce obsahuje důkazy korektnosti a úplnosti silné verze. Je zde také uveden pseudokód jeho slabší varianty včetně implementačních detailů.

2.3.6 Ultraviolet

Algoritmus, kterému se budeme nyní věnovat, se jmenuje Ultraviolet [5,6]. Částečně odstraňuje omezení algoritmu Indigo, pro určité množiny podmínek totiž umí vyřešit cykly.

V pravém smyslu nejde o nový algoritmus, ale jen o nadstavbu nad algoritmy SkyBlue, Indigo, Purple a DeepPurple. Algoritmus nejprve rozdělí graf podmínek na oblasti, které umí řešit jednotlivé specializované řešiče. Funkcionální podmínky potom zpracuje SkyBlue, nerovnosti Indigo, cykly lineárních rovnic Purple a nakonec cykly lineárních nerovností vyřeší DeepPurple. Komunikaci mezi algoritmy pak zajišťují proměnné, které jsou sdíleny více oblastmi, a je tedy nutné, pokud jeden řešič hodnotu sdílené proměnné změní, aby na to reagovaly i ostatní subsystémy, kterým se tato změna promítne do jejich oblastí.

Algoritmus Ultraviolet pokrývá velkou škálu systémů podmínek, které je schopen řešit, je však implementačně poměrně dost náročný. Tím, že popisovaná verze algoritmu je závislá na algoritmu Indigo, nesplňuje také vlastnost inkrementality.

Další informace o algoritmu Ultraviolet lze čerpat z [5] a [6].

2.3.7 Cassowary

Cassowary [1] je algoritmus, který umí řešit systémy lineárních rovnic a nerovnic inkrementálním způsobem. Vypořádá se s cyklickým zadáním množiny podmínek, a nejen to, umí takové zadání vyřešit efektivně. Existují rovněž varianty algoritmu, které pracují nad různými globálními komparátory. Podmínky mohou být stejně jako u algoritmu Houria opatřeny kromě preference také svojí váhou. Pro výběr nejlepšího řešení lze proto použít např. komparátor weighted-sum-better. Tím, že se Cassowary svým přístupem výrazně liší od všech uvedených algoritmů, považujeme za vhodné se s ním seznámit podrobněji.

Algoritmus vychází ze standardního simplexového algoritmu, který řeší následující problém. Pro *n* reálných proměnných, jejichž hodnota je omezena na nezáporná čísla, je definována sada *m* lineárních rovnic a nerovnic nad těmito proměnnými a dále lineární funkce $f(x_1,...,x_n) \rightarrow R$, které říkáme *cílová funkce*. My hledáme takové řešení této soustavy mezi všemi jejími řešeními, které minimalizuje (popř. maximalizuje) hodnotu funkce *f*.

Bohužel původní varianta simplexového algoritmu pro využití v grafických uživatelských rozhraních není vhodná hned z několika důvodů. Prvním důvodem je to, že nesplňuje základní vlastnosti efektivního algoritmu - inkrementalitu. Jak již bylo zmíněno, při řešení omezujících podmínek v grafických uživatelských rozhraních je nutné rychle přepočítávat velmi podobné soustavy, a tak zajistit co nejrychlejší odezvu k uživateli. Za druhé: cílová funkce *f* musí být v původní variantě simplexového algoritmu lineární. V případě omezujících podmínek však tato cílová funkce vyjadřuje komparátory, což často vede k její nelinearitě. Např. u komparátorů locally-error-better a weighted-sum-better je „téměř lineární“, ale u least-square-better je kvadratická. Poslední úpravu původního algoritmu vyžaduje také požadavek na nezápornost proměnných, což v grafických systémech nemusí být vždy postačující.

> Jako příklad si vezměme následující situaci. Máme proměnné x_l, x_r a x_m, přičemž požadujeme, aby proměnná x_m měla hodnotu rovnou aritmetickému průměru hodnot proměnných x_l a x_r. Všechny proměnné mohou nabývat hodnot přirozených čísel mezi 0 a 100. Požadujeme navíc, aby proměnná x_l měla hodnotu alespoň o 10 menší než proměnná x_r. Tím, že nutně platí $x_l < x_m < x_r$, lze zjednodušit množinu omezujících podmínek následně:
>
> $$\begin{aligned} 2\,x_m &= x_l + x_r \\ x_l + 10 &\leq x_r \\ x_r &\leq 100 \\ 0 &\leq x_l \end{aligned}$$
>
> Chceme najít takové řešení, které navíc minimalizuje $x_m - x_l$.

Varianta simplexového algoritmu, kterou se nyní budeme podrobněji zabývat, odstraňuje výše uvedené omezení, a proto je možné ji v grafických aplikacích použít. Hlavní myšlenka spočívá v udržování soustavy podmínek ve *vhodné řešící formě* (basic feasible solved form), v jedné z tzv. rozšířených simplexových forem.

Definice 13: Systém je v *rozšířené simplexové formě*, jestliže soustava podmínek je ve tvaru C_u, C_s a C_i, přičemž C_u a C_s jsou konjunkcí lineárních rovností a C_i je tvaru {x≥0, přičemž x je proměnnou z C_s}, a cílová funkce f je lineární výraz nad proměnnými z C_s. V C_u jsou ty rovnosti, které obsahují tzv. nevázané proměnné, tzn. proměnné, které nejsou omezeny na nezáporné hodnoty.

V první řadě je nutné vyrovnat se s tím, že rozšířená simplexová forma vyžaduje pouze rovnosti. Chceme-li tedy, abychom mohli používat i nerovnosti, je nutné každou nerovnost nahradit odpovídající rovností. K tomuto účelu rozšíříme množinu proměnných o tzv. *slabé proměnné* neboli o pomocné proměnné. U těchto proměnných není nutné, aby byly přístupné mimo algoritmus. Potom lze nerovnost ve tvaru e≤r (*e* je výraz a *r* je reálné číslo) nahradit rovnostmi e+s=r a s≥0, kde *s* je nová slabá proměnná.

Aplikujeme-li toto pravidlo na náš příklad, dostáváme soustavu

$2\,x_m = x_l + x_r$
$x_l+10+s_1 = x_r$
$x_r+s_2 = 100$
$0 \le x_l,s_1,s_2$ (s_1, s_2 jsou nově zavedené slabé proměnné).

V dalším kroku rozdělíme všechny rovnosti mezi C_u a C_s. Zpočátku všechny rovnosti umístíme do C_s a postupně pomocí Gauss-Jordanovy eliminace přemístíme nevázané proměnné do C_u. V každém kroku vybereme jednu rovnost z C_s, např. $x_1=e_1$, která obsahuje nevázanou proměnnou *u*, tuto proměnnou vyjádříme, tzn. získáme rovnost u = e, kde *e* je nějaký výraz, všechny další výskyty proměnné *u* v C_s,C_u i v *f* nahradíme výrazem *e*, rovnost $x_1=e_1$ odstraníme z C_s a do C_u přidáme rovnost u = e. Tyto kroky opakujeme do té doby, dokud je v C_s nějaká nevázaná proměnná. Tak lze obecnou soustavu převést do rozšířené simplexové formy.

Po skončení tohoto kroku vypadá náš příklad takto:

$x_m = 50 + 1/2x_l - 1/2s_2$
$x_r = 100 - s_2$
$x_l + 10 + s_1 = 100 - s_2$
$0 \le x_l,s_1,s_2$

a výraz, který chceme minimalizovat má tvar $50 - 1/2x_l-1/2s_2$. První dvě rovnosti soustavy patří do množiny C_u a třetí rovnost je prvkem množiny C_s.

Definice 14: Soustava podmínek je ve *vhodné řešící formě*, jestliže všechny rovnosti lze napsat ve tvaru $x_0 = c + a_1x_1 + ... + a_nx_n$, přičemž x_0 se nevyskytuje v žádné další rovnosti ani v cílové funkci. Pokud jde o rovnost z C_s, musí *c* být navíc nezáporné. Potom proměnné x_0 nazýváme základní a ostatní proměnné (x_1 ..x_n) parametry. Triviálním řešením (nikoli však optimálním vzhledem k cílové funkci) takové řešené formy je pak $\{x_0 = c, x_1 = 0, x_n = 0\}$.

Pro obecné rovnosti z C_s není zaručeno, že je lze převést do tvaru, který požaduje předchozí definice, tedy do tvaru $x_0 = c + a_1x_1 + ... + a_nx_a$, přičemž c je nezáporné. Jak bude patrné v kapitole 2.3.7.3, algoritmus Cassowary interně reprezentuje soustavu rovnic tak, aby tento případ nemohl nastat.

Převedeme-li náš příklad do tvaru vhodné řešící formy, bude vypadat takto:

$$x_m = 50 + 1/2x_l – 1/2s_2$$
$$x_r = 100 – s_2$$
$$s_1 = 90\text{-}x_l\text{-}s_2,$$

přičemž požadujeme minimalizaci výrazu $50 – 1/2x_l – 1/2s_2$.
Triviálním řešením takové formy je $\{x_m \rightarrow 50, x_r \rightarrow 100, s_1 \rightarrow 90, x_l \rightarrow 0, s_2 \rightarrow 0\}$. Hodnota cílové funkce tohoto řešení je 50.

Pro danou soustavu může existovat více vhodných řešených forem. Je třeba tedy nyní najít mezi všemi řešícími formami tu optimální, tedy takovou, která minimalizuje hodnotu cílové funkce.

Nalezení optimálního řešení podmínek, které jsou v této vhodné řešící formě, odpovídá druhé fázi standardního simplexového algoritmu. Algoritmus v každém kroku prohledává sousední vhodné řešící formy, které vedou ke snížení hodnoty cílové funkce. Pokud taková sousední forma neexistuje, bylo dosaženo optima. Sousední formou míníme takovou formu, které lze dosáhnout záměnou základních a parametrických proměnných původní formy.

Snížení hodnoty výrazu cílové funkce $50 – 1/2x_l – 1/2s_2$ lze dosáhnout buď zvětšením hodnoty proměnné x_l, nebo proměnné s_2.Vybereme si například proměnnou x_l a pokusíme se ji změnit na základní, a tím docílit zvětšení její hodnoty, a protože je zatím parametrická, v řešení jí přiřazujeme hodnotu 0. Hodnotu proměnné x_l ale nelze zvětšovat do nekonečna, neboť tím bychom mohli porušit požadavek na kladnost jiných proměnných. Z třetí rovnosti ($s_1 = 90\text{-}x_l\text{-}s_2$) lze nahlédnout, že hodnotu x_l můžeme zvyšovat až do 90, potom už by proměnná s_1 nabývala záporných hodnot. Vyjádříme-li tedy z třetí rovnosti x_l, dostaneme $x_l = 90\text{-}s_1\text{-}s_2$ .Nahrazením všech výskytů x_l vyjádřeným výrazem dostáváme následující soustavu:

$$x_m = 95 – 1/2\ s_1 – s_2$$
$$x_r = 100 – s_2$$
$$x_l = 90 – s_1 – s_2,$$

přičemž cílová funkce f je rovna výrazu $5+1/2s_1$. Podíváme-li se nyní na cílovou funkci, je zřejmé, že k dosažení jejího minima vede dosazení za s_1 nulové hodnoty. Proměnnou s2 můžeme volit libovolně, protože hodnotu cílové funkce neovlivňuje.Protože jsou obě proměnné parametrické, zvolíme i s_2 rovné 0. Výše

uvedená forma je tedy optimální a řešení, které dává je $\{x_m \rightarrow 95, x_r \rightarrow 100, x_l \rightarrow 90, s_1 \rightarrow 0, s_2 \rightarrow 0\}$. Hodnota cílové funkce tohoto řešení je 5.

Předvedli jsme základní myšlenku, na které je algoritmus postaven. Není smyslem této práce zabíhat do přílišných podrobností vlastního algoritmu, proto se nyní budeme věnovat jen některým jeho klíčovým principům.

2.3.7.1 Přidání a ubrání podmínky

Jak bylo uvedeno v úvodu kapitoly, algoritmus Cassowary je inkrementální. Této vlastnosti lze využít i při hledání počáteční vhodné řešící formy. Při startu ponecháme množinu podmínek prázdnou a požadované podmínky přidáváme do systému postupně. Proces vlastního přidávání a ubírání podmínek není příliš zajímavý. Za povšimnutí ale stojí myšlenka modifikace rovností pro snadné odebírání podmínek.

Přidáním jedné podmínky přibude jedna řádka v soustavě. V průběhu výpočtu však může dojít k tomu, že informace, kterou přinesla tato podmínka, může být obsažena ve více řádcích soustavy, protože při hledání optimální řešící formy vyjadřujeme proměnné z některých rovností a dosazujeme je do ostatních rovností. Pokud tedy chceme tuto podmínku ze soustavy odebrat, je nutné brát ohled na všechny řádky, které odebíraná podmínka ovlivňuje. Myšlenka, jak snadno v průběhu výpočtu evidovat vliv podmínky, spočívá v zavedení tzv. *označovací proměnné* pro každou podmínku. Vliv podmínky v řádcích soustavy je pak určen výskytem její označovací proměnné. Pro některé typy podmínek lze jako označovací proměnné využít již zavedené pomocné proměnné. Např. u nerovností lze využít jejich slabých proměnných. U podmínek, které nejsou povinné (non-required), lze využít chybové proměnné (viz níže). Pouze u povinných rovností je nutné zavést zvláštní vázanou proměnnou, která nesmí sloužit jako základní (musí být vždy parametrická -má vždy hodnotu 0).

2.3.7.2 Realizace nepovinných podmínek

Chceme-li v systému nějak zohlednit hierarchii podmínek, je nutné v závislosti na použitém komparátoru pro každou nepovinnou podmínku přidanou do systému modifikovat cílovou funkci. Snahou je vždy minimalizovat odchylku od požadované skutečnosti.

Před tím než ukážeme příklad cílové funkce, je nutné se zmínit o jednom implementačním detailu. V kapitole 1.2 je vysvětlen rozdíl mezi preferencemi a vahami podmínek. Pokud jsme schopni při implementaci algoritmu zajistit, aby při hledání řešení podmínek nebyl porušen požadavek na respektování hierarchie, lze pro realizaci preferencí

použít váhy. Jednotlivým stupňům preference tak přiřadíme váhy, které jsou od sebe dostatečně vzdáleny. Takže například preferenci strong nahradíme váhou 10 000, preferenci medium váhou 100 a preferenci weak určíme váhu 1. Poslední věcí, na kterou je třeba při realizaci preferencí pomocí vah pamatovat, jsou povinné podmínky, které musí být vždy splněny. Musí být tedy zpracovány samostatně.

Algoritmus Cassowary používá právě realizaci preferencí pomocí vah. Přidáme-li tedy do systému následující podmínky: (strong) x = 20, (medium) x + y = 10; (weak) y = 0, pak cílovou funkci musíme rozšířit o výraz: s*e(x,20)+m*e(x+y,10)+w*e(y,0), kde s,m,w jsou váhy jednotlivých stupňů hierarchie podmínek a funkce e(x,y) vyjadřuje chybu v závislosti na použitém komparátoru. Pro komparátor weighted-sum-better je funkce e(x,y) definována jako |x-y|, pro least-squares-better jako (x^2-y^2). V obou případech však cílová funkce není lineární.

