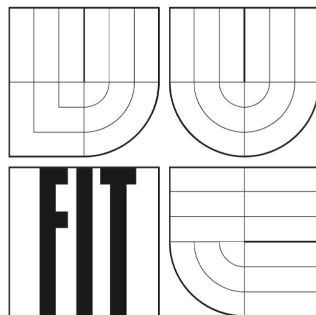
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



Diplomová práce

2007 PETRA HRABCOVÁ

Propojení simulační knihovny SIMLIB s jazykem Prolog

© Petra Hrabcová, 2006.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracovala samostatně pod vedením Ing. Martina Hrubého Ph.D.
Další informace mi poskytl Ing. Michal Cerhák.
Uvedla jsem všechny literární prameny a publikace, ze kterých jsem čerpala.

……………………
Petra Hrabcová
22.ledna 2007

Abstrakt

Tato práce je zaměřena na oblast heterogenního modelování, zejména na modelování zaměřené na spolupráci programovacích jazyků C++ a Prolog.
Výchozím bodem pro tuto práci je navázání na předchozí výzkum (semestrální projekt) v oblasti heterogenního modelování. Během tohoto výzkumu byl vyvinut prototyp knihovny, která umožňuje spolupráci výše zmíněných programovacích jazyků. Prototyp knihovny byl v rámci této diplomové práce dokončen a byly vytvořeny případové studie, které mají prokázat jeho použitelnost. Případové studie, které byly vytvořeny za použití simulační knihovny SIMLIB/C++, se zabývají zejména problémy z oblasti umělé inteligence jako je průchod stavovým prostorem.
Hlavním přínosem práce je srovnání simulací provedených bez použití metod, které nám nabízí umělá inteligence (tedy algoritmů pro průchod stavovým prostorem), a simulací, při kterých jsme použili metod umělé inteligence (prostřednictvím knihovny).

Klíčová slova

Heterogenní simulace, heterogenní modelování, C++, Prolog, umělá inteligence, SIMLIB/C++, propojení Prologu a C++, objektový návrh knihovny pro propojení Prologu a C++, SIMLIB/C++

Poděkování

Ráda bych poděkovala svému vedoucímu diplomové práce, Ing. Martinu Hrubému, Ph.D., za mnohé cenné rady, které jsem beze zbytku využila při tvorbě této práce. Dále bych ráda poděkovala Ing. Michalu Cerhákovi za podporu v oblasti programovacího jazyka C++ a jeho efektivního využití při implementaci případových studií.

Abstract

This thesis is focused on the multimodeling area, especially on the cooperation of the C++ language and the Prolog language.
The recent research is established on my semester study, which also dealt with the multimodeling area. During this research a prototype of interconnection library for cooperation of above mentioned programming languages was developed. This prototype of the library was finished within the scope of this thesis and some case-studies were created, too, using also another simulation library - SIMLIB/C++. These case-studies have their focus in the problems of artificial intelligence.
The main benefit of this thesis is the confrontation of methods with and without using artificial intelligence.

Keywords

Programming languages Prolog and C++, interconnection of Prolog and C++, object-oriented design of a library for the connection of Prolog and C++ programming languages, SIMLIB/C++, heterogenous modeling, multimodeling, multi-formalism modeling

Obsah

Úvod ... 7
1.1 Použitá terminologie ... 8
1.2 Návaznost na předchozí projekty ... 9
2 Subsystémy modelu ... 10
2.1 Prolog ... 10
2.1.1 Základní syntaxe ... 11
2.1.2 SWI-Prolog ... 13
2.2 SIMLIB/C++ ... 13
3 Knihovna pro propojení jazyka C++ a jazyka Prolog ... 14
3.1 Návrh ... 14
3.2 Implementace vybraných tříd ... 16
3.2.1 Rozdělení do jednotlivých souborů ... 16
3.2.2 Třída Fact ... 16
3.2.3 Třída Register ... 18
3.2.4 Třída IntData ... 18
3.2.5 Třída ListData ... 19
3.3 Použití knihovny ... 19
3.3.1 Příprava souboru s pravidly a fakty ... 20
3.3.2 Tvorba vlastní třídy ... 20
3.3.3 Získávání spojení s Prologem ... 21
3.3.4 Instance třídy a její volání ... 21
3.3.5 Překlad programu ... 21
3.3.6 Praktická ukázka ... 22
3.3.7 Rady pro práci s knihovnou ... 25
3.4 Netriviální příklad - průchod sítí zařízení ... 26
3.4.1 Analýza ... 26
3.4.2 Implementace ... 27
3.4.3 Výsledky testování ... 30
3.4.4 Možná vylepšení ... 32
4 Kruhový dopravník ... 34
4.1 Analýza ... 34
4.1.1 Použité termíny ... 34
4.1.2 Princip dopravníku ... 35
4.2 Implementace ... 37

4.2.1 Objektový návrh ... 37
4.2.2 Třída Dopravnik ... 38
4.2.3 Třída Vyrobek ... 39
4.2.4 Třídy typu linka ... 39
4.2.5 Třída Koordinator ... 40
4.2.6 Program napsaný v jazyce Prolog ... 40
4.3 Identifikované problémy ... 41
4.4 Testování ... 41
4.5 Výsledky ... 42
4.6 Diskuze výsledků ... 47
4.7 Možná vylepšení ... 48
5 Metodika pro další projekty ... 49
Závěr ... 51
Literatura ... 52
Přílohy ... 53
I. Objektový návrh knihovny ... 53
II. Rodokmen použitý v ukázkovém příkladu ... 53
III. Zdrojový kód programu rodokmen ... 54

Úvod

Simulace je výzkumná metoda, jejíž podstata spočívá v tom, že zkoumaný dynamický systém nahradíme jeho simulátorem a s ním provádíme pokusy s cílem získat informaci o původním zkoumaném systému. [1]

Modelování a simulace patří k tradičním postupům v některých technických disciplínách například v kybernetice nebo teorii automatického řízení. S rozvojem dostupných počítačů v posledních desetiletích proniklo počítačové modelování (a s ním ruku v ruce i simulace, jakožto experimentování s vytvořeným modelem) do většiny technických věd a stalo se důležitou metodou i v dalších oblastech [2]. S rozvojem těchto oblastí ale přichází i nové problémy, které bychom chtěli s využitím modelování řešit. Ne vždy lze ale celý problém bez omezení obsáhnout jedním programovacím jazykem, nebo je to z nějakého hlediska nevýhodné. Pro problémy, které je výhodnější modelovat několika různými způsoby (v několika programovacích jazycích), se používá tzv. heterogenní modelování.

V disertační práci M. Hrubého ([3]) jsou naznačeny směry vývoje oboru modelování a simulace podle článku Tuncera Örena, podle nějž heterogenní přístupy zaujímají první místo. V tomto článku rovněž definoval několik stěžejních termínů, z nichž pro tuto práci jsou podstatné pojmy multimodels[1] a mixed formalism simulation[2], jejichž význam pro tuto práci si ozřejmíme později.

Heterogenní modelování je velice rozsáhlou kapitolou, která pokrývá mnoho různých oblastí. Jeho cílem je rozklad systému do různých subsystémů, aby bylo možné každý ze subsystémů modelovat samostatně s využitím pro něj charakteristického formalismu [3]. Každý ze subsystémů může být naprosto odlišný, což poskytuje obrovskou šíři možností.

Cílem této práce je ukázat výhody, které nám může heterogenní modelování přinést. Konkrétně se budu zabývat propojením procedurálního jazyka a jazyka deklarativního, jejichž součinnost by měla být pro simulaci velmi přínosná.