2.3.7.3 Kvazi-lineární optimalizace algoritmu Cassowary

Abychom se mohli vypořádat s nelinearitou cílové funkce zmíněnou v předchozím odstavci, je nutné změnit reprezentaci všech rovností. Zavedeme nový typ vázaných proměnných, tzv. *chybové proměnné*. Každou rovnost z C_u a C_s (x = v , kde *x* je proměnná a *v* výraz) rozšíříme o výraz d_{x+} - d_{x-} (tzn. do tvaru x = v + d_{x+} - d_{x-}), kde d_{x+} a d_{x-} jsou nové chybové proměnné a vyjadřují odchylku hodnoty proměnné *x* od hodnoty výrazu *v*. Pokud je podmínka splněna, obě proměnné d_{x+} i d_{x-} nabývají hodnoty 0. Je-li hodnota *x* větší než hodnota *v*, je d_{x+} kladná a d_{x-} nulová, naopak je-li hodnota *x* menší než hodnota *v*, je d_{x-} kladná a d_{x+} nulová.

Abychom docílili splnění podmínky, vyžadujeme, aby obě proměnné d_{x+} i d_{x-} nabývaly hodnoty 0. Je nutné tedy je zařadit do cílové funkce s takovým koeficientem, který odpovídá váze podmínky.

2.3.7.4 Edit podmínky v algoritmu Cassowary

Algoritmus je připraven na ty situace, v nichž je nutné zvládnout rychlé znovuvyřešení systému podmínek, pokud proběhly jen malé změny v systému (reprezentované edit podmínkami). Dosahuje toho ve dvou fázích. Nejprve je třeba modifikovat řešící formu tak, aby odpovídala novým podmínkám. Následně je pro tuto modifikovanou soustavu zapotřebí přepočítat řešení. Modifikaci řešící formy provedeme takto. Je-li původní edit podmínka ve tvaru x = a+d_{x+}-d_{x-} a požadujeme-li ji modifikovat do tvaru x = b + d_{x+} - d_{x-}, musíme ve všech řádcích řešící formy, kde se vyskytují proměnné d_{x+} a d_{x-} (vždy jsou v páru, protože mohou být pouze parametrické a jsou tedy dosazovány ve výrazech vždy spolu), výraz d_{x+} - d_{x-}

nahradit výrazem b - a + d_{x+} - d_{x-}. Ve druhé fázi je pak nutné obnovit vhodnou řešící formu, která mohla být narušena tím, že sloučením některých konstant vznikly záporné hodnoty.

Algoritmus Cassowary lze považovat za velmi rychlý řešič hierarchického systému lineárních rovnic a nerovnic. Jak vyplývá z předchozího textu, splňuje všechny základní požadavky pro možnost praktického využití.

Popis implementace algoritmu Cassowary lze najít v práci [1].

2.4 Shrnutí

Všechny algoritmy, které jsme zde představili, řeší hierarchické systémy omezujících podmínek. Podporují různé typy podmínek a k řešení systémů používají širokou škálu postupů. V následující tabulce proto nejdůležitější vlastnosti každého algoritmu přehledně shrneme, abychom v další části práce mohli snadno vybrat algoritmus, který použijeme pro vlastní realizaci grafického editoru.

tabulka č. 2

Jméno algoritmu	**Typy podmínek**	**Komparátor**	**Cykly**
DeltaBlue	funkcionální s jedním výstupem	L-graph-better	ne
SkyBlue	funkcionální s více výstupy	L-graph-better	podporuje
QuickPlan	funkcionální s více výstupy	L-graph-better	ne, ale najde acyklické řešení
Houria III	funkcionální s více výstupy	Weighted-sum-predicate-better	ne, ale najde acyklické řešení
Indigo	matematické rovnosti a nerovnosti	L-error-better	ne
Ultraviolet	lineární rovnosti a nerovnosti	L-error-better	ano
Cassowary	lineární rovnosti a nerovnosti	Weighted-sum-better	ano

Algoritmy Indigo a Ultraviolet jsou algoritmy, které nepodporují inkrementalitu. QuickPlan a Houria III sice nedovedou řešit případy, kdy v grafu řešení je cyklus, ale pokud lze pro daný systém najít řešení, které má acyklický graf řešení, tak jej najdou. Ultraviolet umí obecně řešit lineární rovnosti a nerovnosti, navíc však vyřeší i případy, kdy v části systému jsou funkcionální podmínky nebo obecné nerovnosti. To ale pouze v tom případě, že jsou tyto části acyklické.

3 Část třetí – implementace grafického editoru

Třetí část je věnována vlastní implementaci grafického editoru pracujícího nad omezujícími podmínkami. Nejdříve přesně specifikujeme, s jakými omezujícími podmínkami budeme pracovat, na základě těchto požadavků pak vybereme vhodný podkladový algoritmus, který tyto omezující podmínky řeší. Dále popíšeme návrh vlastního grafického editoru. Stručně se také seznámíme s některými implementačními detaily a jednotlivými moduly programu. V jednotlivých přílohách lze najít doplňující informace: 1. o způsobu, jakým je podkladový algoritmus využíván a jak je možné jej v budoucnosti nahradit jiným vhodným algoritmem, 2. o možnosti rozšiřovat systém o nové grafické objekty, 3. o uživatelském ovládání grafického editoru. Součástí práce je také elektronická verze referenčního seznamu všech tříd a metod implementace grafického editoru.

3.1 Omezující podmínky v grafickém editoru

Jak již bylo řečeno, v grafických uživatelských rozhraních pro popis grafických objektů vystačíme s numerickými podmínkami. Z velké části půjde o podmínky funkcionální, např. rovnosti, ale nelze se omezit pouze na ně. K popisu vlastností „nalevo“, „nad“ nebo „uvnitř“ jsou potřeba nerovnosti, které nepatří mezi funkcionální podmínky (viz kapitola 1.5). Pokud chceme u grafických objektů navíc měřit vzdálenost nebo úhly, není možné se omezit jen na lineární výrazy. V obecných případech totiž pro určení vzdáleností a úhlů dostáváme kvadratické rovnice nebo rovnice, které na ně lze převést.

> Příklad podmínky pro úsečku pevné délky
> Mějme úsečku určenou dvěma body $A[x_a,y_a]$ a $B[x_b,y_b]$ a konstantu *k* vyjadřující délku. Pak podmínka pro omezení úsečky na pevnou délku *k* může znít takto:
> $(x_a-x_b)^2 + (y_a-y_b)^2 = k^2$.

Takže chceme-li plnohodnotně využít možností omezujících podmínek v GUI, je třeba mít algoritmus, který řeší obecné numerické podmínky. Je velmi vhodné, aby takový algoritmus uměl řešit cykly podmínek, a tak nám usnadnil návrh omezujících podmínek pro jednotlivé grafické objekty.

Bohužel, ve druhé části, kterou jsme věnovali právě popisu algoritmů, není uveden žádný algoritmus, který by se s těmito požadavky zcela vyrovnal. Není to však způsobeno tím, že byl takový algoritmus opomenut, ale spíše tím, že takový efektivní algoritmus není znám.

Musíme si tedy položit otázku, ze kterých požadavků slevíme. V druhé části jsou uvedeny dva algoritmy (Ultraviolet a Cassowary), které jsou schopny řešit lineární numerické podmínky včetně nerovností. Oba navíc dovedou pracovat s cykly podmínek. Ustoupíme-li tedy z požadavku na obecné numerické podmínky a omezíme-li se pouze na lineární podmínky, můžeme tyto algoritmy použít. Znamená to však, že nebudeme schopni realizovat některé grafické objekty. Podrobnějšímu rozboru, které grafické objekty jsme schopni popsat a které ne, je věnována následující kapitola.

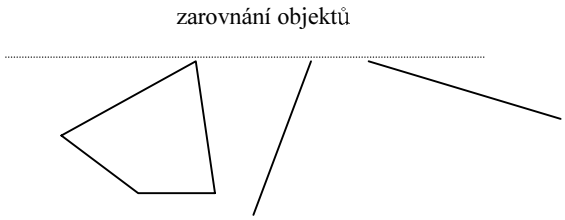
Tím, že oba algoritmy Ultraviolet i Cassowary splňují požadované vlastnosti, je možno v grafickém editoru využít kterýkoli z nich. Pro reálné použití však byl z následujících důvodů zvolen algoritmus Cassowary. Jednak algoritmus Ultraviolet je spíše ukázkou koncepce, jak navrhnout obecný algoritmus využitím sady specializovaných řešičů, a v současné verzi není algoritmus inkrementální. Navíc v době vývoje aplikace nebyla k dispozici vhodná implementace algoritmu v programovacím jazyce, v němž je grafický editor naprogramován.

3.2 Grafické objekty realizované v grafickém editoru

Tím, že jsme se omezili na lineární podmínky, není obecně možné popsat objekty, v nichž by bylo zapotřebí měřit vzdálenost nebo úhel. V obecných případech nejsme tedy schopni realizovat podmínky typu: vzdálenost dvou bodů je pevná, dvě úsečky jsou rovnoběžné, popř. kolmé, mají stejnou velikost nebo spolu svírají určitý úhel.

Nelze tedy ani popsat obecný obdélník. Ten dokážeme popsat pouze v případě, že má strany rovnoběžné s osami souřadnic. Pak podmínky pro kolmost sousedních stran a rovnoběžnost protilehlých stran jsou lineární. V takovém případě jsme dokonce schopni změřit i velikost strany, a je tedy možné omezit takový obdélník na čtverec. Dále lze popsat obecnou úsečku, ale jak již bylo řečeno, není možné ji omezit na úsečku dané velikosti.

Vlastnost, kterou lze snadno obecně realizovat, je zarovnání. Netýká se sice jednotlivých objektů, spíše popisuje chování celých skupin objektů. Zarovnáním rozumíme udržování objektů ve stavu, ve kterém mají své krajní body ve stejné linii s krajními body ostatních objektů skupiny, nad kterou byla operace zarovnání aplikována. Na obrázku je ukázka takového zarovnání pro objekty čtyřúhelník a dvě úsečky. Objekty jsou zarovnány nahoru, tečkovanou čarou je vyznačena společná horní linie objektů.



Objekty, které tedy jsou v grafickém editoru realizovány, jsou bod, úsečka, obecný čtyřúhelník a úsečka s bodem. Lze je pomocí volitelných podmínek různě omezovat. Například tak, že z obecného čtyřúhelníku vznikne kosodélník, popř. obdélník, který má strany rovnoběžné s osami; z obecné úsečky vznikne rovnoběžka s osou x, popř. osou y; nebo obecný bod je pevně zafixován na určitou pozici.

Grafické objekty jsou v aplikaci navrženy tak, že jeden objekt se může skládat z jiných objektů. Konkrétním příkladem jsou objekty úsečka a čtyřúhelník, které jsou tvořeny skupinou bodů, nebo objekt úsečka s bodem skládající se z úsečky a bodu. Omezující podmínky lze tedy klást nejen na objekt samotný, ale i na objekty, které obsahuje.

Omezující podmínky nad grafickými objekty v aplikaci lze rozdělit do dvou základních skupin. První typem jsou podmínky *stálé*, neboli takové, které jsou v systému zařazeny po celou dobu, po kterou je grafický objekt v editoru. Příkladem takových podmínek jsou podmínky omezující rozsah domén proměnných odpovídajících souřadnicím bodů. Druhou skupinu pak tvoří podmínky *volitelné*. U nich se může v průběhu práce s editorem uživatel rozhodnout, zda mají být na objekt aplikovány, nebo mají být ignorovány. U některých volitelných podmínek je navíc možné určovat i konstantu, kterou podmínka využívá jako parametr. Příkladem takové volitelné podmínky je podmínka omezující obecnou úsečku na rovnoběžku s osou x.

V kapitole 3.4.1 je detailně rozebráno využití hierarchie u stálých i volitelných podmínek tak, aby byla možné se vypořádat s případy, v nichž některé podmínky nelze splnit zároveň.

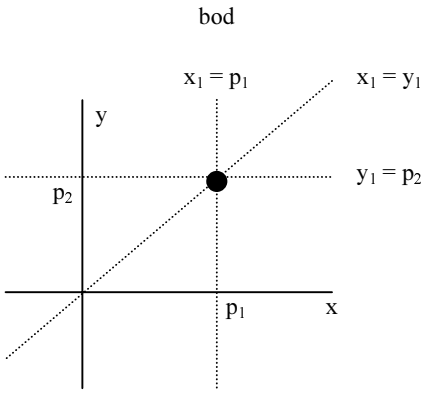
3.2.1 Bod

Bod je nejjednodušší grafický objekt. Je určen dvojicí proměnných-souřadnicemi v rovině, řekněme $[x_1,y_1]$. Obě proměnné jsou omezeny stálými podmínkami na určitý interval, daný rozměry kreslící plochy, v níž jsou grafické objekty vykreslovány. Volitelné podmínky pro bod jsou:

- souřadnice x_1 je fixována na danou pozici ($x_1 = p_1$)

- souřadnice y_1 je fixována na danou pozici ($y_1 = p_2$)
- bod se může pohybovat pouze po ose prvního kvadrantu ($x_1 = y_1$)

Z tohoto přehledu je patrné, že volitelné podmínky mohou při svém výběru způsobit konflikt s jinými vybranými podmínkami. A proto je třeba zvážit, zda mají být volitelné podmínky v hierarchickém systému zařazeny mezi required podmínky, nebo jejich nutnost splnění oslabit (viz kapitola 3.4.1). Na obrázku je znázorněn bod a tečkované čáry naznačují, jak jednotlivé volitelné podmínky omezují možnou polohu daného bodu.



Hlavní význam bodu je především v tom, že je hojně využíván pro kompozici složitějších grafických objektů.

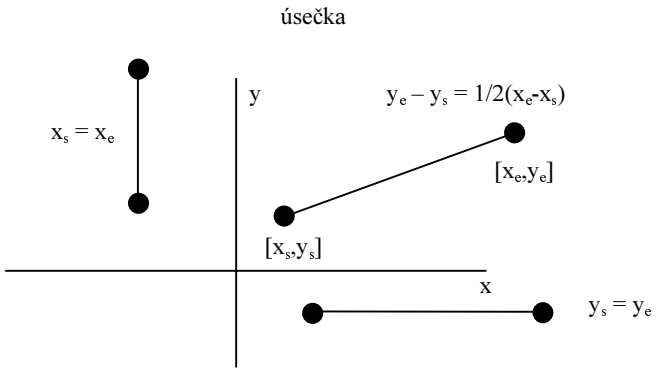
3.2.2 Úsečka

Úsečka je určena dvěma body [x_s, y_s] a [x_e, y_e]. Oba body jsou svázány svými stálými podmínkami a popř. aktivními volitelnými podmínkami, o kterých se zmiňuje předchozí kapitola. Navíc je možné využít tyto volitelné podmínky:

- úsečka je rovnoběžná s osou x ($y_s = y_e$)
- úsečka je rovnoběžná s osou y ($x_s = x_e$)
- úsečka svírá s osou x daný úhel ($y_e - y_s = p_1(x_e - x_s)$)

Třetí podmínka ukazuje, jak lze pomocí lineární rovnice zapsat vztah závisející na úhlu, pokud je tento úhel (resp. tangens úhlu) pevně dán.