Problematikou těchto jazyků a přínosu jejich propojení se zabývá druhá kapitola této práce. Tato kapitola bude věnována seznámení s programovacím jazykem Prolog, konkrétně s jeho implementací SWI-Prolog, který v této práci reprezentuje deklarativní jazyky a také seznámení se simulační knihovnou SIMLIB/C++, jež je napsána v jazyce C++, který reprezentuje jazyky procedurální. Dohromady tvoří tyto programovací jazyky multimodel, kdy simulace běží v jazyce C++

---

[1] Seskupení úzce svázaných modulů jednoho modelu. Běh simulace přepíná mezi jednotlivými moduly tak, že jeden je vždy aktivní

[2] Simulace je spojena s jiným typem zpracování znalostí (například expertní systém dodávající konkrétní parametry do simulace)

a předává řízení jazyku Prolog. C++ a Prolog navíc vyhovují termínu mixed formalism simulation, jak je popsáno výše. Prolog totiž v tomto případě zastává funkci vysoce specializovaného jazyka, který slouží jen pro konkrétní složité výpočty.

Třetí kapitola je věnována knihovně, která vznikla jako součást tohoto projektu. Umožňuje na objektové úrovni přistupovat k funkčnostem jazyka SWI-Prolog. Knihovna pro propojení jazyka C a jazyka SWI-Prolog existovala už před vznikem tohoto projektu, její použití však bylo uživatelsky nepřívětivé. Proto vznikla knihovna nová, která využívá funkčnosti knihovny původní, přidává však uživatelský komfort a objektový přístup. Prototyp této knihovny byl prezentován již v mém semestrálním projektu, jeho dokončení, náročnější testování a odstranění nedostatků bylo pak součástí této diplomové práce. V rámci třetí kapitoly se též zmíním o použití knihovny a ukáži její užití na několika příkladech.

Čtvrtá kapitola je zaměřena na netriviální příklad kruhového dopravníku, jenž je jádrem této práce. Na tomto příkladu bych ráda ukázala nejen funkčnost vytvořené knihovny, ale také její přínos v oblasti heterogenního modelování. V této kapitole se zaměřím jednak na analýzu a návrh kruhového dopravníku a jednak na samotnou implementaci. Zmíním se rovněž o průběhu testování tohoto příkladu a výsledcích experimentů, které byly nad kruhovým dopravníkem provedeny. Na konci této kapitoly pak provedu rozbor těchto výsledků a na jeho základě se pokusím navrhnout další možná vylepšení, která by vedla k dalšímu zefektivnění práce dopravníku.

Poslední kapitola je pak zaměřena na metodiku, která by měla být návodem autorům budoucích projektů. Je zde rozebrán přístup k heterogennímu modelování jako takovému a zvláštní důraz je kladen na dobré rozdělení systému na jednotlivé subsystémy.

1.1 Použitá terminologie

V této práci je použito několik termínů, jejichž vysvětlení čtenáři přiblíží probíranou tematiku.

Mezi základní pojmy patří (převzato z [4]):

- **Systém** – soubor elementárních částí (prvků systému), které mezi sebou mají určité vazby. Systémy mohou být reálné nebo nereálné (fiktivní, dosud neexistující)
- **Model** – napodobenina systému jiným systémem
  - Abstraktní model – zjednodušený popis systému, abstrahovaný ode všech nedůležitých skutečností vzhledem k cíli a účelu modelu
  - Simulační model – abstraktní model zapsaný formou programu
- **Modelování** – vytváření modelů systému
- **Simulace** – metoda získávání nových znalostí o systému experimentováním s jeho modelem
- **Verifikace modelu** – ověřuje korespondenci simulačního a abstraktního modelu (zpravidla jejich izomorfní vztah)

- **Validace modelu** (ověřování platnosti modelu) – ověření, že se pracuje s modelem adekvátním modelovanému systému

Dále bych chtěla zmínit pojmy týkající se simulační knihovny SIMLIB/C++, které budou používány v dalším textu (převzato z [5]):

- **Proces** – objekt s vlastním dynamickým chováním. Toto chování bývá popsáno zpravidla v metodě *Behavior().*
- **Zařízení** - objekt, určený k popisu specifického typu interakcí procesů, označovaného jako výlučný (exkluzivní) přístup. Problém výlučného přístupu lze formulovat takto: každý z *m* procesů systému požaduje takový přístup k abstraktnímu zařízení nebo zdroji *Z*, který vylučuje, aby v kterémkoli okamžiku sdílel zařízení *Z* více, než jeden proces. Příkladem může být benzinové čerpadlo, poštovní úředník u přepážky, nebo terminál počítače

1.2 Návaznost na předchozí projekty

Tato práce navazuje na výsledky mého semestrálního projektu. V tomto projektu jsem zpracovala návrh knihovny, která je určena pro heterogenní modelování za použití programovacích jazyků C++ a Prolog a simulační knihovny SIMLIB/C++. Tuto knihovnu jsem v rámci semestrálního projektu rovněž implementovala na úrovni prototypu a vytvořila sadu triviálních příkladů, které ukázaly její funkčnost.

Prototyp knihovny byl při ukončení semestrálního projektu funkční na úrovni, která byla vhodná pro triviální příklady. Při testování na složitějších příkladech, které jsem vytvořila během tvorby diplomové práce, byly odhaleny některé jeho nedostatky, které jsem následně odstranila. Dále jsem také implementovala některé metody nezbytné pro složitější příklady a upravila některé metody stávající, aby byl celkově vylepšen objektový návrh knihovny (např. metoda *setValue* sloužící pro ukládání informací do faktů byla nahrazena přetíženým operátorem „=“).

2 Subsystémy modelu

Každý heterogenní systém se skládá z několika subsystémů, které jsou vytvořeny s použitím různých formalismů. Každý subsystém je určen pro řešení jiných úkolů. S výhodou lze vytvořit kombinace univerzálních programovacích jazyků s jazyky specializovanými. Univerzálními jazyky rozumíme jazyky pro neomezené programování jako C/C++, Java, Smalltalk, specializované jazyky bývají navrženy pro zjednodušení programování jednoho druhu aplikace. Pro univerzální jazyky jsou často vytvářeny tzv. knihovny (hotové aplikace zobecněné na sadu funkcí).

Knihovna rozšiřuje univerzální jazyk o další funkce a jazyk tak specializuje. Nikdy však neposkytuje takový komfort jako specializovaný jazyk, protože knihovní funkce obvykle vyžadují zavedení spousty datových typů, konstant, velké škály obtížně zapamatovatelných funkcí se spoustou parametrů atd. (citováno z [3]). Typickou knihovnou, která je využívána v tomto projektu, je knihovna SIMLIB/C++, která umožňuje univerzální jazyk C++ specializovat jako jazyk simulační.

Programovací jazyky můžeme též dělit na deklarativní a procedurální. Logické jazyky, které patří do kategorie deklarativních, například jazyk Prolog, dovolují s úspěchem aplikovat principy umělé inteligence. Subsystém vytvořený v Prologu pak může být v simulaci stimulem pro správný výběr dalšího kroku. Naopak procedurální jazyky, které jsou v tomto projektu zastoupeny jazykem C++, nejsou příliš vhodné pro umělou inteligenci, ale mohou být využity pro řízení simulace.

Propojení jazyka C++ a jazyka Prolog tedy výsledné simulaci dovoluje provádět v SWI-Prologu (specializovaném jazyce) výpočty, které by byly v jazyce C++ velmi složité, a zároveň vést simulaci jednoduše, za použití simulační knihovny SIMLIB/C++, v univerzálním jazyce C++.

V rámci semestrálního projektu byla otestována knihovna, která umožňuje propojení jazyka C a jazyka SWI-Prolog. Tato knihovna je sice mocným nástrojem pro využívání funkčností Prologu, není ale příliš přívětivá k uživateli. Po prozkoumání této knihovny, jsem naznala, že pro běžného uživatele bude vhodnější, když nad touto knihovnou postavím objektovou knihovnu, která sice nebude poskytovat tak široké možnosti, ale se kterou se uživateli bude pracovat lépe.

Hlavními požadavky na knihovnu bylo především pohodlné vkládání a rušení faktů obsažených v databázi Prologu z programu v jazyce C++ a možnost pokládání dotazů nad těmito daty taktéž z programu v C++.

2.1 Prolog

Prolog se řadí mezi logické programovací jazyky. Název Prolog pochází z francouzského programmation en logique („logické programování“). Byl vytvořen Alainem Colmerauerem v roce 1972 jako pokus vytvořit programovací jazyk, který by umožňoval vyjadřování v logice místo psaní počítačových instrukcí. (převzato z [2]).

Programování v Prologu, jakožto v deklarativním jazyce, je vlastně formulací známých faktů, pravidel a cílů. K těmto cílům se pak pokoušíme dobrat pomocí odvozování z daných faktů. Tyto fakty a pravidla jsou zapsány pomocí vhodně zapsaných Hornových klauzulí[3], jež jsou ukládány do databáze Prologu.

Abychom dokázali sestrojit program v Prologu, je třeba se soustředit na dvě základní otázky – jaké formální skutečnosti a vztahy se v řešeném problému vyskytují a které vztahy jsou vzhledem k řešení problému pravdivé (volně přeloženo z [6]).

Programování v Prologu můžeme rozdělit na 3 fáze:

1. specifikace faktů o objektech a o vztazích mezi nimi
2. specifikace pravidel, které se týkají objektů a vztahů mezi nimi
3. pokládání dotazů o objektech a o vztazích mezi nimi.

Protože je Prolog koncipován jako dialogový programovací jazyk, uživatel dostane na každý dotaz okamžitou odpověď.

V Prologu se používají dva základní režimy - režim konzultační a režim dotazování. V konzultačním režimu se píší programy (zadávají se známá fakta a vztahy mezi nimi). V režimu dotazování pak klademe otázky. Otázky se kladou pomocí predikátů, které jsou v daném programu definovány.

2.1.1 Základní syntaxe

Základní strukturou tohoto programovacího jazyka je term. Termem mohou být konstanty, proměnné nebo složitější struktury.

Konstantu chápeme jako číslo (celé nebo reálné), nebo jako atom. Atomem je posloupnost písmen, číslic a speciálních znaků, která začíná malým písmenem nebo je uzavřena do apostrofů. Konstanty slouží k pojmenování objektů nebo vztahů mezi těmi objekty.

Proměnnou chápeme jako posloupnost znaků a číslic, která začíná písmenem velkým, nebo znakem „_“. Pokud chceme říct, že hodnota této proměnné pro nás není užitečná, použijeme znak „_“ samostatně jako jméno proměnné.

Prolog disponuje jedinou standardní strukturou – seznamem. Seznam se v Prologu zapisuje do hranatých závorek a jednotlivé jeho prvky jsou odděleny čárkou. Pokud není seznam prázdný, může být jeho první prvek pojmenován jako hlavička seznamu a zbytek seznamu jako jeho tělo. Seznamy jsou v Prologu velmi hojně využívanou strukturou.

Abych pomocí zmíněných termů mohla vytvářet programy, je nutné ještě definovat pojmy fakt, predikát, pravidlo a dotaz.

---

[3] Hornova klauzule – jedná se o klauzule s nejvýše jedním pozitivním literálem atomických predikátových formulí $p_1, p_2, \ldots, p_n, q$, tj. klauzule, které mají tvar $\neg p_1 \vee \neg p_2 \vee \ldots \vee \neg p_n \vee q$ (více v [7])

Poznatky, které pro nás mají význam vztahů mezi objekty či vztahů mezi vlastnostmi objektů vyjadřujeme pomocí predikátů. Predikát charakterizuje určitý vztah, který je buď pravdivý nebo nepravdivý. V následujícím příkladu jsou uvedeny příklady faktů.

Příklad 1:

```
jí(lea, ovoce).
jí(ivana, ovoce).
jí(michal, maso).
jídlo(ovoce,banan).
```

Pokud informace, na kterou se dotazujeme není v databázi uložena přímo, ale lze ji z uložených faktů odvodit, používáme takzvaná pravidla. Jedná se o další způsob zadávání predikátů a to pomocí podmíněných výrazů. Například, pokud chceme zjistit, zda Lea jí banány, můžeme to zjistit pomocí následujícího pravidla jí2.

Příklad 2:

```
jí2(Kdo,Co):- jí(Kdo,Druh_jídla), jídlo(Druh_jídla,Co).
```

Na příkladu můžeme vidět, že pravidlo se skládá ze dvou částí oddělených symbolem „:-“. Část, která se nachází vlevo od tohoto symbolu, se nazývá hlava pravidla a část vpravo nazýváme tělo pravidla. Jak je patrné z příkladu, pokud za sebe řadíme v pravidle více predikátů, oddělujeme je symbolem „ , “.

Posledním termínem, který je třeba si osvětlit, je dotaz. Otázky se pokládají v tzv. režimu dotazování. Dotaz může být dvojího typu. Prvním typem otázky je otázka, zda daný predikát je pravdivý, jinými slovy zda v databázi existuje takový fakt, nebo fakty a pravidla, z nichž lze tento závěr odvodit.

Příklad 3:

```
?-jí(lea,ovoce).
Yes
?-jí2(lea,jablko).
No
```

Na takovouto otázku Prolog odpovídá No, případně Yes, podle toho, zda je predikát pravdivý.

Druhým typem otázky je otázka zjišťovací. Při pokládání takovéto otázky chceme zjistit, zda existuje nějaký fakt, který otázce vyhovuje a zároveň chceme znát konkrétní hodnoty, pro které je predikát pravdivý. Pro takovéto otázky využíváme výše popsané proměnné.

Příklad 4:

```
?-jí(Kdo, ovoce).
```

V příkladě jsem položila dotaz, „Kdo jí ovoce?“. Prolog se nyní pokusí proměnnou Kdo (zapsanou s velkým počátečním písmenem), nahradit za konstanty tak, aby predikát byl pravdivý. V databázi faktů může nalézt hned dvě možná řešení – lea a ivana. Prolog ale prochází databází postupně a předkládá vždy jedno řešení. Pokud mi toto řešení stačí, stisknu klávesu Enter a Prolog již po dalším

řešení nepátrá. Pokud chci získat další řešení, stisknu „;“ a Prolog se pokusí nalézt další řešení. Pokud už žádné další řešení neexistuje vypíše No. Výsledek takového dotazu jsem zachytila v následujícím příkladu.

Příklad 5:

```
?-jí(Kdo, ovoce).
Kdo = lea ;
Kdo = ivana ;
No
```

2.1.2 SWI-Prolog

SWI-Prolog (dále uváděný spíše jen jako Prolog) je implementace jazyka Prolog distribuovaná pod licencí GPL. Tato implementace je poměrně rychlá a poskytuje řadu zajímavých možností. Je pro ni přichystáno také rozšíření pro propojení s jinými programovacími jazyky, které bylo využito pro vytvoření vlastní knihovny.

Bližší informace o SWI-Prologu i knihovnách použitých v této práci naleznete na [8].

2.2 SIMLIB/C++

Aby bylo možné vytvořenou knihovnu plnohodnotně a jednoduše použít pro modelování a simulaci, je nutné, aby byla schopna spolupracovat s knihovnami podporujícími simulaci. Já jsem zvolila knihovnu SIMLIB/C++.

SIMLIB/C++ je experimentální objektově orientovaná simulační knihovna, která se používá pro modelování a simulaci v programovacím jazyce C++. Objektová orientace v jazyce C++ přímo vybízí ke spolupráci mezi touto knihovnou a knihovnou, která je součástí tohoto projektu. Knihovna SIMLIB/C++ usnadňuje efektivní popis modelů přímo v jazyce C++, není tedy nutný překladač simulačního jazyka.

Knihovna SIMLIB/C++ začala vznikat roku 1990 na Ústavu informatiky a výpočetní techniky FEI VUT v Brně (nyní Ústav inteligentních systémů FIT VUT) a je distribuována pod licencí LGPL, která povoluje jakékoli modifikace kódu knihovny, pokud jsou zároveň zveřejněny výsledky ve zdrojovém tvaru pod stejnou licencí.