Na obrázku je nakresleno několik úseček, na které byly aplikovány jednotlivé volitelné podmínky.



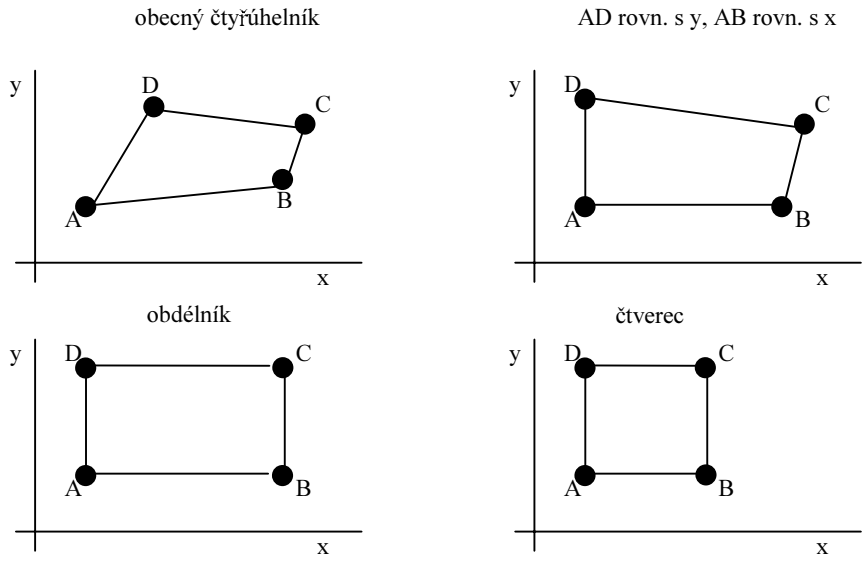
3.2.3 Čtyřúhelník

Čtyřúhelník je určen čtyřmi body A[x_a, y_a], B[x_b, y_b], C[x_c, y_c], D[x_d, y_d]. Jednotlivé body jsou omezeny svými stálými a aktivními volitelnými podmínkami. Vlastní volitelné podmínky čtyřúhelníku jsou:

- strana AD rovnoběžná s osou y ($x_a = x_d$)
- strana AB rovnoběžná s osou x ($y_a = y_b$)
- $x_a-x_d = x_b-x_c$
- $y_a-y_b = y_d-y_c$
- $x_a-x_b = y_b-y_c$

Třetí a čtvrtá volitelná podmínka vyjadřují lineární způsobem jakousi rovnoběžnost stran. Samostatně ji nezaručují, ale pokud jsou aplikovány obě, je čtyřúhelník omezen na kosodélník. Poslední pátá podmínka je lineárním vztahem, který, pokud jsou všechny strany čtyřúhelníku rovnoběžné s osami souřadnic (2 s osou x a 2 s osou y), omezují obdélník na čtverec. Neboli při vybrání podmínek 1-4 je čtyřúhelník obdélníkem, pokud navíc vybereme i poslední podmínku je objekt omezen na čtverec.

Na sérii obrázků nyní předvedeme, jak postupná aplikace jednotlivých volitelných podmínek omezuje obecný čtyřúhelník. První obrázek ukazuje obecný čtyřúhelník, na druhém obrázku je omezen prvními dvěma podmínkami (rovnoběžnost strany AD s osou y a strany AB s osou x). Třetí obrázek ukazuje čtyřúhelník omezený podmínkami 1-4 a konečně poslední obrázek znázorňuje čtyřúhelník, který má všech pět volitelných podmínek aktivních.



3.2.4 Úsečka s bodem

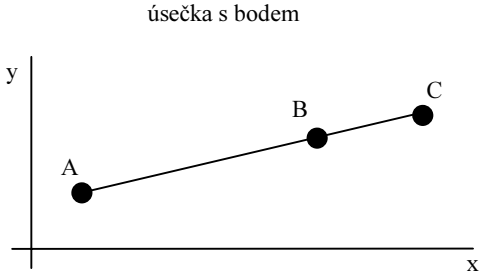
Objekt je určen úsečkou AC a bodem B. Pro další použití určeme body následovně: $A[x_l, y_l]$, $B[x_m, y_m]$ a $C[x_r, y_r]$. Úsečka AC i bod B mají vlastní stálé i volitelné podmínky. Navíc přibývá volitelná podmínka:

- bod B dělí úsečku AC v daném poměru ($x_m-x_l = p(x_r-x_l)$; $y_m-y_l = p(y_r-y_l)$),

která omezuje objekt tak, že body A, B a C jsou kolineární a navíc bod B dělí úsečku AC v daném poměru. Dosadíme-li za p číslo v rozsahu <0;1>, je bod B vnitřním bodem úsečky. Pro p=1/2 je bod B středem úsečky AC.

Volitelná podmínka tohoto grafického objektu se liší od všech předchozích případů tím, že je určena dvěma výrazy. Je tedy nutné při implementaci dbát na to, aby nenastal případ, že úsečka je omezena jen jedním z výrazů. Tato situace by mohla nastat tehdy, kdyby preference volitelné podmínky byla required a kvůli ostatním omezením by nebylo možné jeden z výrazů přidat do systému.

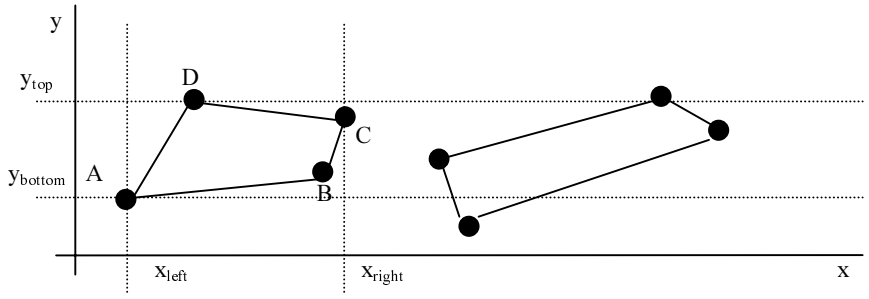
Na obrázku je znázorněn grafický objekt úsečka s bodem pro p=2/3.



Výše uvedené grafické objekty nejsou úplným výčtem možností, kterých lze dosáhnout pomocí lineárních podmínek. Spíše jde o několik základních ukázek. Jak ukazuje příloha 1, další objekty lze snadno do programu přidat, je tedy možné později tento výčet rozšířit.

V této kapitole se ještě zmíníme o jednom typu volitelných omezujících podmínek, který není spjat s jedním konkrétním objektem, ale spíše jde o podmínku nad skupinou grafických objektů. Jde totiž o zarovnání objektů, které můžeme učinit v libovolném ze čtyř směrů: nalevo, napravo, nahoru, nebo dolů. Například pokud jsou objekty zarovnané nahoru, znamená to, že horní okraje všech zarovnaných objektů mají stejnou souřadnici y. V praxi toho lze docílit: 1. použitím nových proměnných (x_{left}, x_{right}, y_{top}, y_{bottom}), které pro každý objekt udávají jeho hranice. Proměnná x_{left} udává souřadnici x bodu objektu nejvíce vlevo, x_{right} bodu nejvíce vpravo, y_{top} udává souřadnici y bodu, který je nejvíce nahoře, a y_{bottom} souřadnici y bodu nejvíce dole. Za 2. nalezením minima resp. maxima z hodnot všech proměnných. V obou případech musí grafický objekt udávat na vyžádání proměnnou, která je nejvíce vlevo (resp. vpravo, nahoře, dole), tzv. *okrajovou proměnnou*, a omezující podmínky, které zarovnání realizují pak jsou rovnostmi těchto okrajových proměnných jednotlivých objektů. Obrázek ukazuje dva čtyřúhelníky, na které je uplatněno zarovnání nahoru. Pro levý čtyřúhelník jsou tečkovaně zvýrazněny hodnoty jeho okrajových souřadnic.

zarovnání nahoru dvou čtyřúhelníků



Na podmínkách zarovnání je výjimečné také to, že se často v rámci systému přidávají a ubírají. Totiž v případě, že objekt změnil své proporce, mohlo dojít k tomu, že nyní je u objektu jiná okrajová proměnná. Pak je nutné původní podmínku v systému nahradit podmínkou novou, ve které figuruje již aktuální okrajová proměnná.

3.3 Návrh grafického editoru

Uveďme nyní hlavní myšlenky, které stály u zrodu vlastního programu. Aplikace by měla umožňovat vkládání a následnou editaci grafických objektů. Měla by uživateli nabídnout

způsob, jak objekty přesouvat a měnit jejich proporce s ohledem na omezení, které byly na objekty položeny. Uživatel by také měl mít možnost výběru, které podmínky objekt splňovat má a které nikoli. Použitím hierarchie podmínek lze navíc uživateli nabídnout možnost každé podmínce nastavit libovolnou preferenci, a tím ovlivnit důležitost splnění jednotlivých volitelných podmínek pro případ, že některé volitelné podmínky nelze splnit zároveň. Kvůli přehlednosti by také měl program ukazovat jednak seznam objektů, které jsou v systému, jednak seznam vybraných volitelných podmínek nad těmito objekty. Program by měl být otevřený, aby bylo možné jej kdykoliv rozšířit použitím jiného řešícího algoritmu, či přidáním nových grafických objektů nebo volitelných podmínek nad nimi.

3.4 Implementace grafického editoru

V této kapitole uvedeme důležitá fakta, která se týkají vlastní implementace grafického editoru. V žádném případě nepůjde o přesný popis jednotlivých instrukcí programu, spíše o přiblížení klíčových myšlenek. Nejdříve navrhneme vhodnou hierarchii podmínek pro popsání všech podmínek, které budou v grafickém editoru figurovat. Stručně shrneme základní fakta o programovacím jazyku, ve kterém je editor napsán, dále se seznámíme s objektovým modelem aplikace, zmíníme se o způsobu zapouzdření řešiče omezujících podmínek a nakonec popíšeme stručně uživatelské rozhraní aplikace.

3.4.1 Hierarchická struktura podmínek v grafickém editoru

Doposud jsme určili jednotlivé typy podmínek, které budou v implementaci figurovat. Je nutno ještě přesně určit, jak vhodně rozdělit všechny podmínky s využitím hierarchie tak, aby byl systém co nejflexibilnější a dával intuitivní výsledky. Je zřejmé, že stay podmínky musí mít co nejnižší preferenci (viz kapitola 1.3). Rovněž je třeba, aby podmínky, které omezují doménu proměnných byly povinné (required –nejvyšší preference). Edit podmínky by neměly být povinné, ale měly by mít vyšší preferenci než stay podmínky (viz kapitola 1.3). U volitelných podmínek by mělo být umožněno uživateli rozhodnout o jejich preferenci až v době jejich uplatnění. Přesto je důležité těmto podmínám přidělit nějakou vhodnou preferenci automaticky s možností změny, např.: pro volitelné podmínky objektů preferenci o stupeň nižší než povinné a pro podmínky zarovnání ještě o jeden stupeň nižší. Důvodem takové volby je respektování představy intuitivního chování objektů.

Výsledkem je tedy pěti-stupňová hierarchie s preferencemi 1 (required), 2 (strong), 3 (high), 4 (medium) a 5 (weak). V následující tabulce je pak shrnuto zařazení jednotlivých

typů podmínek k těmto stupňům. Pro volitelné podmínky toto zařazení není závazné, podmínkám může být uživatelem přidělena libovolná preference z této pětistupňové škály.

tabulka č. 3

Required	omezení domén proměnných
Strong	volitelné podmínky nad objekty
High	volitené podmínky nad skupinami (zarovnání)
Medium	edit podmínky
Weak	stay podmínky

Jak již bylo řečeno, váhy podmínek slouží k diferenciaci podmínek se stejnou preferencí. Vzhledem k tomu, jak byl systém podmínek navržen, není prakticky nutné vah využívat. Pouze na jediném místě v grafickém editoru mohou zlepšit chování aplikace. Použijeme je k určení pořadí důležitosti jednotlivých stay podmínek v rámci jednoho grafického objektu.

Máme-li nějaký grafický objekt, tak očekáváme, že při změně jeho proporcí tažením některého jeho bodu se bude objekt chovat podle určitých pravidel, která nám dovolí chování předpovídat. Měníme-li tedy polohu některého bodu objektu, lze předem říci, které další body budou přesunuty tak, aby platily aktivní omezující podmínky objektu. Toto chování je ale závislé na interním řešiči podle toho, které stay podmínky upřednostní před jinými. Abychom této závislosti zabránili, volíme pro jednotlivé body různé váhy tak, abychom mohli určit, které stay podmínky jsou pro objekt důležitější než jiné. Například pro čtyřúhelník jsou váhy stay podmínek pro jednotlivé body voleny následovně: stay podmínky pro proměnné bodu A mají váhu 1,0; bodu B mají 2,0; bodu C 4,0 a bodu D 8,0. Je tedy zaručeno, že bod D je nejstabilnější. Podobným způsobem jsou voleny váhy stay podmínek i u ostatních grafických objektů.

3.4.2 Programovací jazyk

K realizaci grafického editoru byl vybrán programovací jazyk Java z těchto důvodů. Jde o relativně nový, moderní objektově orientovaný jazyk, který programátorům přináší nové možnosti. Ve svých základech vychází především z C++ a Smalltalk. Za své masové rozšíření vděčí hlavně své nezávislosti na operačním systému. Toho dosahuje tím, že je interpretovaný. Pro spuštění programu vyvinutého v jazyce Java na libovolné platformě tedy stačí, aby pro tento operační systém existoval interpret jazyka Java. Na druhou stranu je třeba upozornit, že software v Javě není neúnosně pomalý, třebaže programy nejsou kompilovány do nativního

kódu. Ve skutečnosti jsou programy po napsání v textové podobě kompilovány do kódu, který nativní sice kód připomíná, ale je zcela nezávislý na platformě. V této podobě je pak možno jej distribuovat.

Vzhledem k tomu, že popis implementace grafického editoru vychází ze základní znalosti jazyka Java, je nutné se nyní seznámit alespoň s některými jeho rysy a myšlenkami. Každý objekt v Javě je popsán *třídou* (class). Ta zapouzdřuje všechny *položky* (fields) a *metody* (methods) objektu. Položky nesou informaci o objektu, zatímco metody realizují funkce objektu. V objektovém programování je běžné, aby položky objektu byly zapouzdřené, a tedy nebyly přímo přístupné pro ostatní objekty. V Javě by proto pro každou položku objektu, která má být *veřejná* (public), měly být i dvě metody. Jedna hodnotu mění (set metoda), druhá hodnotu zpřístupňuje (get metoda). Dědičnost je v Javě dvojího druhu. Jednak je možné využít standardní dědičnosti známé ze základů objektového programování, jednak je možné použít tzv. *rozhraní* (interface). Rozhraní je seznamem hlaviček metod, jejichž těla musí objekt implementovat, aby mohl splňovat určitou funkčnost. Tím, že každý objekt může splňovat více rozhraní, a tím realizovat větší funkčnost, lze docílit stejného efektu jako u vícenásobné dědičnosti. Význam rozhraní spočívá v tom, že na objekt se lze dívat nejen jako na objekt určitého typu (určeného předkem v hierarchii dědičnosti), ale také jako na objekt, který umí realizovat určitou funkčnost danou rozhraním. Je tedy možné například vytvořit pole odkazů na konkrétní rozhraní, do jeho prvků pak vkládat libovolné objekty, které dané rozhraní splňují a využít tak polymorfismus aniž by objekty měli společného předka.