Knihovna SIMLIB/C++ je knihovnou pro popis diskrétních, spojitých i kombinovaných modelů, pro spojité modely dokonce disponuje 3D rozšířením. Dále obsahuje také nástroje pro sběr informací o chování modelu při simulaci a rozšíření pro optimalizaci parametrů simulačních experimentů. Dalším jejím rysem je možnost modelování systémů s fuzzy popisem.

Bližší informace o této knihovně naleznete na [5].

3 Knihovna pro propojení jazyka C++ a jazyka Prolog

Již v rámci semestrálního projektu jsem vytvořila prototyp knihovny, která byla později použita pro zde prezentované příklady. Tato knihovna dovoluje prostředky, jež jsou dostupné v Prologu, používat z prostředí jazyka C++. V první řadě se jedná o postupy a metody užívané v umělé inteligenci, které jsou mnohem snadněji realizovatelné v prostředí deklarativního jazyka, než v prostředí jazyka procedurálního. V této práci je to zejména procházení stavového prostoru za účelem nalezení nejkratší cesty.

Tato knihovna je vytvořena na úrovni prototypu, tedy stále existují funkčnosti, které nejsou z jazyka C/C++ dostupné, avšak rozsah její funkčnosti je již takový, že ji lze bez významnějších omezení používat pro účely modelování.

3.1 Návrh

Nejvyšším předkem je třída **PrologObject**, která zastřešuje celou strukturu. Má jediný atribut – *name*, který je využíván na straně faktů a predikátů pro definování jména faktu či pravidla na straně Prologu. V původním návrhu jsem počítala s využitím atributu *name* i na straně datových objektů, od této myšlenky jsem však upustila, protože pro tuto funkčnost nebylo nalezeno dostatečné uplatnění.

Od této třídy dědí nejvyšší třída objektů, se kterými se bude pracovat, **NullObject,** a třída **Register**, která zajišťuje znalost všech parametrů v každém faktu.

Třída **NullObject** definuje základní atribut pro datové objekty a to, zda daný objekt nese hodnotu či nikoli. K tomuto atributu přísluší metody pro nastavení a zrušení hodnoty objektu.

Od této třídy dědí třída **DataObject**, která reprezentuje všechny reálné datové objekty, se kterými se můžeme při použití knihovny setkat. Tato třída je abstraktní, neboť slouží k unifikaci všech datových typů – v mnoha případech je nutné s objekty všech typů zacházet jednotně, ať se jedná o kterýkoli typ. Již při tvorbě tohoto objektu, tedy přesněji jeho potomků, je nutné zadat, ke kterému faktu se bude daný objekt vztahovat. Vzhledem k určení knihovny pozbývá smyslu vytvářet datové objekty mimo fakty, případně predikáty. Výjimkou může být dále zmíněná tvorba seznamů.

První instancovatelnou třídou je **UnboundVariable**. Jedná se o volnou proměnnou, která je vrácena, pokud některý z hledaných parametrů faktu nebo predikátu není zjistitelný. Protože se nachází v hierarchii dědičnosti, musí mít všechny potřebné metody, i když některé nevyužije a zůstanou prázdné. Přetížila jsem operátor výstupu, aby bylo možné vypisovat obsah všech tříd v této větvi hierarchie.

Potomky této třídy jsou třídy typů, které nalézáme v Prologu. Pro každou třídu byl dvakrát přetížen operátor „=“. Poprvé přetížený operátor slouží pro vnitřní potřebu knihovny a jejím parametrem je proměnná typu t_term. Tento typ je poskytován knihovnou, kterou používám jako základ pro komunikaci s Prologem. Podruhé byl operátor „=“ přetížen, aby bylo možné nastavit hodnotu proměnné tohoto typu uživatelem, jejím parametrem je proto datový typ v C++, který odpovídá danému typu v Prologu. Všechny tyto třídy mají také metodu pro zjišťování hodnoty proměnné dané třídy – *getValue*.

Nově implementovanou třídou v rámci diplomové práce je třída **ListData**. Tato třída se zabývá zpracováním seznamů. Její implementace má na funkčnost knihovny významný vliv, neboť seznamy jsou jednou ze základních struktur používaných v jazyce Prolog.

Poměrně zásadní funkci má, již zmíněná, třída **Register**. Jejím atributem je kolekce objektů třídy **DataObject**, tedy jednotlivé parametry faktů. Objekt třídy **Register** vzniká spolu s objektem třídy **Fact** a udržuje informace o jeho atributech v ADT mapa. V případě volání faktu obdrží z této mapy potřebné atributy a poskytne je pro volání Prologu. Na druhé straně při získávání výsledků z Prologu je schopen výsledné hodnoty předat zpět do proměnných uložených v mapě. Jeho jedinou metodou je *registerData*, která při zadávání parametrů faktu o nich zaznamená všechny důležité informace.

Od třídy **Register** dědí abstraktní třída **Fact**. Od této třídy si uživatel knihovny zdědí svoji třídu, která bude reprezentovat konkrétní fakt. Tato třída poskytuje metody pro vkládání faktů do databáze Prologu, jejich volání i jejich odstraňování.

Ke komunikaci s Prologem je ale nutné nejprve ustavit spojení. Pro tento účel je vytvořena třída **PrologConnection**, která vytvoří spojení pomocí základové knihovny.

Základní struktura objektového návrhu je zobrazena v příloze k této práci.

3.2 Implementace vybraných tříd

V této kapitole se budeme zabývat implementací vybraných tříd, jejichž znalost by mohla být uživateli knihovny ku prospěchu.

3.2.1 Rozdělení do jednotlivých souborů

Každá třída má svůj vlastní soubor pojmenovaný jménem třídy. V něm se nachází implementace složitějších metod. Přehled o všech třídách této knihovny lze získat ze souboru PrologLib.h, který obsahuje definici všech tříd, některé jednodušší metody a všechny potřebné definice typů. Tento soubor je pak vložen do všech ostatních souborů.

3.2.2 Třída Fact

Protože od této třídy budou dědit všechny třídy uživatele reprezentující konkrétní fakty, je dobré ji probrat poměrně podrobně. Třída, jak již bylo řečeno, dědí od třídy **Register**. Sama o sobě neobsahuje žádné atributy. Jediný atribut, *name,* je zděděn od základního typu objektu **PrologObject**. Třída obsahuje několik důležitých metod:

- int *getArity()* - jedná se o metodu, která zjistí velikost mapy, ve které se shromažďují informace o atributech faktu. O této mapě budou uvedeny bližší informace při popisu třídy **Register**.
- *FactList * call(PrologConnection)* - metoda, která umožňuje volání faktů a predikátů v Prologu. Jejím parametrem je současné spojení do Prologu a jejím výstupem je seznam faktů, které odpovídají položenému dotazu. Nejprve je pomocí základové knihovny vytvořen fakt (predikát) se jménem faktu (příp. pravidla), na kterém je metoda volána, a z mapy, která tomuto faktu náleží, jsou do nového predikátu přeneseny známé hodnoty. Následuje otevření dotazu do Prologu a získání všech výsledků. Tyto jsou uloženy do seznamu faktů (jedná se o základní typ faktu vytvořený přímo ze třídy **Fact**). Poslední akcí je uzavření dotazu v Prologu.
- *bool insert(PrologConnection)* - jak již název napovídá, jedná se o metodu, která umožňuje za běhu programu vkládat fakty do Prologu. Vkládání predikátů, které mělo být implementováno během diplomové práce, bylo nakonec odloženo (při tvorbě příkladů se ukázalo, že by tato funkčnost nebyla v podstatě využívána).

  Vkládání probíhá pomocí volání prologovského predikátu **asserta/1**. Jeho parametrem je funktor (další z typů základové knihovny), který se skládá ze jména vkládaného predikátu a všech jeho parametrů. Parametry predikátu opět získáme z již zmiňované mapy, pomocí metody **assemblyTerm**, kterou obsahuje každá třída

reprezentující prologovský typ. Po zjištění všech potřebných údajů otevřeme dotaz do Prologu a po vložení faktu jej uzavřeme. Návratová hodnota pak značí, zda bylo vložení do Prologu úspěšné.