Více o programovacím jazyce Java je možné najít v pracích [9], [10], nebo na internetu [19], [20].

3.4.3 Objektový model aplikace

Následující kapitola popisuje význam a použití významných tříd grafického editoru. Nejde o detailní popis jednotlivých objektů ani o výčet tříd a jejich metod. Ten lze nalézt v elektronické referenční příručce. Kapitola je rozdělena do třech celků. První je věnován třídám realizujícím omezující podmínky a vlastnímu řešiči podmínek. Druhý představuje grafické objekty, se kterými aplikace pracuje, a třetí část je zaměřena na některé objekty uživatelského rozhraní.

3.4.3.1 Realizace omezujících podmínek

Implementace algoritmu Cassowary, kterou grafický editor využívá, je kompletní realizací řešiče. Obsahuje tedy i objekty, které popisují omezující podmínky a jejich proměnné. Tím, že chceme docílit jisté nezávislosti aplikace na konkrétní implementaci a typu řešiče, není možné těchto objektů obecně využít. Bylo tedy nutné navrhnout vlastní objekty, které realizují omezující podmínky a proměnné.

Přestože uživatel, který bude s grafickým editorem pracovat, je od přímého použití omezujících podmínek odstíněn způsobem popsaným v kapitole 3.4.3.2, není nezajímavé se krátce seznámit s jejich implementací.

3.4.3.1.1 Proměnné

Proměnné jsou realizovány třídou Variable. Mají dvě důležité položky: hodnotu (value) a jméno (name). Hodnota proměnné je desetinné číslo. Jméno proměnné je generováno tak, aby bylo v rámci běhu aplikace jednoznačné, a mohlo tedy být klíčem při vyhledávání v různých seznamech.

3.4.3.1.2 Omezující podmínky

Omezující podmínky jsou rozvrženy tak, že jejich jednotlivým typům jsou určeny samostatné třídy. Existují tedy samostatné třídy pro edit a stay podmínky, rovněž tak pro obecné lineární numerické podmínky. Všechny typy podmínek jsou navíc odvozeny od abstraktní podmínky, mají tedy určité klíčové vlastnosti společné.

Struktura dědičnosti tříd typů omezujících podmínek vypadá následovně:

```
AbstractConstraint
      Constraint
            EditConstraint
            StayConstraint
            LinearConstraint
                  Equality2VConstraint
                  TightenVariableConstraint
      MultiConstraint
            InterObjectConstraint
                  AlignmentConstraint
```

Lze nahlédnout, že existují dvě základní skupiny podmínek. Jedna má jako společného předka třídu Constraint. Jde o běžné typy omezujících podmínek, o kterých jsme se zmiňovali v předchozích částech práce. Druhá skupina, která vychází z třídy MultiConstraint, realizuje možnost, jak zastřešit množinu podmínek jedinou třídou. Využijeme ji v případě, v němž jednu omezující podmínku není možné popsat jedním výrazem, např. k dosažení kolinearity

bodu a úsečky (viz 3.2.4). Tím, že jsou takto jednotlivé výrazy sloučeny do jediné podmínky typu MultiConstraint, je, například při odebrání podmínky ze systému, zaručeno korektní zpracování všech jejích výrazů.

Navržená hierarchická struktura tříd je úplná pro potřeby současné verze grafického editoru. Nicméně, v případě nutnosti je tato struktura otevřená, takže je možné ji kdykoliv rozšířit o nové typy omezujících podmínek. Pak je samozřejmě zapotřebí implementovat jiný algoritmus pro řešení omezujících podmínek, který dovede řešit i tyto nové typy podmínek. Způsobu, jak v systému nahradit stávající řešič vlastní implementací, je věnována příloha 1.

Každá podmínka má svoji preferenci, která vyjadřuje její důležitost v hierarchickém uspořádání podmínek. Systém na základě rozboru v kapitole 3.4.1 nabízí pět symbolických stupňů preference: required, strong, high, medium a weak. Je však možné tento počet rozšířit. Pokud je podmínka označená preferencí required, je její splnění požadováno povinně. Může se ale stát, že při přidání takové podmínky řešič oznámí, že není schopen tuto podmínku splnit a není možné ji do systému zařadit. Tím je zajištěno, že řešič nikdy nedospěje do stavu, kdy je systém příliš omezený. Pokud by tato situace hrozila, jednoduše odmítne nově přidávanou podmínku do systému zařadit. Uvědomme si ještě, že tato situace se týká pouze volitelných podmínek objektů. Stálé podmínky (jde o podmínky omezující domény proměnných) jsou do systému podmínek zařazovány při přidávání nového objektu, tedy vždy před volitelnými podmínkami, a nemohou být v konfliktu s žádnými podmínkami systému, protože jsou podmínkami nad zcela novými proměnnými.

K tomu, aby bylo možné ještě mezi jednotlivými podmínkami stejné preference určovat jejich důležitost, slouží položka strength. Jde o číslo, které udává váhu podmínky. Standardně je tato hodnota nastavena na 1,0. V kapitole 3.4.1 je uvedeno, jak jsou váhy využity pro rozlišení jednotlivých stay podmínek.

Věnujme nyní pozornost jednotlivým typům základních podmínek, tedy těm, které vychází z třídy Constraint. Třídy EditConstraint, resp. StayConstraint, realizují podmínky typu edit, resp. stay, tedy podmínky svazující vždy jednu proměnnou s konkrétní hodnotou. Obě třídy proto obsahují odkaz na proměnnou, kterou omezují. Třída LinearConstraint realizuje všechny podmínky, které lze vyjádřit jako lineární rovnost nebo nerovnost. Způsob, jakým matematický výraz vnitřně reprezentuje odpovídá zápisu $c_1x_1+c_1x_1+\ldots+c_nx_a$ *op* c_0, kde c_i jsou reálná čísla, x_i proměnné a *op* je operátor =, ≤ nebo ≥.

Třídy Equality2VConstraint, resp. TightenVariable, nepřinášejí nový typ podmínek, jsou jen realizací často se vyskytujících podmínek typu x = y, resp. x *op* c, kde x,y jsou proměnné, c je reálné číslo a op je operátor =, ≤ nebo ≥.

Nakonec se ještě zmiňme o třídě AlignmentConstraint, která je speciálním typem složené podmínky (MultiConstraint). Realizuje vztah zarovnání obecných grafických objektů. Jde tedy o jediný typ podmínky, která vyjadřuje obecný vztah grafických objektů. Ve skutečnosti však jde vnitřně jen o sadu rovností mezi určitými proměnnými (viz 3.2).

3.4.3.1.3 Řešič omezujících podmínek

Interní řešič omezujících podmínek postavený na algoritmu Cassowary odpovídá třídě EDU.Washington.grad.gjb.cassowary.ClSimplexSolver. Jak již bylo řečeno, aplikace je ale od implementace tohoto algoritmu odstíněna. To je realizováno následující koncepcí. Rozhraní SolverInterface určuje hlavičky metod, jejichž těla musí třída řešiče implementovat, aby mohla být v aplikaci použita. Třída CassowarySolver je příkladem třídy, která rozhraní implementuje, a je vlastně zapouzdřením algoritmu Cassowary, jehož třída ClSimplexSolver rozhraní nesplňuje. Třída CassowarySolver tvoří samostatný modul, který lze v aplikaci využít. Třída ConstraintSolver pracující již nad grafickými objekty využívá tento modul jako nástroj pro řešení podmínek, které popisují jednotlivé grafické objekty.

V rozhraní SolverInterface jsou tyto metody:

- boolean addConstraint(Constraint c);

Slouží k přidání podmínky *c* do systému. Vrací logickou hodnotu, která říká, zda byla podmínka úspěšně přidána.

- boolean removeConstraint(Constraint c);

Pokusí se o odebrání podmínky *c* ze systému. Vrací logickou hodnotu, která říká, zda byla podmínka úspěšně odebrána. Situace, kdy podmínku nejde odebrat, by nastat nikdy neměla, pokud tedy metoda vrátí hodnotu *false*, znamená to nejspíše chybu v programu(např. pokus o odebrání neexistující podmínky).

- void addEditVariable(Variable v);

Označí proměnnou *v* jako parametrickou a vloží do systému odpovídající edit podmínku.

- void removeEditVariable(Variable v);

Zruší označení proměnné *v* jako parametrické a odebere ze systému odpovídající edit podmínku.

- void beginEdit();

Připraví systém na rychlé přepočítávání. Předpokladem je, že po tuto dobu nebude měněna sada podmínek a nebude se měnit seznam parametrických proměnných.

- void endEdit();

Ukončí připravenost systému na rychlé přepočítávání. Odstraní zároveň všechny edit podmínky – zruší označení parametrických proměnných.

- void reset();

Inicializuje řešič.

- void suggestValue(Variable v, int value);

Pokusí se o nastavení parametrické proměnné *v* na hodnotu *value*. Na proměnnou *v* musí být před voláním této metody aplikována metoda addEditVariable(v).

- void resolve();

Přepočítá řešení tak, aby byly podmínky opět splněny. Využívá maximální část z předchozího řešení.

- void solve();

Přepočítá řešení tak, aby byly podmínky opět splněny bez využití předchozího řešení.

V následujícím příkladu je možné vysledovat praktické použití jednotlivých metod. Ke korektní funkci algoritmu je nutné zachovat určité pořadí jejich volání.

> Příklad využití volání jednotlivých metod.
> Předpokládejme, že v systému je vložen grafický objekt úsečka s bodem (viz kapitola 3.2.4). Proběhlo tedy volání metody addConstraint(c) pro každou podmínku *c*, kterou bylo třeba do systému zařadit (nejdříve podmínky pro úsečku AC, dále pro bod B a nakonec podmínka nastavující vztah mezi bodem a úsečkou). V případě, že uživatel vybere myší grafický bod A (počátek úsečky určený proměnnými $[x_1, y_1]$) proběhnou následující volání. Nejdříve jsou zařazeny edit podmínky voláním addEditVar(x_1) a addEditVar(y_1), potom je systém připraven metodou beginEdit(). Dokud uživatel drží stisknuté tlačítko myši, tak při každém pohybu myší (pozice dána parametry a,b) voláme metody suggestValue(x_1,a) a suggestValue(y_1,b), které naplní parametrické proměnné novou hodnotou, a nakonec voláním metody resolve() přivedeme systém zpět do konzistentního stavu. V případě, že uživatel pustí tlačítko myši, je operace tažení ukončena voláním metody endEdit(). Nakonec, pokud se uživatel rozhodne grafický objekt odstranit ze systému, zavoláme pro každou jeho podmínku *c* metodu removeConstraint(c).

Jak je zřejmé z příkladu, pro volání každé metody rozhraní je vhodná jiná doba. Volání beginEdit odpovídá počátku tažení objektu myší. Před tímto voláním je nutné systém připravit na to, které proměnné se budou měnit, a to pomocí metody addEditVar. V průběhu tažení se

pak střídá volání metody suggestValue, která nastavuje parametrické proměnné, s voláním metody resolve, která systém přepočítá. Při ukončení tažení myší je třeba zavolat endEdit. Mezi voláními metod beginEdit a endEdit není navíc možné využít jinou metodu než výše zmíněné suggestValue a resolve.

Následující fragmenty kódu tříd CassowarySolver a ConstraintSolver ukazují na metodě endEdit, jak je prakticky provedená realizace zapouzdření interního řešiče a jak je v třídě ConstraintSolver k internímu řešiči přistupováno pouze přes rozhraní SolverInterface.

```
public class CassowarySolver implements SolverInterface {
      EDU.Washington.grad.gjb.cassowary.ClSimplexSolver solver;

      public void endEdit() {
            try
            {
                  solver.endEdit();
            }
            catch (ExCLInternalError ex) {
                  handleException(ex);
            }
      }
…
}

public class ConstraintSolver{
      SolverInterface internalSolver;

      public void endEdit(){
            internalSolver.endEdit();
      }
…
}
```

Proměnná internalSolver ve třídě ConstraintSolver je naplněna při vytvoření instance třídy z vnějšku. Proto při výměně interního řešiče není nutný žádný zásah do kódu třídy ConstrainSolver, jen je třeba při vytváření její instance dodat jako parametr nově požadovaný řešič. Příloha 1 popisuje způsob výměny interního řešiče včetně podrobného popisu třídy CassowarySolver.

3.4.3.2 Reprezentace grafických objektů

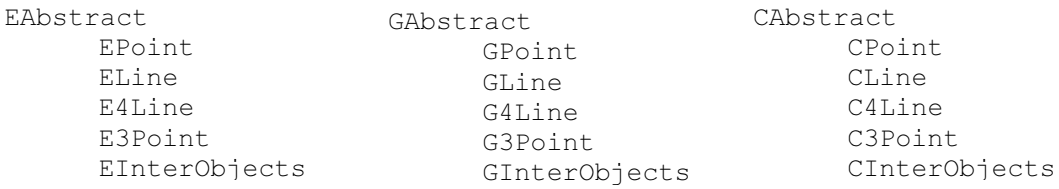
Tato kapitola popisuje způsob, jak jsou implementovány obecně grafické objekty. Zaměříme se především na objekty popsané v kapitole 3.2.

Základní snahou při návrhu vhodné reprezentace grafických objektů bylo oddělit způsob zobrazení grafického objektu od jeho popisu pomocí omezujících podmínek. Tak vznikla následující struktura: všechny grafické objekty jsou třídami odvozenými ze společného

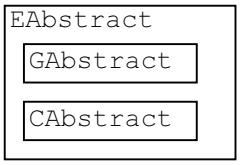
předka EAbstract, každá taková třída obsahuje potom odkaz na třídu zajišťující zobrazení objektu a odkaz na třídu realizující popis objektu omezujícími podmínkami. Všechny třídy zobrazení objektů mají společného předka GAbstract a popisné třídy CAbstract. Grafické objekty jsou navíc koncipovány tak, aby mohly být složeny z jiných grafických objektů. Takže například grafický objekt úsečka s bodem (viz kapitola 3.2.4) je složen z grafických objektů úsečka a bod.

Na prvním obrázku je zobrazena struktura dědičnosti grafických objektů a jejich částí. Na druhém pak je k nahlédnutí způsob, jakým grafický objekt vlastní své části, popř. i jiné grafické objekty.