Volání pomocí **asserta/1** bylo zvoleno proto, že na začátku běhu programu se do databáze vloží všechny fakty a pravidla uložená v uživatelském souboru, který je přilinkován při překladu, a fakty přidávané za běhu aplikace by bylo dobré mít ještě před těmito predikáty. V případě kolize pak budou použity tyto dynamicky vložené fakty.

Pokud by to bylo posouzeno jako účelné, je možné přidat ještě metody zabezpečující vkládání na konec databáze pomocí predikátu **assertz/1** nebo vkládání pomocí predikátu **assert/1**.

- *bool retract()* - metoda určená k odstranění faktu, na kterém je zavolána, z databáze Prologu. Její volání probíhá podobně, jako u metody insert, jen místo na predikátu **asserta/1** je volána na predikátu **retract/1**.
- *bool retractall()* - metoda, která je určena k vyjmutí všech predikátů daného jména z databáze Prologu. Je volána na konkrétním faktu. Pokud fakt, na kterém je metoda zavolána, nemá nastavený žádný parametr, jsou odstraněny všechny fakty (resp. pravidla) tohoto jména, které se v databázi Prologu nacházejí. Pokud je některý z atributů nastaven, je odstraňovaná množina menší a odstraní se jen ty fakty (resp. pracvidla), které vyhovují včetně nastavených hodnot.
- *static void destroyList(FactList **list)* - metoda, která slouží k odstranění seznamu s výsledky volání metody *call*. Protože nejsem schopna posoudit, kdy bude možné automaticky odstranit výsledky volání, nechávám tuto akci na uživateli knihovny.
- *void printMap(int typ)* - metoda, která slouží k vypsání obsahu mapy náležející danému faktu, tedy všech jeho parametrů oddělených mezerami. Je určena spíše pro testovací účely.
- *void copy(Fact & aFact)* – metoda určená pro kopírování instance třídy **Fact** (příp. potomka této třídy) do jiné instance stejné třídy (příp. potomka této třídy). Je využívána zejména po získání výsledků z Prologu, kdy s výsledky potřebujeme dále pracovat. Výsledky jsou v seznamu *FactList* uloženy jako instance třídy **Fact,** která je pro další zpracování nevhodná, neboť není možné přistupovat k jednotlivým proměnným dané třídy. Výsledky proto mohou být metodou *copy* zkopírovány do instance třídy, kterou používáme v programu.

3.2.3 Třída Register

Stěžejní třída celé knihovny. Její instance je vytvářena ke každé instanci faktu a obsahuje všechny potřebné informace o tomto faktu. Třída obsahuje atribut typu DataMap, mapu, jejímž klíčem je bezznaménkové číslo (v praxi je to pořadí ukládaného parametru faktu), a hodnotou ukazatel na objekt třídy DataObject. Vybraným typem atributu je mapa, neboť je možné, že bude například definována jen hodnota třetího parametru a nebo požadována právě hodnota jednoho parametru.

Třída obsahuje jedinou metodu:

- *void registerData(DataObject*)* - tato metoda má velice jednoduchou implementaci. Jejím úkolem je pouze uložit daný objet do mapy na místo, které je určeno jeho pozicí v predikátu či faktu.

3.2.4 Třída IntData

Za všechny třídy reprezentující primitivní datové typy v Prologu jsem vybrala tuto. Implementace ostatních tříd se liší především v použití datových typů C++ a metod pro převod těchto typů do Prologu.

K těmto třídám existují dva druhy přístupu: první, kdy k objektu této třídy přistupuje uživatel a nastavuje mu nějakou hodnotu ručně, a druhý, kdy k objektu přistupuje knihovna a nastavuje jeho hodnotu podle výsledků dotazů. Původní plán byl nedovolit uživateli vůbec přístup k funkcím, které používá výlučně knihovna, ale implementace by byla natolik složitá (použití dvoustranných friend tříd atd.), že od ní bylo upuštěno. Protože se jedná o knihovnu pro úzce specializovanou skupinu uživatelů, předpokládá se, že ji budou používat dle daných pravidel.

Třída obsahuje jediný atribut, *value*, který slouží k uchování jeho hodnoty. Pro tento atribut existuje několik metod, které jsou převážně implementacemi metod definovaných výše v hierarchii tříd:

- *operator=(int)* - tato metoda slouží k uživatelskému nastavení atributu *value*. Je proto různá z hlediska typů pro každou ze zmiňovaných tříd.
- *operator=(term_t)* - metoda, která se řadí mezi ty, které jsou určeny jen pro vnitřní potřebu knihovny. Slouží k nastavení hodnoty z výsledku dotazu v Prologu. Typ term_t je typem základové knihovny a jeho převod na typ C++ je realizován funkcemi základové knihovny na úrovni jednotlivých tříd.
- *term_t assemblyTerm()* - již zmíněná metoda pro převod hodnoty datového objektu z typu C++ na typ term_t. Je realizována opět funkcemi základové knihovny. Používá se při konstrukci predikátu pro volání Prologu.
- *int getValue()* - uživatelská metoda pro získání hodnoty datového objektu. Návratový typ se liší podle typu třídy.

- *DataObject * clone(Fact &)* - metoda, která je používána při ukládání výsledků volání Prologu. Slouží ke klonování datových objektů. Při získávání výsledků z Prologu je nutné výsledky uložit do datového objektu správného typu. Aby se nemusel datový typ složitě zjišťovat, jednoduše se původní typ zkopíruje.

Třída samozřejmě používá také metody tříd, od kterých dědí. Například od třídy **DataObject** dědí metodu *unsetObject()*, která slouží k vymazání hodnoty objektu. Ve skutečnosti tato hodnota vymazána není, jen je v objektu nastavena hodnota **false** atributu *isSet*, což značí, že objekt v tuto chvíli nenese užitečnou hodnotu. Další takovou metodou je *getOrder()*, která vrací pořadí daného objektu v parametrech faktu.

3.2.5 Třída ListData

Ač se jedná rovněž o jeden ze základních typů, se kterými Prolog pracuje, implementace této třídy je díky struktuře tohoto typu odlišná. Tato třída se zabývá prací se seznamy. Obsahem seznamu může být libovolný jiný datový typ nebo jiný seznam. Tato třída vznikla až jako součást diplomové práce, neboť k jejímu dokončení bylo třeba, aby byly bez výhrad funkční jednotlivé datové typy.

Do seznamu se vkládají na úrovni C++ objekty jednotlivých tříd reprezentující datové typy v Prologu. V tuto chvíli lze seznam plnit jen popořadě, tedy v pořadí, jak mají být data uložena v seznamu. Jelikož je ale seznam v C++ reprezentován vektorem, je možné funkčnost rozšířit i na vkládání na zadanou pozici.

Třída samozřejmě obsahuje všechny výše zmíněné metody jako třída **IntData**, navíc však obsahuje metody pro vkládání jednotlivých prvků do seznamu – metody *addItem*. Parametry této metody se liší dle datového typu, který je do seznamu vkládán – číslo (integer nebo double), atom, znovu seznam nebo nic (aby mohl být vytvořen prázdný seznam). Aby bylo možné vkládat libovolnou strukturu, je nutné mít možnost si vytvořit instanci této třídy mimo fakt (při vkládání seznamu do seznamu). Jedná se o výjimku z pravidla uvedeného v návrhu, tedy že není příliš účelné vytvářet datové objekty mimo fakty, příp. predikáty určené ke komunikaci s Prologem.

V této třídě jsou také dvě další soukromé metody, *setList(DataVector& aValue)* a *clearList()*, které slouží k vložení hodnot do soukromého atributu a k vymazání seznamu. Tyto metody jsou ale určeny k práci se seznamem uvnitř knihovny a pro uživatele jsou tudíž nezajímavé.