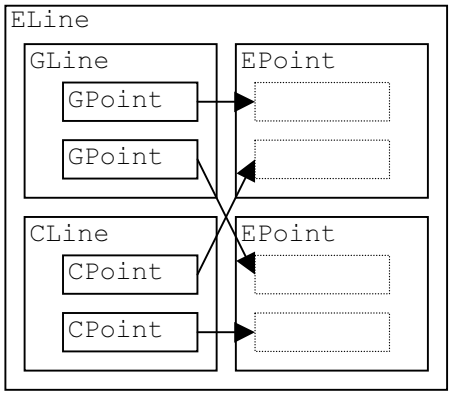
Třídy grafických objektů



Abstraktní grafický objekt



Grafický objekt úsečka



Jednotlivé třídy odpovídají grafickým objektům, o kterých jsme se zmiňovali v kapitole 3.2. Takže třída EPoint odpovídá bodu, ELine úsečce, E4Line čtyřúhelníku a E3Point třem kolineárním bodům. Třída EInterObjects z tohoto rámce vybočuje. Slouží totiž jako objekt, který svazuje podmnožinu všech grafických objektů přítomných v systému, a nabízí tak uživateli nad touto podmnožinou volbu podmínek pro zarovnání objektů.

Nyní se zmíníme o některých rysech implementace grafických objektů. Tím, že všechny třídy objektů jsou potomky třídy EAbstract, sdílejí tyto společné položky: odkaz na GAbstract (zobrazení objektu), odkaz na CAbstract (popis objektu), odkaz na vlastníka, odkaz na vlastněné objekty a jméno, které je stejně jako u proměnných generováno automaticky tak, aby bylo v rámci jednoho běhu aplikace jednoznačné. Objekty typu EAbstract nemají prakticky žádnou funkčnost-slučují především objekty pro popis omezujícími podmínkami a pro grafickou reprezentaci. Při vytváření instancí těchto objektů dojde k naplnění výše uvedených položek odpovídajícími daty.

Třídy pro grafickou reprezentaci objektů (jsou potomky třídy GAbstract) zajišťují, aby byl grafický objekt vykreslen na místě k tomu určeném. Umožňují provádět s grafickou reprezetací tyto operace: vykreslení objektu (metoda draw), označení objektu jako vybraný (setSelection) a dotázání se, zda je pozice myši nad objektem (contains).

Popsání grafických objektů omezujícími podmínkami zajišťují třídy zděděné z třídy CAbstract. Ty jsou klíčovým místem každého grafického objektu. Jak již bylo řečeno, třída ConstraintSolver, která zajišťuje vlastní zpracování omezujících podmínek v aplikaci, nepracuje přímo s omezujícími podmínkami (třídami AbstractConstraint), ale právě s třídami typu CAbstract. Nabízí proto metody addConstraintObject(CAbstract), removeConstraintObject(CAbstract) a updateConstraintObject(CAbstract).

Každý potomek třídy CAbstract popisuje grafický objekt dvěma druhy podmínek: stálými a volitelnými (viz kapitola 3.2). Stálé podmínky jsou zařazeny do řešiče po celou dobu, po kterou je grafický objekt přítomen v grafickém editoru, zatímco o zařazení volitelných podmínek do řešiče může uživatel rozhodovat sám. Takže pokud přidáme nový grafický objekt O do aplikace, zavolá se metoda addConstraintObject(O), která mimo jiné zařadí do řešiče všechny stálé podmínky objektu O a z volitelných podmínek jen ty, které jsou defaultně vybrané. Pokud uživatel pracuje s objektem O a změní některé volitelné podmínky– aktivuje, popř. deaktivuje některou podmínku, změní její parametr nebo preferenci– zavolá se metoda updateConstraintObject(O), která upraví v řešiči volitelné podmínky objektu O tak, aby odpovídaly požadavkům uživatele. A nakonec, jestliže se uživatel rozhodne objekt O odstranit z aplikace, metoda removeConstraintObject(O) z řešiče odstraní všechny stálé podmínky objektu O a všechny volitelné, které byly v době odebrání objektu aktivovány.

Aby byl přehled reprezentace grafických objektů v aplikaci úplný, je třeba se ještě zmínit o způsobu, jak jsou řešeny volitelné podmínky. Jde sice o standardní omezující podmínky stejně jako podmínky stálé, je však nutné si u nich navíc pamatovat i údaje o tom,

v jakém stavu se volitelná podmínka nachází, neboli jestli je aktivována nebo není, popř. pro parametrické volitelné podmínky ještě hodnotu jejího parametru. Třídy, které volitelné podmínky realizují, jsou odvozeny z abstraktní třídy OAbstract. Implementací standardní volitelné podmínku je třída OYesNo, zatímco parametrické podmínky realizuje třída O1Number. Jak bude uvedeno v objektovém popisu uživatelského rozhraní (3.4.3.3), uživatel má k dispozici seznam všech volitelných podmínek, které jsou aktivovány. Je tedy nutné, aby každá volitelná podmínka určitým způsobem slovně uváděla svůj význam, který je potom v tomto seznamu prezentován. Následující fragment kódu ukazuje způsob vytvoření standardní volitelné podmínky pro úsečku. Jde o podmínku, která úsečku omezuje na rovnoběžku s osou y, neboli o rovnost x-ových souřadnic počátečního a koncového bodu úsečky.

```
public class CLine extends CAbstract{
public CPoint sP,eP;
private Constraint optCons1;
private OYesNo opt1;

private Constraint getOptionConstraint1(int str){
      optCons1 = new Equality2VConstraint(sP.getXVar(),eP.getXVar(),str);
      return optCons1;
}
private OYesNo getOption1(){
if (opt1 == null){
  opt1 = new OYesNo(getOptionConstraint1(ConstClass.STRONG),false,
           "stejná x-ová souřadnice",ConstClass.STRONG);
  }
return opt1;
}
```

Zatímco metoda getOptionConstraint1 vytváří vlastní omezující podmínku, metoda getOption1 vytváří instanci třídy OYesNo, tedy volitelnou podmínku právě s využitím omezující podmínky vytvořené v metodě getOptionConstraint.

3.4.3.3 Uživatelské rozhraní

Jednotlivé objekty uživatelského rozhraní jsou sice jedinou částí grafického editoru, která je přímo v kontaktu s uživatelem, je však nejméně zajímavá z funkčního hlediska. Zaměříme se jen na vybrané komponenty, které stručně popíšeme.

Celá aplikace je koncipována jako modulární. Skládá se z několika samostatných celků, jejichž vzájemnou interakci zajišťuje zastřešující objekty MainPanel a MainFrame. V dalším textu pro ně budeme používat pojem centrální modul. Jde o třídy, v kterých se jednotlivé moduly vytváří, propojují a vyvolávají na nich patřičné akce v závislosti na požadavcích uživatele. Pokud bychom tedy chtěli některý z modulů přeprogramovat a v grafickém editoru použít novou verzi modulu, musíme provést zásah jen v centrálním modulu. Takto by

například bylo možné nahradit stávající řešič Cassowary jiným algoritmem, samozřejmě s ohledem na předpoklady uvedené v kapitole 3.1.

Následující seznam je výčtem jednotlivých modulů aplikace.

- kreslící plocha třída DrawPanel

Modul realizuje vykreslování grafických objektů. K tomu potřebuje úplný seznam všech grafických reprezentací (tříd typu GAbstract) jednotlivých grafických objektů.

- řešič omezujících podmínek třída ConstraintSolver

Zpracovává popisné objekty (typu CAbstract) jednotlivých grafických objektů a zajišťuje spolupráci s interním řešičem omezujících podmínek.

- seznam základních grafických objektů systému třída ListPanel

Dává uživateli přehled o všech základních grafických objektech (objektech odvozených z EAbstract vyjma EInterObjects), které uživatel přidal do systému.

- seznam aktivních skupin objektů třída ListOfInterObjects

Umožňuje uživateli snadno a rychle vybrat požadovanou skupinu objektů, nad kterými je zvoleno nějaké zarovnání. Jde tedy o seznam všech EInterObjects objektů, které jsou v systému aktivní.

- dialog pro zadávání volitelných podmínek třída OptionsDialog

Pokud je v seznamu grafických objektů vybraný jeden objekt, pak umožňuje zadávat a měnit volitelné podmínky tohoto objektu. Jestliže je vybráno několik objektů, nabízí dialog volitelné podmínky objektu CInterObjects, tedy podmínky pro zarovnání skupiny vybraných objektů.

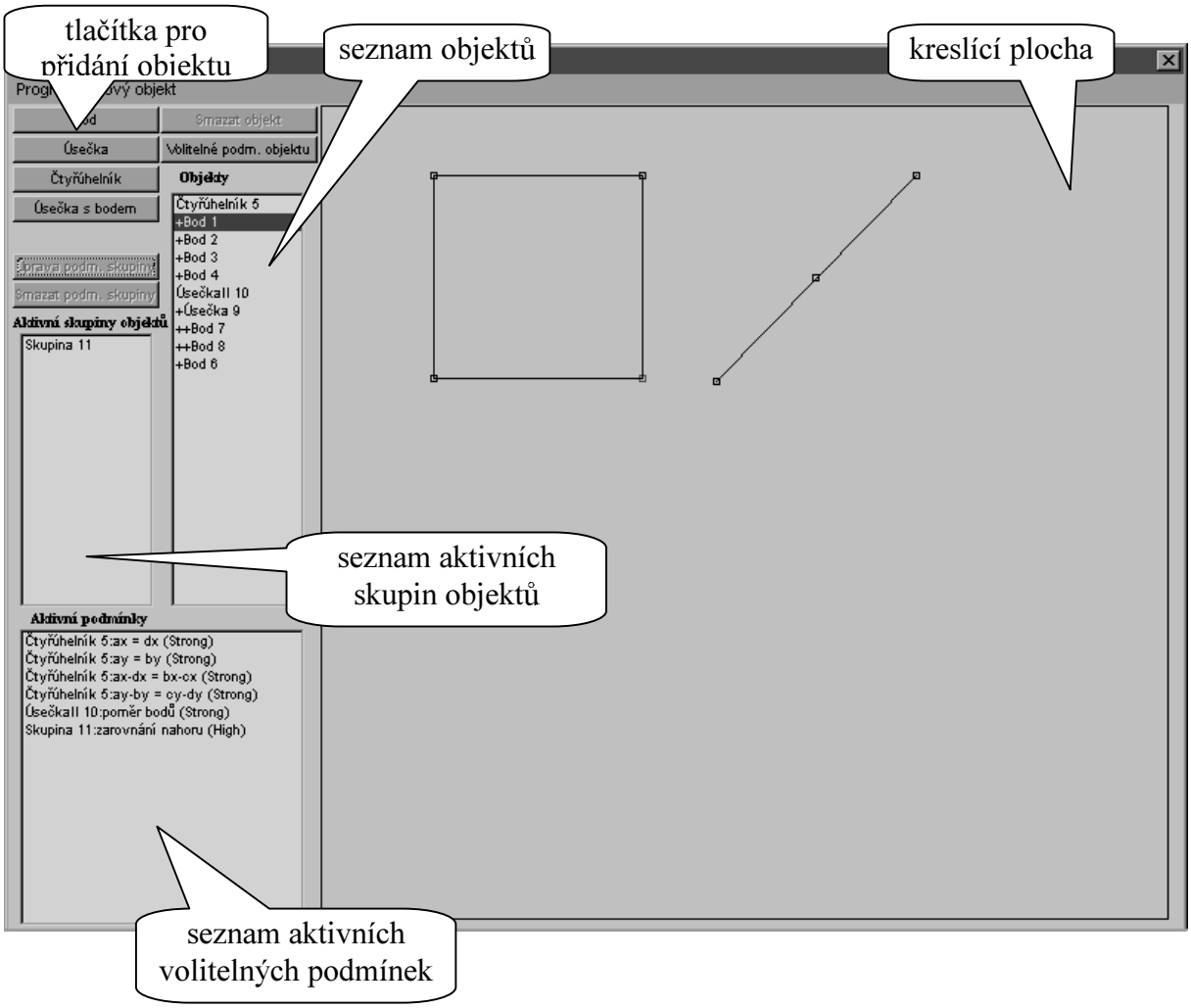
- seznam aktivních volitelných podmínek třída ListOfConstraints

Informuje přehledně o všech volitelných podmínkách jednotlivých objektů, které uživatel aktivoval, a jsou tedy zařazeny do řešiče.

- tlačítka pro přidání grafických objektů třída ControlPanel

Slouží k vkládání požadovaných grafických objektů do grafického editoru.

Na následujícím obrázku, který je kopií hlavní obrazovky aplikace, jsou výše popsané moduly označeny popiskami. Na obrázku nenalezneme modul řešiče omezujících podmínek, protože není vizuální, a dialog volitelných podmínek, který se objevuje pouze na vyžádání v samostatném okně.



Tím, že je aplikace takto modulárně rozdělena, je práce centrálního modulu velmi jednoduchá. Většinou pouze přijme požadavek k vykonání určité operace a podle jeho charakteru jej rozdělí na samostatné části a ty pak pošle ke zpracování jednotlivým modulům.

Pro ilustraci popíšeme činnost, kterou centrální modul musí vykonat, jestliže se uživatel rozhodne přidat do systému nový objekt. Tento požadavek vychází z modulu tlačítek. Jakmile centrální modul tento požadavek přijme tak:

1. vytvoří nový grafický objekt O požadovaného typu
2. požádá modul seznam grafických objektů o zařazení objektu O
3. požádá modul kreslící plocha o zařazení grafické reprezentace objektu O
4. požádá modul řešič omezujících podmínek o zařazení popisné reprezentace objektu O
5. požádá modul seznam volitelných podmínek, aby provedl úpravy s ohledem na popisnou reprezentaci objektu O.

Popis jednotlivých tříd uživatelského rozhraní nemá za cíl seznámit se způsobem ovládání grafického editoru. Stručná uživatelská příručka, která popisuje možnosti práce s programem, je uvedena v příloze 3.

Závěr

Diplomová práce zpracovává problematiku použití omezujících podmínek v grafických uživatelských rozhraních. Je demonstrací jak využít této progresivní technologie. Předvedený rozsah použitelnosti je závislý na algoritmu řešení omezujících podmínek, z hlediska všech možných grafických objektů tedy není příliš velký. Práce tak jistým způsobem odůvodňuje, proč je třeba hledat nové efektivní algoritmy, které zvládnou větší spektrum typů omezujících podmínek.

Práce je rozdělená do tří částí. V první je podán výklad základních pojmů a principů, na kterých jsou omezující podmínky založené. Druhá část je věnována popisu algoritmů, které řeší hierarchie omezujících podmínek. Je orientována především směrem k použití jednotlivých algoritmů v grafických uživatelských rozhraních, podává tedy i přehled klíčových vlastností, které by měl efektivní algoritmus mít.

Třetí část je vlastním přínosem do studia omezujících podmínek v GUI. Dává popis implementace grafického editoru, který umožňuje správu grafických objektů popsaných omezujícími podmínkami. V jejím úvodu (kapitola 3.1) je zhodnocení použitelnosti jednotlivých algoritmů, kterým je věnována část druhá, v grafických uživatelských rozhraních. Dále navrhuje nejvhodnější z nich k použití v implementaci grafického editoru. Představuje možné grafické objekty a jejich popisy omezujícími podmínkami (viz 3.2). Navrhuje také obecné rozhraní pro snadné použití libovolného algoritmu v grafických uživatelských rozhraních (viz 3.4.3.1). Velká část je pak věnována popisu hlavních myšlenek vlastní implementace programu.

Význam práce spočívá především v demonstraci možnosti využití omezujících podmínek v grafických uživatelských rozhraních. V žádném případě neuzavírá tuto oblast studia, spíše je impulsem k dalšímu výzkumu. Tím, že je navržená implementace grafického editoru otevřená, lze ji snadno modifikovat a rozšířit její možnosti použití, jak tomu ukazují přílohy 1 a 2.

Další možný vývoj by byl zajímavý v souvislosti s novým algoritmem, který by dovedl efektivně řešit obecnější vztahy než lineární rovnice a nerovnice. Pak by bylo možné rozšířit stávající program o některé grafické objekty, pro něž popis pomocí lineárních rovnic a nerovnic není.

Literatura

[1] Bardos, G. J., Borning, A., The Cassowary Linear Arithmetic Constraint Solving Algorithm: Interface and Implementation, Technical Report UW-CSE-98-06-04, Department of Computer Science and Engineering, University of Washington, June 1998

[2] Barták, R., Expertní systémy založené na omezujících podmínkách, PhD Thesis, Department of Computer Science, Charles University, January 1997

[3] Borning, A., Anderson, R., Freeman-Benson, B., Indigo: A Local Propagation Algorithm for Inequality Constraints, Proceedings UIST ´96, Department of Computer Science and Engineering, University of Washington,1996

[4] Borning, A., Anderson, R., Freeman-Benson, B., The Indigo Algorithm, Technical report 96-05-01, Department of Computer Science and Engineering, University of Washington, July 1996

[5] Borning, A., Freeman-Benson, B., The OTI Constraint Solver: A Constraint Library for Constructing Interactive Graphical User Interfaces, in: Proceedings of the First International Conference on Principles and Practice of Constraint Programming, pp. 624-628, Cassis, France, September 1995

[6] Borning, A., Freeman-Benson, B., Ultraviolet: A Constraint Satisfaction Algorithm for Interactive Graphics in Constraints: An International Journal 3, pp. 1-26, 1998

[7] Borning, A., Marriott, K., Stuckey, P., Xiao, Y., Solving Linear Arithmetic Constraints for User Interface Applications: Algorithm Details, Technical Report 97-06-01, Department of Computer Science and Engineering, University of Washington, September 1997

[8] Bouzoubaa, M., Neveu, B., Hasle, G., Houria III: Solver for Hierarchical Systém, Planning of Lexicographic Weight Sum Better Graph For Functional Constraints, in: the Fifth INFORMS Computer Science Technical Section Conference on Computer Science and Operations Research, Dallas, Texas, Jan. 8-10,1996

[9] Flanagan, D., Programování v jazyce JAVA, Computer Press, Praha, 1997

[10] Grand, M., JAVA - Referenční příručka jazyka, Computer Press, Praha, 1998

[11] Harvey, W., Stuckey, P.J., Borning, A., Compiling Constraint Solving Using Projection, Technical Report, Department of Computer Science, University of Melbourne, 1997

[12] Kumar, V., Algorithms for Constraint Satisfaction Problems: A Survey, in AI Magazine 13(1), pp. 32-44, 1992

[13] Marriott, K., Chok, Sitt Sen, Finlay, A., A Tableau Based Constraint Solving Toolkit for Interactive Graphical Applications, Proceedings of CP98, pp. 340-354, Pisa, October 1998

[14] Sannella, M., The SkyBlue Constraint Solver, Technical Report 92-07-02, Department of Computer Science and Engineering, University of Washington, February 1993

[15] Sannella, M., Freeman-Benson, B., Maloney, J., Borning, A., Multi-way versus One-way Constraints in User Interfaces: Experience with the DeltaBlue Algorithm, Technical Report 92-07-05, Department of Computer Science and Engineering, University of Washington, July 1992

[16] Vander Zanden, Brad, An Incremental Algorithm for Satisfying Hierarchies of Multi-way, Dataflow Constraints, Technical Report, Department of Computer Science, University of Tennessee, March 1995

[17] Sannella, M., Constraint Satisfaction and Debugging for Interactive User Interfaces, Technical Report 94-09-10, Dept. of Computer Science and Engineering University of Washington, September 1994

[18] Barták, R.,Guide to Constraint Programming, http://kti.ms.mff.cuni.cz/~bartak/constraints/index.html

[19] The Java Tutorial, http://java.sun.com/docs/books/tutorial/index.html

[20] JDK 1.1.x Documentation, http://java.sun.com/products/jdk/1.1/docs/index.html

Příloha 1 – změna interního řešiče

Program grafický editor je koncipován tak, že umožňuje snadno změnit podkladový řešič omezujících podmínek.

Jsou dvě základní možnosti, jak implementovat do aplikace jiný algoritmus:

1. při implementaci řešiče využít navržené třídy pro realizaci omezujících podmínek a proměnných (viz kapitola 3.4.3.1),

2. využít samostatnou implementaci řešiče a naprogramovat pouze její zapouzdření do rozhraní aplikace.

Dle první varianty bychom vytvořili zcela novou implementaci postavenou na třídách grafického editoru (Variable, AbstractConstraint). Tato implementace by tedy byla na těchto třídách závislá. Druhá varianta odpovídá převzetí již existující samostatné implementace a je pouze jejím zabudováním do aplikace.

Není jednoduché říci, která varianta je lepší, protože každá má své výhody. První typ může být efektivnější, protože nevyžaduje transformaci mezi jednotlivými implementacemi omezujících podmínek, zatímco druhý dovoluje využít již hotový a odladěný kód řešiče. Má však nevýhodu v tom, že třídy pro proměnné a podmínky pravděpodobně nebudou kompatibilní s rozhraním grafického editoru, a je tedy nutné provést jejich mapování.

Při implementaci jiného interního řešiče omezujících podmínek také nejspíše budeme požadovat rozšíření třídy typů podmínek. Pro zařazení nového typu podmínky je třeba zvolit vhodnou vnitřní reprezentaci této podmínky a vytvořit třídu, která vychází z třídy Constraint. Více o objektové hierarchii podmínek v kapitole 3.4.3.1.2.

Na závěr dodejme: jestliže je nový řešič úspěšně implementován, je třeba provést zásah do centrálního modulu grafického editoru. Ve třídě MainPanel je metoda getConstraintSolver, která vytváří instanci třídy ConstrainSolver. Jde o objekt, který zajišťuje správu popisných tříd jednotlivých grafických objektů. Jako parametr při jeho vytváření figuruje instance interního řešiče. Právě tu je třeba nahradit instancí nové implementace. Následující kód ukazuje dvě verze metody getConstraintSolver: původní a pozměněnou. Předpokládejme, že třída implementující nový řešič se jmenuje AnotherSolver.

```
// původní verze
private ConstraintSolver getConstraintSolver(){
if (constraintSolver == null)
      constraintSolver = new ConstraintSolver(new CassowarySolver());
 return constraintSolver;
}
```

```
// nová verze
private ConstraintSolver getConstraintSolver(){
if (constraintSolver == null)
      constraintSolver = new ConstraintSolver(new AnotherSolver());
 return constraintSolver;
}
```

Řešič Cassowary, který je využit v současné implementaci grafického editoru, je příkladem samostatné implementace. Jeho zapouzdření do standardního rozhraní, které realizuje třída CassowarySolver, lze tedy považovat za demonstraci toho, jak může být v grafickém editoru externí řešič využit. Probereme si tedy třídu CassowarySolver podrobněji. Neuvedeme zde ale celý kód třídy, zaměříme se jen na jeho nejzajímavější části. Zbytek je většinou jen variací na stejné téma.

Jak již bylo řečeno, interní řešič Cassowary je samostatnou implementací. Má tedy vlastní třídy pro realizaci omezujících podmínek, takže je třeba udržovat v třídě CassowarySolver dvě převodní tabulky. Jedna z nich zajišťuje převod mezi jednotlivými realizacemi proměnných, mezi typy Variable (z implementace grafického editoru) a ClVariable (z implementace řešiče Cassowary). Druhá tabulka slouží podobnému účelu, převádí mezi sebou třídy podmínek Constraint (z implementace grafického editoru) a ClConstraint (z implementace řešiče Cassowary).

Fragment kódu třídy CassowarySolver ukazuje metody, které realizují zmiňované převodní tabulky, a metodu addConstraint, na které lze demonstrovat, jak je zpracován požadavek na přidání nové podmínky do systému.

```
package cz.cuni.mff.rmecl.gce;

import java.util.*;
import EDU.Washington.grad.gjb.cassowary.*;
public class CassowarySolver implements SolverInterface {
// cassowary solver
      ClSimplexSolver solver = null;

      Hashtable clVariables = new Hashtable();
      Hashtable ClConstraints = new Hashtable();

// konstruktor vytvoří instanci interního řešiče ClSimplexSolver
public CassowarySolver() {
      super();
      solver = new ClSimplexSolver();
}
// realizace přidání podmínky c do systému
public boolean addConstraint(Constraint c) {
      try
      {
            solver.addConstraint(getClConstraint(c));
      }
      catch (ExCLError ex) {
```

```
            handleException(ex);
            return false;
      }
      return true;
}
/**
 * transformuje Constraint na cassowary ClConstraint
 */
private ClConstraint getClConstraint(Constraint con) {
ClConstraint clc = (ClConstraint)ClConstraints.get(con);
if (clc == null)
// jde o novou podmínku
{
 try {
  if (con instanceof EditConstraint)
// převede EditConstraint na ClEditConstraint
  {
      ClVariable v = getClVariable(con.getVariables()[0]);
      clc = new ClEditConstraint(v,getClStrength(ConstClass.MEDIUM));
  }
 else if (con instanceof StayConstraint)
// převede StayConstraint na ClStayConstraint
  {
      ClVariable v = getClVariable(con.getVariables()[0]);
      clc = new ClStayConstraint(
            v,getClStrength(ConstClass.WEAK),con.getWeigth());
  }
  else if (con instanceof LinearConstraint)
// převede LinearConstraint na ClLinearEquation nebo ClLinearInequality
  {
      ClStrength str = getClStrength(con.getStrength());
      Variable[] avar = con.getVariables();
      double[] cons = ((LinearConstraint)con).getConstants();
      double c0 = ((LinearConstraint)con).getC0();
      int op = ((LinearConstraint)con).getOp();
      ClLinearExpression exp = new ClLinearExpression(-c0);
      for (int i = 0; i<avar.length;i++)
            exp.addVariable(getClVariable(avar[i]),cons[i]);
      if (op == LinearConstraint.EQ) clc = new ClLinearEquation(exp,str);
      else
      {
        ClLinearExpression exp2 = new ClLinearExpression(0.0);
        if (op == LinearConstraint.LEQ)
             clc = new ClLinearInequality(exp,CL.LEQ,exp2,str);
        else clc = new ClLinearInequality(exp,CL.GEQ,exp2,str);
      }
  }
  else return null;
// zařaď podmínku do transformačního seznamu
 ClConstraints.put(con,clc);
 } catch (ExCLError ex) {handleException(ex);}
}
return clc;
}
/**
 * transformuje Variable na cassowary ClVariable
 */
private ClVariable getClVariable(Variable var) {
      ClVariable clv = (ClVariable)clVariables.get(var);
      if (clv == null)
      {
```

```
            clv = new ClVariable(var.getName(),(double)var.getValue());
            clVariables.put(var,clv);
      }
      return clv;
}
// provede synchronizaci hodnot proměnných s hodnotami interních proměnných
private void updateValues() {
 Enumeration k = clVariables.keys();
 Enumeration e = clVariables.elements();
 while (k.hasMoreElements())
 {
 ((Variable)k.nextElement()).setValue(
      (int)((ClVariable)e.nextElement()).value());
 }
}
// kód dalších povinných metod rozhraní SolverInterface
.........
}
```

Z výše uvedeného kódu je patrné, že vlastní realizace volání interního řešiče není implementačně náročná, nejobtížnější je vlastní převod mezi reprezentacemi podmínek a proměnných jednotlivých algoritmů.

Příloha 2 – Zařazení nového grafického objektu do aplikace

Otevřenost návrhu grafického editoru lze využít také v případě, když nám nepostačuje stávající výběr grafických objektů, které grafický editor nabízí. Popíšeme zde způsob, jak lze aplikaci rozšířit o nový grafický objekt. Abychom nemluvili příliš obecně, realizujme tento postup na příkladě.

Pokusíme se navrhnout a naprogramovat grafický útvar skládající se z trojúhelníku a jednoho bodu. Objekt bude mít jedinou volitelnou podmínku: samostatný bod je těžištěm trojúhelníku.

Celý postup rozdělíme do několika fází.

- Nejdříve je nutné vytvořit vhodný návrh grafického objektu.

Útvar bude tvořen 4 body A,B,C,D. Body A,B,C jsou body trojúhelníku, bod D je pak samostatný bod. Jednotlivé body mají své volitelné podmínky (viz kapitola 3.2.1). Celý objekt pak má jedinou podmínku, kterou popisují následující dvě rovnice:

$(x_A+x_B+x_C)/3 = x_D$ pro naše potřeby: $x_A+x_B+x_C-3\,x_D= 0$

$(y_A+y_B+y_C)/3 = y_D$ pro naše potřeby: $y_A+y_B+y_C-3\,y_D= 0$.

Jde o základní volitelnou podmínku, kterou uživatel může pouze povolit nebo zakázat.

Dále vytvoříme třídy, které realizují jednak grafický objekt jako celek, jednak jeho grafickou podobu a popisnou část.

- Třída ETriangle zastupuje grafický objekt jako celek

```
public class ETriangle extends EAbstract{
// odkazy na jednotlivé body
private EPoint p1,p2,p3,p4;
//konstruktor třídy: hodnoty ax,ay udávají pozici, kde má být objekt
//umístěn
 public ETriangle(int ax, int ay){
// vytvoříme body
  p1 = new EPoint(ax,ay,1.0);
  p2 = new EPoint(ax,ay,2.0);
  p3 = new EPoint(ax,ay,4.0);
  p4 = new EPoint(ax,ay,8.0);
// vytvoříme grafickou reprezentaci objektu
  graphicObject = new GTriangle(
            (GPoint)(p1.getGObject()),(GPoint)(p2.getGObject()),
            (GPoint)(p3.getGObject()),(GPoint)(p4.getGObject()));
// vytvoříme popisnou část objektu
  constraintObject = new CTriangle(
            (CPoint)(p1.getCObject()),(CPoint)(p2.getCObject()),
            (CPoint)(p3.getCObject()),(CPoint)(p4.getCObject()));
// korektně nastavíme vazby mezi objekty
  p1.setParent(this);p2.setParent(this);
  p3.setParent(this);p4.setParent(this);
```

```
  sons = new EAbstract[4];
  sons[0] = p1;sons[1] = p2;sons[2] = p3;sons[3] = p4;
// vytvoříme jednoznačné jméno objektu
  counter++;
  name = "TrojúhelníkII "+counter;
 }
}
```

Třída ETriangle je jednoduchá, protože jedinou metodu, kterou musíme napsat, je její konstruktor. V něm vytvoříme jednotlivé objekty pro body spolu s grafickou a popisnou reprezentací trojúhelníku.