3.3 Použití knihovny

Použití této knihovny nevyžaduje žádné odborné znalosti kromě znalosti objektového programování v jazyce C++ a znalosti programovacího jazyka Prolog.

3.3.1 Příprava souboru s pravidly a fakty

Aby při každém spuštění programu nebylo nutné znovu ručně vkládat všechny fakty a pravidla, přikládá se ke kompilaci zdrojových textů také soubor s pravidly a fakty Prologu, které se budou používat. V praxi to tedy může být tak, že pravidla a fakty, o kterých se ví, že budou jistě potřeba při výpočtu, se vloží do tohoto souboru a ve vlastním programu se vkládají jen specifické fakty, které se budou s každým výpočtem měnit.

V tomto souboru musí být zapsána všechna pravidla, která budou v průběhu výpočtu potřeba, neboť pravidla nelze vkládat dynamicky. Ve většině případů se však mění od výpočtu k výpočtu pouze data (tedy fakty), takže je možné si pravidla připravit předem.

3.3.2 Tvorba vlastní třídy

Každý fakt, na který se chceme v programu ptát, případně jej vkládat atd., má svou vlastní třídu, která dědí od třídy **Fact**. Na počátku je tedy nutné vytvořit si třídu, která bude mít stejný počet atributů, jako je parametrů v daném predikátu. Také typy parametrů se musí shodovat. Pokud tedy budeme volat pravidlo *soucet(X,Y,Z):- Z is X + Y*, pak potřebujeme třídu se třemi atributy typu IntData nebo FloatData podle toho, pro jaká čísla plánujeme predikát použít. Ačkoli v Prologu můžeme stejné pravidlo použít pro celá i reálná čísla, v prostředí C++ je třeba počítat s typovou kontrolou. Ukázka je záměrně zvolena pro pravidlo, nikoli pro fakt, protože se dají názorněji ukázat některé vlastnosti. Vkládání a rušení již budou ukázána pouze na faktech.

To, že pravidlo má mít jméno *soucet*, uplatníme při tvorbě objektu této třídy. Vraťme se ale k samotné třídě. V uvedeném pravidle si můžeme všimnout, že není libovolné umístění jednotlivých parametrů – konkrétně toto pravidlo bude mít pro Prolog řešení jen v případě, kdy zadáme první dva parametry. Proto musíme definovat, který parametr je který, i když zdánlivě jsou všechny stejného typu, takže by na pořadí nemuselo záležet.

V konstruktoru naší třídy zavoláme konstruktor třídy Fact, který zaručí vytvoření registru pro atributy nového faktu a poté konstruktor třídy **IntData**, resp. **FloatData**, ve kterém definujeme pořadové číslo konkrétního atributu a můžeme rovněž zadat jeho hodnotu. Na ukázku, jestliže parametry X, Y, Z z našeho příkladu ponecháme pojmenované stejně a chceme například sčítat čísla 3 a 2, pak může třída vypadat například takto:

Příklad 1:

```
class MyFact : public Fact
{
      public:
            MyFact(string aName): Fact (aName), X(*this, 0, 3),
            Y(*this, 1, 2), Z(*this,2) {}
            IntData X;
            IntData Y;
            IntData Z;
```

```
} ;
```

U prvních dvou parametrů jsme rovnou zadali jejich hodnotu a patrně plánujeme se na třetí parametr zeptat. Všimněme si, že u datových objektů zapisujeme, ke kterému z faktů náleží – to pro lehčí práci při jejich registraci.

Třída jako taková, může sloužit pro několik faktů se stejnými parametry, proto se jméno faktu zadává až při tvorbě instance. Kdybychom měli také pravidlo *minus(A,B,C):- C is A – B*, pak by se dala použít stejná třída, jen by se její instanci přiřadilo jiné jméno.

3.3.3 Získávání spojení s Prologem

Abychom mohli vytvořenou třídu využít, musíme si mezi naším programem a Prologem ustavit spojení. K tomu slouží třída **PrologConnection**. Vytvoříme tedy její instanci, v jejímž konstruktoru budou tři parametry. Prvním z nich je jméno ustavovaného spojení a další dva jsou parametry příkazové řádky.

3.3.4 Instance třídy a její volání

Pokud máme ustavené spojení s Prologem, můžeme začít s dotazy. K tomu potřebujeme instanci třídy, kterou jsme si vytvořili. Její jméno zvolíme dle volaného predikátu, v našem případě tedy dle pravidla *soucet*. Na této instanci již bez obav můžeme zavolat metodu *call*, neboť máme všechny potřebné parametry nastaveny. Výsledek volání si uložíme do seznamu faktů a postupně si jednotlivé výsledky můžeme vypsat pomocí iterátoru a metody *printMap*, což není příliš efektivní, ale může to být vhodné například při testování a ladění. V případě, že s výsledky chceme dále pracovat, použijeme metodu *copy(Fact & aFact),* která umožňuje vložit výsledky získané z Prologu do instance třídy, jež má atributy stejného typu jako výsledek volání (např. IntData, IntData, IntData).

Po zpracování výsledků, které Prolog poskytl, je vhodné uvolnit paměť zabranou seznamem výsledků. K tomuto účelu slouží statická metoda *destroyList (FactList**).*

Pokud místo pokládání dotazu chceme vytvořený fakt vložit do databáze Prologu, použijeme místo metody *call* metodu *insert*.

3.3.5 Překlad programu

Knihovna je dostupná ve zdrojových kódech (spolu s dávkou pro vytvoření knihovny) a přeložitelná v operačním systému Linux. K překladu je nutná instalace SWI-Prologu, dále základové knihovny *SWIProlog*, *SWI-stream* a je možné také používat simulační knihovnu SIMLIB/C++. Pro správné sestavení je nutné mít nastavenu cestu k SWI-Prologu.

Pro snadnější překlad je vhodné vytvořit dávky pro překlad. Nejprve je nutné přeložit všechny zdrojové soubory s optimalizacemi na druhé úrovni. K sestavení se pak použije linker plld, který umožňuje sestavit program včetně vstupního souboru, který obsahuje fakty a pravidla pro Prolog.

Dávka umožňuje také sestavení s knihovnou SIMLIB/C++, která slouží pro simulační výpočty, neboť se předpokládá využití knihovny v tomto směru. Dále se sestavuje také s matematickou knihovnou (-lm) a explicitně také se standardní knihovnou c++ (-lstdc++).

K příkladům uvedeným dále v této práci byly překladové dávky vytvořeny. Pro snadnější pochopení překladu doporučuji prozkoumat tyto překladové dávky.

3.3.6 Praktická ukázka

Pro první praktickou ukázku práce s knihovnou jsem vybrala téměř klasický příklad vztahů v rodině. Do vstupního souboru pro jazyk Prolog byla zařazena všechna obecná pravidla. Samotní příslušníci rodiny budou přidáváni až v rámci programu. Vstupní soubor má tuto podobu:

Příklad 2:

```
:-dynamic rodic/2,sourozenec/2,predek/2.

rodic(X,Y):-otec(X,Y).
rodic(X,Y):-matka(X,Y).
sourozenec(X,Y):-rodic(Z,X),rodic(Z,Y),X \== Y.
predek(X,Y):-rodic(X,Y).
predek(X,Y):-rodic(Z,Y),!,predek(X,Z).
```

Na prvním řádku tohoto souboru říkáme Prologu, že všechna pravidla a fakty, které se v tomto souboru nacházejí, mohou být měněny i v průběhu programu. Pokud bychom tento řádek neuvedli a pokusili se za běhu programu přidat například fakt `rodic(petr, ivo)`, Prolog by vypsal chybovou hlášku, neboť má tento predikát veden jako statický (nelze jej tedy za běhu měnit).

Pravidla v tomto souboru určují vztahy X je rodič Y, X je sourozenec Y a X je předek Y.