- Třída GTriangle zajišťuje vnější vzhled.

```
public class GTriangle extends GAbstract {
// odkazy na grafické reprezentace jednotlivých bodů
      GPoint p1,p2,p3,p4;
// konstruktor objektu
public GTriangle(GPoint g1, GPoint g2, GPoint g3, GPoint g4){
      p1 = g1;
      p2 = g2;
      p3 = g3;
      p4 = g4;
// naplní pomocné proměnné
      variablesX = new Variable[4];
      variablesX[0] = p1.getXVar();
      variablesX[1] = p2.getXVar();
      variablesX[2] = p3.getXVar();
      variablesX[3] = p4.getXVar();
      variablesY = new Variable[4];
      variablesY[0] = p1.getYVar();
      variablesY[1] = p2.getYVar();
      variablesY[2] = p3.getYVar();
      variablesY[3] = p4.getYVar();
}
// vrací odkaz objekt, kterému odpovídají souřadnice [ax,ay]
public GAbstract contains(int ax,int ay) {
      GAbstract pC = p1.contains(ax,ay);
      if (pC != null) return pC;
      pC = p2.contains(ax,ay);
      if (pC != null) return pC;
      pC = p3.contains(ax,ay);
      if (pC != null) return pC;
      pC = p4.contains(ax,ay);
      if (pC != null) return pC;
      return null;
}
// vykresluje objekt na ploše
public void draw(java.awt.Graphics g) {
      java.awt.Color col = g.getColor();
// vykreslí body
      p1.draw(g);
      p2.draw(g);
      p3.draw(g);
      p4.draw(g);
      if (selected)     g.setColor(ConstClass.selectedColor);
// nakreslí trojúhelník
      g.drawLine(p1.getX(),p1.getY(),p2.getX(),p2.getY());
      g.drawLine(p2.getX(),p2.getY(),p3.getX(),p3.getY());
      g.drawLine(p3.getX(),p3.getY(),p1.getX(),p1.getY());
```

```
      if (selected)     g.setColor(col);
}
// podle parametru aSel zvýrazní objekt nebo zruší zvýraznění objektu
public void setSelection(boolean aSel) {
      selected = aSel;
      p1.setSelection(aSel);
      p2.setSelection(aSel);
      p3.setSelection(aSel);
      p4.setSelection(aSel);
}
}
```

V třídě GTriangle je tedy nutné uvést kódy čtyř klíčových metod: konstruktor a metody draw, contains a setSelection. Všechny mají souvislost s vnějším vzhledem objektu. Metodu contains aplikace využívá, když hledá objekt, který uživatel myší označil a chce s ním dále pracovat. Metoda setSelection ovlivňuje vzhled objektu. Je-li objekt vybraný, aby s ním mohl uživatel pracovat, má jinou barvu.

- Poslední třídou, kterou budeme muset vytvořit, je třída CTriangle. Jde o popisnou část celého objektu, ve které se objeví námi požadované omezující podmínky.

```
public class CTriangle extends CAbstract{
 CPoint p1,p2,p3,p4;
// odkaz na volitelnou podmínku
 private AbstractConstraint optCons1;
 private OYesNo opt1;
// konstruktor
 public CTriangle(CPoint c1, CPoint c2,CPoint c3,CPoint c4){
  p1 = c1;
  p2 = c2;
  p3 = c3;
  p4 = c4;
 }
// vrací seznam všech volitelných podmínek objektu
 public Vector getAllOptions(boolean withSons){
  Vector vec = new Vector();
  vec.addElement(getOption1());
  if (withSons){
   ConstClass.concat(vec,p1.getAllOptions(withSons));
   ConstClass.concat(vec,p2.getAllOptions(withSons));
   ConstClass.concat(vec,p3.getAllOptions(withSons));
   ConstClass.concat(vec,p4.getAllOptions(withSons));
  }
  return vec;
 }
// udává, která proměnná objektu je hraniční
 public Variable getBorderVariable(int borderType){
   Variable var = null;
   switch (borderType){
      case ConstClass.LEFT:
            var = p1.getXVar();
      if (p2.getXVar().getValue() < var.getValue()) var = p2.getXVar();
      if (p3.getXVar().getValue() < var.getValue()) var = p3.getXVar();
      return var;
      case ConstClass.RIGHT:
            var = p1.getXVar();
      if (p2.getXVar().getValue() > var.getValue()) var = p2.getXVar();
```

```
      if (p3.getXVar().getValue() > var.getValue()) var = p3.getXVar();
      return var;
      case ConstClass.TOP:
            var = p1.getYVar();
      if (p2.getYVar().getValue() < var.getValue()) var = p2.getYVar();
      if (p3.getYVar().getValue() < var.getValue()) var = p3.getYVar();
      return var;
      case ConstClass.BOTTOM:
            var = p1.getYVar();
      if (p2.getYVar().getValue() > var.getValue()) var = p2.getYVar();
      if (p3.getYVar().getValue() > var.getValue()) var = p3.getYVar();
      return var;
   }
 return null;
 }
// vrací seznam všech stálých podmínek objektu (včetně podmínek bodů)
 public Vector getConstraints(){
   if (constraints == null){
         constraints = new Vector();
         Vector pc = null;
         pc = p1.getConstraints();
         ConstClass.concat(constraints,pc);
         pc = p2.getConstraints();
         ConstClass.concat(constraints,pc);
         pc = p3.getConstraints();
         ConstClass.concat(constraints,pc);
         pc = p4.getConstraints();
         ConstClass.concat(constraints,pc);
   }
   return constraints;
 }
// vytvoření vlastní volitelné podmínky
 private OYesNo getOption1(){
  if (opt1 == null){
   opt1 = new OYesNo(getOptionConstraint1(ConstClass.STRONG),true,"bod D je
těžiště",ConstClass.STRONG);
  } else{
if (opt1.getNewConstraintNeeded()){
         opt1.setConstraint(getOptionConstraint1(opt1.getStrength()));
   }
  }
  return opt1;
 }
// vytvoření omezujících podmínek, které tvoří základ volitelné podmínky
 private AbstractConstraint getOptionConstraint1(int str){
   Constraint[] inta = new Constraint[2];
   Variable[] va = new Variable[4];
   va[0] = p1.getXVar(); va[1] = p2.getXVar();
   va[2] = p3.getXVar();   va[3] = p4.getXVar();
   double[] da = {1,1,1,-3};
// podmínka pro x-ové souřadnice
   inta[0] = new LinearConstraint(da,va,LinearConstraint.EQ,0.0,str);
   va = new Variable[4];
   va[0] = p1.getYVar(); va[1] = p2.getYVar();
   va[2] = p3.getYVar();   va[3] = p4.getYVar();
   double[] da2 = {1,1,1,-3};
// podmínka pro y-ové souřadnice
   inta[1] = new LinearConstraint(da2,va,LinearConstraint.EQ,0.0,str);
// slučující multi podmínka
   optCons1 = new MultiConstraint(inta,str);
  return optCons1;
```

```
 }
}
```

Může se zdát, že třída CTriangle je mnohem komplikovanější než předchozí třídy. Je to způsobeno tím, že je nutné některé pomocné metody přeprogramovat, aby odpovídaly potřebám objektu. Klíčové jsou dvě metody getOption1 a getOptionConstraint1. Ty jsou jádrem, protože vytvářejí vlastní volitelnou podmínku, která z bodu D dělá těžiště trojúhelníku ABC.

Nyní zbývá jen zařazení grafického objektu ETriangle do aplikace. Pro to je třeba udělat zásahy do centrálního modulu – tříd MainPanel a MainFrame.

- Ve třídě MainPanel je potřeba opravit metodu insertObject. Rozšíříme její kód o vytvoření nového objektu ETriangle. Přidaná část kódu je zvýrazněna tučně.

```
private void insertObject(int ax, int ay){
      if(insertingObject.equals("Bod"))
            addEditorObject(new EPoint(ax,ay));
      else if(insertingObject.equals("Usecka"))
            addEditorObject(new ELine(ax,ay));
      else if(insertingObject.equals("Ctyruhelnik"))
            addEditorObject(new E4Line(ax,ay));
      else if(insertingObject.equals("3body"))
            addEditorObject(new E3Point(ax,ay));
      else if(insertingObject.equals("Trojuhelnik"))
            addEditorObject(new ETriangle(ax,ay));
      getDrawPanel1().repaint();
 insertingObject = null;
 }
```

- Na závěr je třeba vykonat zásah v uživatelském rozhraní.

Je nutné dát uživateli možnost, aby mohl grafický objekt trojúhelník přidat do systému. Bylo by tedy třeba změnit modul ControlPanel, který obsahuje jednotlivá tlačítka. My nyní pro jednoduchost zvolíme jinou variantu. Změníme pouze nastavení metody performRequest ve třídě MainFrame tak, aby tlačítko pro původní přidání čtyřúhelníku přidalo nyní náš trojúhelník. Změněná část kódu je opět zvýrazněna tučně.

```
public void performRequest(int type) {
      switch (type) {
            case 1: getMainPanel1().executeCommand("Bod"); break;
            case 2: getMainPanel1().executeCommand("Usecka"); break;
            case 3: getMainPanel1().executeCommand("Trojuhelnik"); break;
      ...
      }
}
```

Tímto je grafický editor rozšířen o nový grafický objekt „trojúhelník s těžištěm“. Zcela analogicky lze postupovat u jakéhokoli jiného objektu. Jediným předpokladem je, že takový objekt lze popsat sadou omezujících podmínek, jejichž typ podporuje modul řešiče omezujících podmínek- tedy v případě řešiče Cassowary lineárními rovnicemi a nerovnicemi.

Příloha 3 –uživatelská příručka

Realizovaná implementace grafického editoru pracujícího nad omezujícími podmínkami má snadné a intuitivní ovládání. Protože nejde o rozsáhlou aplikaci, je možné se se způsoby jejího ovládání seznámit velice rychle. Není proto třeba věnovat uživatelské příručce mnoho místa.

Nejprve se zaměříme na popis základních funkcí a principů ovládání aplikace a potom takto získané znalosti použijeme na praktické ukázce. Pokusíme se vytvořit jednoduchý obrazec a ukážeme si, jak se grafický editor chová při změnách proporcí tohoto obrazce.

Jak již bylo řečeno, program je postaven na grafických objektech. Veškeré operace, které grafický editor nabízí, jsou tedy operacemi nad grafickými objekty. Lze je vkládat, upravovat jejich volitelné podmínky, měnit jejich polohu a proporce a samozřejmě také mazat. Dále je možné vytvářet skupiny objektů a upravovat volitelné podmínky pro tyto skupiny objektů. Odstraněním skupiny pak nedojde k fyzickému smazání objektů, které do skupiny patří, pouze jsou zrušeny všechny volitelné podmínky, které byly nad odstraněnou skupinou vytvořeny.

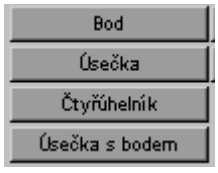
Ke správnému pochopení následujícího textu je třeba se seznámit s některými pojmy z kapitoly 3.4.3.3. Jde především o pojmy: kreslící plocha, seznam objektů, seznam skupin a dialog volitelných podmínek. Jako samozřejmý předpoklad je také chápána jistá zkušenost jednak s programy v grafických uživatelských prostředích vůbec, jednak s grafickými editory.

Práce s objekty

Nejdříve se seznámíme s principy práce s jednotlivými grafickými objekty.

Vytvoření objektu

Grafický objekt je přidáván ve dvou fázích. Nejdříve je třeba pomocí tlačítek nebo odpovídajícími položkami v menu Nový objekt vybrat, který objekt chceme vytvořit.

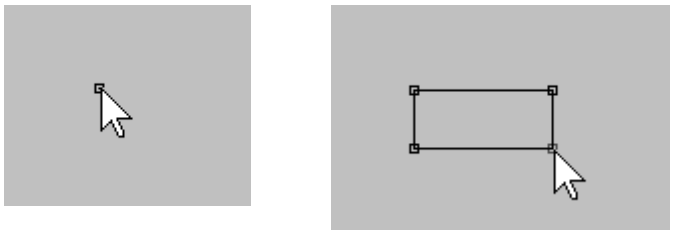


tlačítka pro přidání objektu

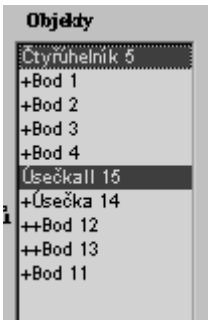
Ve druhé fázi je pak nutné ukázat myší na místo na kreslící ploše, kam chceme objekt umístit. Grafický objekt se po stisknutí levého tlačítka myši umístí na požadované místo tak, že to vypadá, jako kdyby byl grafickým bodem. Podržíme-li však tlačítko myši stisknuté

i poté, co se grafický objekt objeví na kreslící ploše, lze tažením myši docílit změny jeho proporcí.

před tažením objektu po tažení objektu



V grafickém editoru může být zároveň umístěno více grafických objektů. Je tedy třeba mít možnost, jak některý z nich označit, aby s ním bylo možné dále pracovat. Pokud chceme pracovat s grafickým bodem, můžeme jej označit myší přímo na kreslící ploše. Jinou alternativou je výběr objektu v seznamu objektů. Tato možnost je pak jediným způsobem označení pro jiné grafické objekty (úsečka, čtyřúhelník).

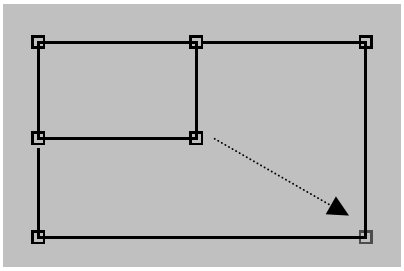


více vybraných objektů

Změna polohy nebo proporce objektu

Na kreslící ploše můžeme s objekty pracovat využitím techniky drag&drop. Objekt, se kterým chceme pracovat, označíme a po dobu, kdy máme stisknuté tlačítko myši, se objekt pokouší pohybovat tak, jak se pohybuje kurzor myši. Je nutné si ale uvědomit, že ne vždy může grafický objekt vyhovět našemu přání. Jestliže je totiž svázán nějakými omezujícími podmínkami, jejichž preference je vyšší než preference edit podmínek, musí jejich platnost dodržovat. Porušeny tedy mohou být pouze ty podmínky, které mají preferenci stejnou (medium) nebo nižší (weak).

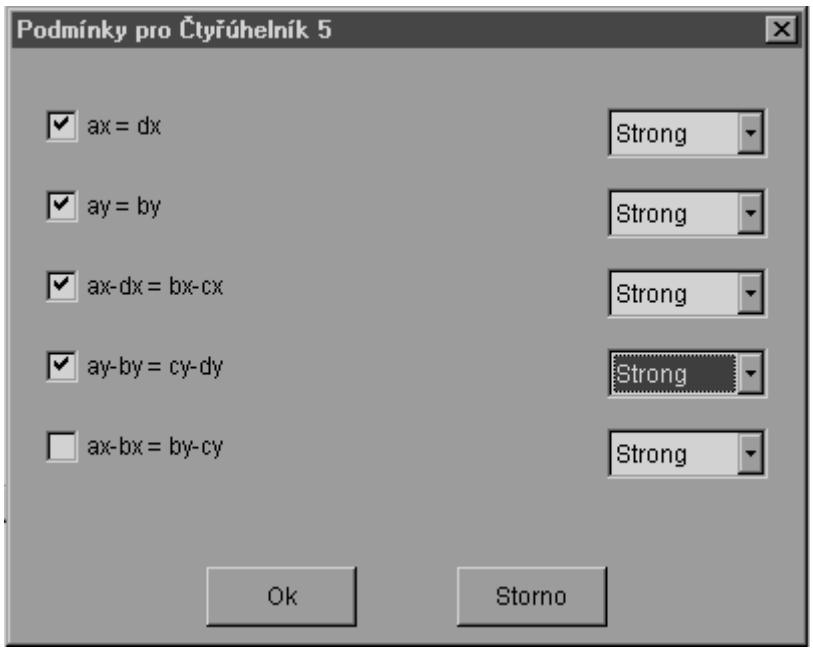
změna proporcí objektu
tažením bodu



Úprava volitelných podmínek objektu

Vybereme-li nějaký grafický objekt, můžeme měnit nastavení jeho volitelných podmínek. Stisknutím tlačítka Volitelné podmínky objektu se vyvolá dialog, ve kterém jsou zobrazeny všechny volitelné podmínky daného objektu. Pro každou podmínku lze 1. zadat, zda má být aktivní nebo pasivní, 2. určit její preferenci a 3. u parametrických podmínek lze navíc určit hodnotu jejího parametru. Potvrdíme-li dialog tlačítkem *Ok*, promítnou se změny v celém systému, naopak tlačítko *Storno* uvede systém do stavu, ve kterém se nacházel před vyvoláním dialogu.

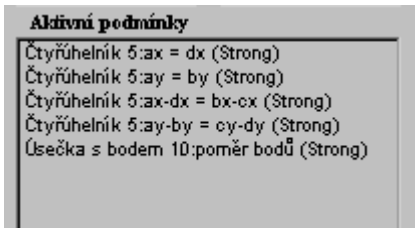
dialog volitelných podmínek
objektu čtyřúhelník



V souvislosti se zadáváním volitelných podmínek je třeba ještě upozornit na jednu skutečnost. Pokud se pokusíme objekt příliš omezit, neboli svázat jej povinnými podmínkami, které nemohou být splněny zároveň, jsou podmínky, které k tomuto přílišnému omezení

vedou, automaticky deaktivovány. Na tuto skutečnost je pak uživatel upozorněn varovným hlášením.

Všechny aktivní volitelné podmínky, které systém grafických objektů omezují, se objevují v seznamu aktivních podmínek. Jde pouze o informativní seznam, který může dát odpověď na otázku, proč se grafický editor v daném okamžiku chová nějakým způsobem. Například jestliže není možné přesunout grafický bod, je pravděpodobné, že v seznamu je podmínka, která tento bod svazuje s konkrétní souřadnicí. O tom bychom se sice mohli přesvědčit i vyvoláním dialogu na úpravu podmínek, ale v seznamu je tato podmínka vidět hned.



seznam aktivních volitelných podmínek

Smazání objektu

Jestliže chceme některý objekt smazat, stačí jej označit a stisknout tlačítko Smazat objekt. Jsou dvě příčiny, které mohou zabránit smazání objektu.

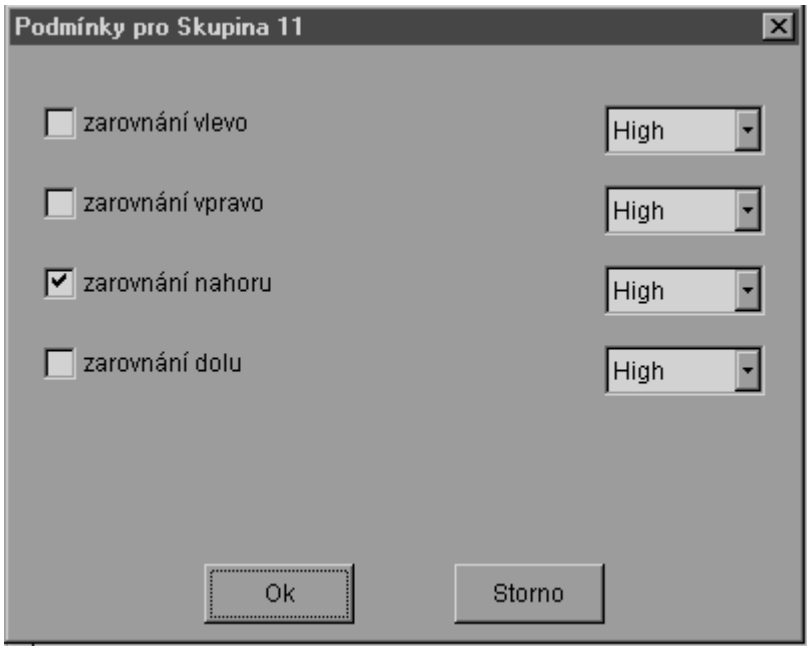
- Vybraný objekt má nějaký nadřízený objekt (například bod úsečky). Smazat můžeme pouze celý nadřízený objekt, proto pro vybraný objekt není tlačítko *Smazat objekt* přístupné.
- V případě, že tento objekt je prvkem nějaké aktivní skupiny, není tato operace provedena a chceme-li objekt přesto smazat, je nutné nejdříve zrušit ty skupiny, jejichž je grafický objekt prvkem (viz dále).

Práce se skupinami objektů

Dosavadní text popisoval způsoby, jak zadávat, měnit a mazat jednotlivé grafické objekty. Abychom mohli některé operace provádět zároveň nad několika objekty, je třeba se seznámit s tzv. *skupinami objektů.*

Každá skupina objektů je tvořena alespoň dvěma různými grafickými objekty. Získáme ji tedy tak, že v seznamu objektů postupně vybereme všechny požadované grafické objekty.

Takto vytvořenou skupinu lze omezovat zadáním volitelných podmínek. K tomu slouží tlačítko Volitelné podmínky skupiny. Objeví se dialog všech volitelných podmínek pro skupiny objektů. Způsob ovládání je stejný jako u dialogu podmínek pro jednotlivé objekty (viz kap. Úprava volitelných podmínek objektu).

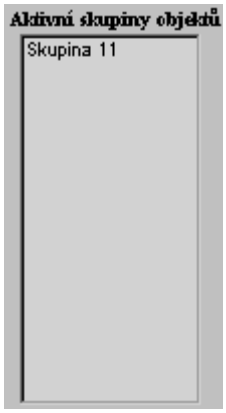


dialog volitelných podmínek skupiny

Jestliže skupinu omezíme některou volitelnou podmínkou, objeví se tato skupina v seznamu aktivních skupin. Zde setrvá až do doby, kdy všechny aktivní volitelné podmínky skupiny deaktivujeme. Toho lze docílit jednak v dialogu volitelných podmínek skupiny, jednak pomocí tlačítka Smazat podmínky skupiny.

Seznam aktivních skupin kromě toho, že informuje, také nabízí způsob jak rychle označit všechny grafické objekty, které do skupiny patří. To oceníme zvláště tehdy, jestliže v systému je více aktivních skupin a je nutné s nimi střídavě pracovat.

seznam aktivních skupin objektů

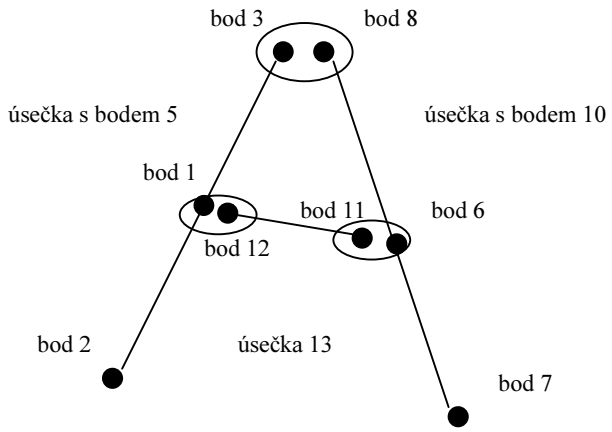


Tip: V seznamu objektů máme možnost vybírat více objektů. Jednoduchý způsob, jak seznam nastavit, aby nebyl vybraný žádný objekt, je stisknutím tlačítka myši na kreslící ploše v místě, kde není žádný grafický objekt.

Příklad

Na následujícím příkladě demonstrujeme praktické užití jednotlivých ovládacích prvků. Pokusíme se nakreslit složený grafický objekt, který by připomínal písmeno A. Složíme ho ze dvou úseček s bodem (viz 3.2.4) a jedné úsečky (viz 3.2.2). Dále omezíme některé dvojice bodů tak, aby splývaly. Nakonec omezíme vrchol písmene A tak, aby zůstával na pevné pozici, a z obecné úsečky uděláme rovnoběžku s osou x. Na obrázku jsou schématicky znázorněny jednotlivé použité grafické objekty a způsob omezení dvojic bodů.

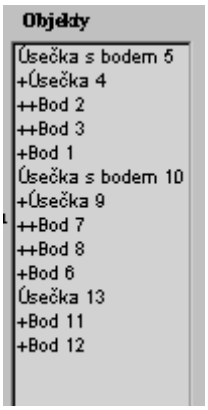
schéma objektu Písmeno A



Postup:

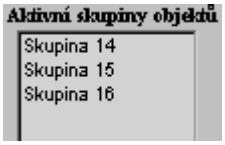
1. Přidáme jednotlivé grafické objekty pomocí tlačítek *Úsečka s bodem* a *Úsečka*. Seznam objektů bude vypadat následnovně.

seznam objektů



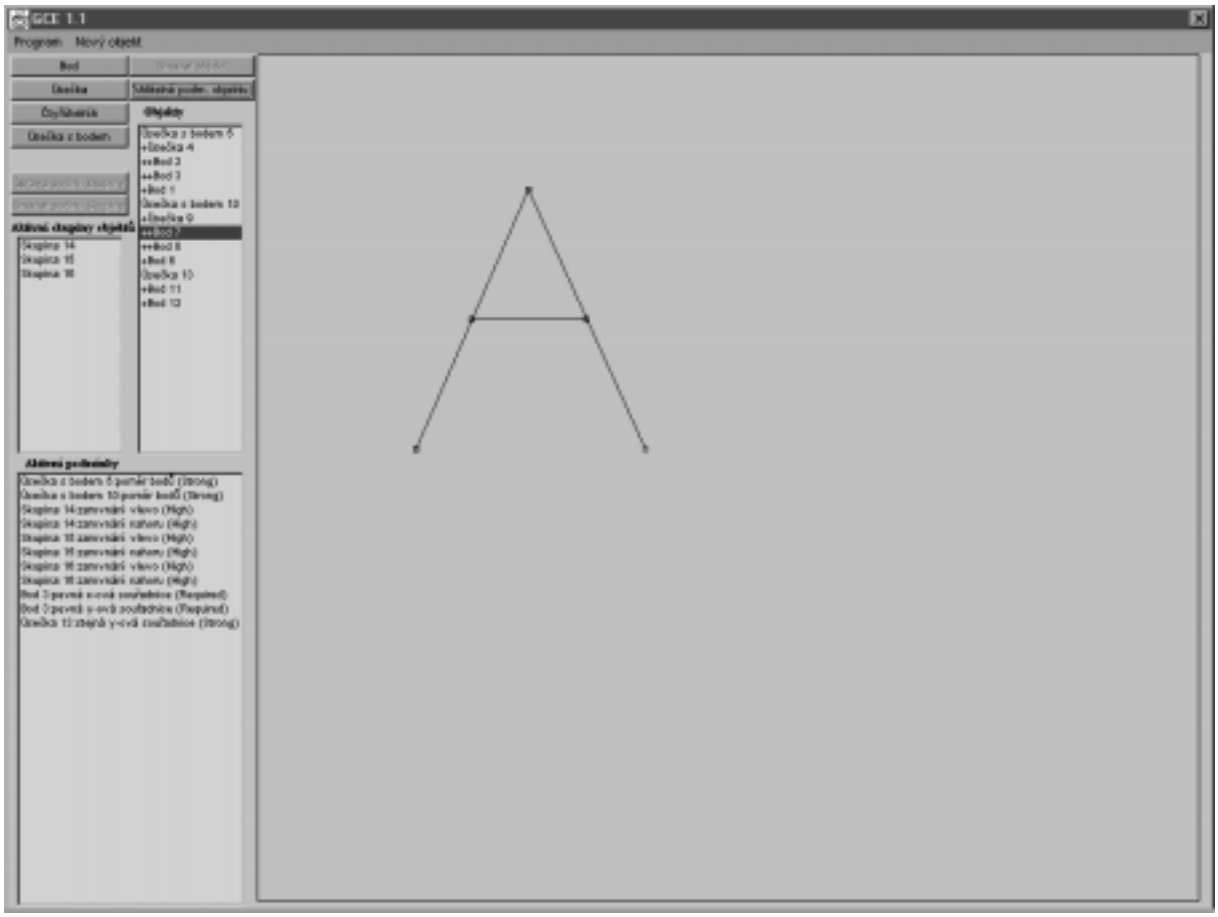
2. Spojíme tyto dvojice bodů: [bod3;bod8], [bod1;bod12], [bod6;bod11]. Abychom omezili dvojici bodů, musíme např. pro tuto dvojici vytvořit aktivní skupinu se zarovnáním nahoru a vlevo (viz 3.2). Pro každou dvojici aplikujeme tento postup. Vybereme tedy nejdříve oba body dvojice a po stisknutí tlačítka *Úprava podm.* skupiny zvolíme v zobrazeném dialogu výše uvedené zarovnání. Tuto volbu potvrdíme tlačítkem *Ok*. Na obrázku je seznam aktivních skupin, jak by měl vypadat po správném provedení kroku 2.

seznam skupin



3. Omezíme bod 3 (nebo bod 8 – splývají) na pevné souřadnice x i y. Po označení bodu 3 v seznamu objektů stiskneme tlačítko *Volitelné podm. objektu* a v zobrazeném dialogu vybereme odpovídající volitelné podmínky.
4. Podobným způsobem jako v 3. omezíme úsečku 13 na rovnoběžku s osou x (v dialogu vybereme volitelnou podmínku *Stejná y-ová souřadnice*).

Konečnou podobu vytvořeného grafického objektu ukazuje obrázek.



obrazovka zachycující vytvořený
objekt Písmeno A

Na následujících obrázcích je pak vidět, jak se při tažení bodu 7 mění proporce celého grafického objektu.

série objektů Písmeno A