V souboru main.cc vytvoříme třídu, která bude schopna obsluhovat tato pravidla. Všechna mají právě dva parametry řetězcového typu, proto stačí jen jediná třída, která bude základem pro všechna pravidla.

Příklad 3:

```
class MyFact: public Fact
{
      public:
            MyFact(std::string aName): Fact(aName),
            osoba1(*this, 0),
            osoba2(*this,1){}
            AtomData osoba1;
            AtomData osoba2;
};
```

V metodě main ustavíme spojení s Prologem a poté můžeme začít se zadáváním dat do databáze Prologu. Vytvoříme si instanci výše popsané třídy se jménem *otec*. Do datových objektů

osoba1 a *osoba2* vložíme jména z rodokmenu, který vytváříme, a pomocí metody insert je vložíme do databáze.

Příklad 4:

```
int main(int argc,char *argv[])
{
      PrologConnection conn("database",argc,argv);

      MyFact otec("otec");

      otec.osoba1 = "adam";
      otec.osoba2 = "jan";

      otec.insert(&conn);
…
}
```

Stejným způsobem vytvoříme i fakty *matka(X,Y)*. Nyní se již můžeme dotazovat na rodinné vztahy definované ve vstupním souboru.

Příklad 5:

```
      std::cout<<"Otec a syn jsou: ";
      otec.osoba1.unsetObject();
      otec.osoba2.unsetObject();
      otec.osoba2="ivo";

      FactList * list;
      FactList::iterator it;

      list = otec.call(&conn);

      for(it = list->begin();it != list->end();++it)
      {
            (*it)->printMap();
      }
```

V tomto dotazu použijeme již vytvořenou instanci *otec*. Nejprve zrušíme hodnoty jejich atributů a poté nastavíme druhý atribut na hodnotu *ivo*. Připravíme si seznam, do kterého se budou zapisovat výsledky, a iterátor nad tímto seznamem. Ten nám poté bude sloužit k vypsání výsledků.

Voláním metody *call* se do připraveného seznamu vrátí všechny výsledky, které odpovídají dotazu určenému predikátem, nad kterým metodu *call* voláme. K vypsání výsledků se použije metoda *printMap*.

Můžeme se ale dotazovat i na predikáty, které jsou obsaženy ve vstupním souboru, případně k nim dodávat nové fakty.

Příklad 6:

```
      std::cout<<std::endl;
```

```
std::cout<<"Sourozenci Michala:"<<std::endl;
MyFact bratr("sourozenec");

bratr.osoba1="michal";
bratr.osoba2="jeronym";

bratr.insert(&conn);

bratr.osoba2.unsetObject();

Fact::destroyList(&list);

list = bratr.call(&conn);

for(it = list->begin();it != list->end();++it)
{
      bratr.copy(**it);
      bratr.osoba2.print();
      std::cout<<std::endl;
}
```

Byla vytvořena nová instance dříve zmiňované třídy se jménem *sourozenec*. Do databáze se potom vložil fakt, že Michal a Jeroným jsou bratři. Ve vytvořené instanci byl odstraněn obsah atributu osoba2 a hledají se všichni sourozenci Michala. Pro správnou práci s pamětí se ještě před novým voláním, a tedy i ukládáním do seznamu faktů, tento seznam vyprázdnil.

Informace, které nám vrátil Prolog, jsme tentokráte nevypisovali rovnou na obrazovku, ale nejprve jsme si je uložili do faktu, se kterým bychom mohli dále pracovat. Výsledky tohoto dotazu si opět zobrazíme pomocí metody *printMap,* tentokráte metodu ale voláme na faktu, do nějž jsme si hodnoty přenesli.

Posledním bodem této praktické ukázky je výsledek celého popsaného programu.

Příklad 7:

```
      Welcome to SWI-Prolog (Multi-threaded, Version 5.2.11)
      Copyright (c) 1990-2003 University of Amsterdam.
      SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free
software,
      and you are welcome to redistribute it under certain
conditions.
      Please visit http://www.swi-prolog.org for details.

      For help, use ?- help(Topic). or ?- apropos(Word).

      Otec a syn jsou: ivo jan

      Sourozenci Michala:
```

```
jeronym
lukas
ivo
```

Na začátku každého výpisu se zobrazí uvítací výpis SWI-Prologu, který se při startu programu spustil. Tento výpis lze potlačit, neboť je standardně vypisován na chybový výstup. Poté už následují výsledky našeho programu. To, že všechna jména jsou psána s malými písmeny, není chybou. Jedná se totiž o takzvané atomy. Dalo by se říct, že atomy jsou vlastně nulární fakty, jsou tedy vždy splněny. Pokud by jména byla napsána s velkými písmeny, považoval by je Prolog za proměnné, za které je nutné dosadit nějakou hodnotu. Jména s velkým písmenem se sice do Prologu vložit dají pomocí uvozovek, ale práce s tímto typem je poněkud složitější, neboť Prolog tyto hodnoty reprezentuje pomocí jejich ordinálních hodnot. Tento datový typ dosud v knihovně není podporován.

Zdrojový kód celého programu včetně popisovaného rodokmenu je uveden v příloze.

3.3.7 Rady pro práci s knihovnou

Prvním krokem při navrhování programu, který bude spolupracovat s touto knihovnou a bude tedy používat programovací jazyky C++ a Prolog, je správně posoudit, která část programu má být zpracovávána v jakém programovacím jazyce. Dobré rozhodnutí na této úrovni vývoje programu pak nesmírně ulehčí další práci.

Dalším krokem by mělo být rozhodnutí, jaká pravidla (příp. fakty) budou použitelná pro všechny výpočty, a která bude nutné měnit s každým výpočtem. Část, která se nebude měnit od výpočtu k výpočtu, potom přeneseme do souboru, který připojíme při překladu programu, kdežto část, která se měnit bude, budeme zpracovávat až za běhu programu.

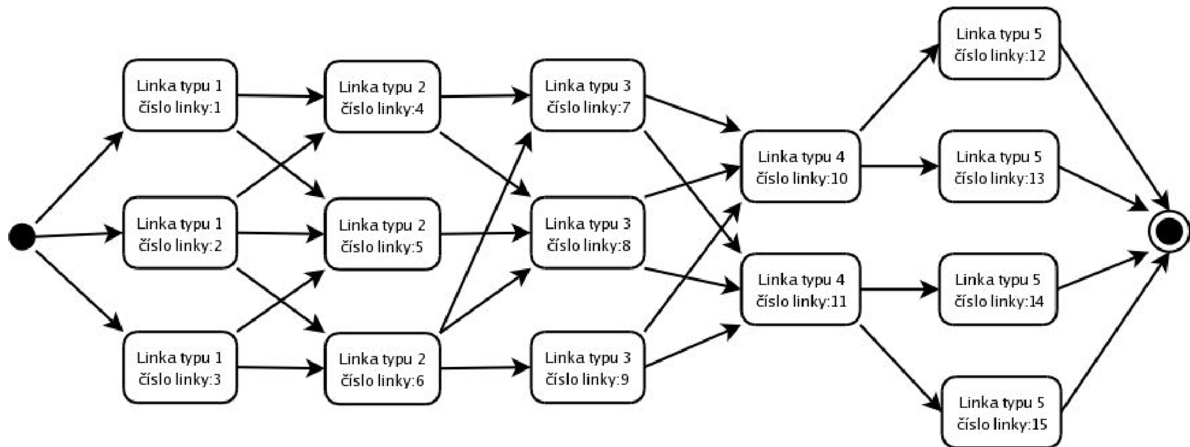
Pozor na statické fakty a pravidla z hlediska Prologu. Pokud se do vstupního souboru explicitně neuvede, že fakty a pravidla mají být dynamická (viz praktický příklad), pak již nebude možné fakty a pravidla uložená v souboru během výpočtu přidávat a měnit.

Při práci s Prologem je nutno mít na zřeteli, že jeho paměťový prostor není neomezený a je tedy nutné fakty, které již nebudeme potřebovat, z databáze odstranit, zvláště pokud jich používáme větší množství.

Jak již bylo řečeno, pokud používáme více faktů, příp. pravidel, které mají stejný počet parametrů stejného typu (např. atom, atom, int), není nutné vytvářet pro tyto objekty různé třídy. Lze využít jedné třídy, jejíž instance budou mít různá jména.

3.4 Netriviální příklad - průchod sítí zařízení

Prvním, ne zcela triviálním, příkladem je průchod sítí zařízení, která je zobrazena na následujícím obrázku. V tomto příkladě jsem použila složitější konstrukce a to jak na straně Prologu, tak na straně C++. Oproti předchozímu příkladu je navíc zapojena simulační knihovna SIMLIB/C++.



Obrázek 1: Síť zařízení

Síť představuje zařízení několika druhů (znázorněno stavy - obdélníky), která jsou vzájemně propojena přechody (orientovanými hranami znázorněnými šipkami). Vždy v jednom sloupci jsou znázorněna zařízení jednoho druhu. Úkolem výrobku, který je postaven do vstupního bodu systému (černé kolečko) je projít co nejrychleji touto sítí a dostat se do koncového bodu (kolečko s černým středem). Výrobek ale neputuje touto sítí sám, musí se proto rozhodnout, jakou cestu zvolí, aby se vyhnul jiným výrobkům, které putují sítí, a nemusel tak čekat ve frontě na některé zařízení.

3.4.1 Analýza

K tomuto testovacímu příkladu jsem přistupovala třemi různými přístupy:

1. Výrobek se nijak nerozmýšlí a zvolí první cestu, kterou „vymyslí“. Neboť k realizaci bylo použito algoritmu bez náhodnosti, vybere každý výrobek vždy stejnou cestu a síť v podstatě vůbec nevyužije.
2. Výrobek si naplánuje cestu sítí, jakmile je postaven do výchozího bodu. Tohoto plánu se pak drží po celou cestu sítí.
3. Výrobek se o své další cestě rozhoduje po průchodu každým zařízením. Na začátku své cesty si tedy vytvoří plán, který ale reviduje podle aktuální situace na zařízeních, kterými může na své cestě ještě projít.

3.4.2 Implementace

Tento příklad nevyžadoval náročnou implementaci v programovacím jazyce C++, proto byl celý zpracován v rámci jednoho souboru main.cc. Hlavní roli v něm hraje třída **Vyrobek**, která je potomkem třídy **Process** (jedná se o třídu, která je součástí simulační knihovny SIMLIB/C++ a která zachycuje objekty s dynamickým chováním). Objekty této třídy představují výrobky, které přicházejí do systému a mají projít sítí zařízení. Metoda *Behavior()* této třídy má za úkol zabírat zařízení, které jí doporučí metoda *rozhodni(const char * start_box)*.

Tato metoda má jako parametr linku, kterou výrobek uvolnil jako poslední. Od této linky až po výstupní bod systému se snažím nalézt nejvhodnější cestu sítí linek. Podle toho, zda je tato metoda zavolána jen na začátku života výrobku, nebo po každém uvolnění linky, rozlišuji 2.a 3. typ řešení, které jsou popsané v kapitole 3.4.1 Analýza.

Metoda *rozhodni* využívá pro komunikaci s programem napsaným v jazyce Prolog metodu *cesta(Resclass& result, const char * start_box)*. Jejími parametry jsou objekt třídy **Resclass**, který je využíván pro komunikaci s Prologem a jméno zařízení, linky, kterou výrobek opustil jako poslední. Název této linky je předán do Prologu, v němž je zachycena celá síť linek a pomocí algoritmu A* je nalezena nejvhodnější cesta k výstupnímu bodu systému.

Algoritmus A* dovoluje najít nejlevnější cestu k cíli. Každý přechod z jednoho místa do druhého má svou cenu. Použitý algoritmus dokáže najít cestu po takových přechodech, aby celková cena byla co nejnižší. Pro tento příklad jsem použila oceňování orientovaných hran prostřednictvím front na zařízení, které je na konci hrany (hrana má začátek v zařízení, kde se potenciálně nachází výrobek, a konec v zařízení, které je naším cílem, jedná se o nejmenší možný přesun). Ocenění hrany v daném okamžiku je provedeno tak, že se zjistí aktuální fronta u zařízení na konci hrany.

Aby bylo možné jednoduše zpracovat i začátek a konec sítě, jsou tyto body pro Prolog taktéž označeny jako linky – v případě vstupní linky se jedná o linku 0 a v případě výstupní linky se jedná o linku se jménem linka100.

3.4.2.1 Průběh života výrobku

Po vzniku výrobku (a spuštění metody *Behavior()*) je výrobku naplánována první cesta sítí. Jsou oceněny hrany mezi linkami (viz výše) a spolu s aktuální polohou výrobku (linka0) se odešlou do Prologu, kde jsou zpracovány a zpět do programu v C++ se vrátí seznam linek, kterými by měl výrobek projít. Pokud se jedná o druhý typ řešení, pak práce Prologu pro tento výrobek skončila a výrobek se tohoto plánu bude držet. Výrobek se tedy postaví do fronty k první lince ze seznamu a po jejím uvolnění jen vybere ze seznamu linku další, dokud se nedostane na konec sítě.

Pokud se jedná o řešení 3. typu, pak ze seznamu s linkami na nejlepší cestě k cíli je vybrána první a výrobek se zařadí do její fronty. V okamžiku, kdy ale tuto linku opustí, jsou znovu odeslány informace o ocenění hran a aktuálním umístění výrobku do Prologu, který opět nalezne nejlepší

možnou cestu pro daný okamžik a linky, které se na ní nachází vrátí v podobě seznamu. Tento cyklus se opakuje, dokud výrobek nedosáhne do koncového bodu sítě.

3.4.2.2 Metoda cesta(Resclass& result, const char * start_box)

Protože se jedná o metodu, která realizuje komunikaci s Prologem, budeme se zabývat její implementací hlouběji.

V této metodě se vyskytují dvě důležité třídy **Boxclass** a **Resclass**. První z nich slouží k reprezentaci faktu linka, který Prologu dává informace o frontách u jednotlivých linek.

Příklad 8:

```
class Boxclass: public Fact
{
        public:
        Boxclass(std::string aName): Fact(aName),
        jmeno(*this, 0), fronta(*this,1){}
        AtomData jmeno;
        IntData fronta;
};
```

Tato třída slouží k reprezentaci faktu, jenž má dva parametry typu AtomData a IntData. První z nich, *jmeno*, reprezentuje jméno linky a druhý, *fronta*, frontu, která je v daný okamžik před touto linkou.

Druhou třídou, která je popisované metodě předána jako parametr, je třída **Resclass** a její implementace je ukázána v následujícím příkladě.

Příklad 9:

```
class Resclass: public Fact
{
        public:
        Resclass(std::string aName): Fact(aName),
        start(*this, 0),
        end(*this,1),trvani(*this,2),cesta(*this,3){}
        AtomData start;
        AtomData end;
        IntData trvani;
        ListData cesta;
};
```

Tato třída reprezentuje fakt, který má 4 parametry – dva z nich jsou typu AtomData (jedná se o jména počáteční a koncové linky), další z nich je typu IntData a jedná se o cenu cesty a poslední z nich je typu ListData. Jedná se tedy o seznam, v tomto případě o seznam míst, která je potřeba projít na cestě od začátku do konce cesty.

Po seznámení se se základními třídami, jejichž objekty se objevují v této metodě, mohu přistoupit k rozboru samotné metody.

Jejími parametry, jak již bylo řečeno, jsou objekt třídy **Resclass**, který ponese výsledek dotazu a linka, ve které cesta sítí začíná. Linka, která je konečnou stanicí, je pro všechny výrobky stejná – linka100, proto není nutné ji předávat jako parametr.

Protože tato metoda je volána cyklicky a pokaždé nad jinými daty, je potřeba aby si nejprve „uklidila" databázi Prologu. K tomu slouží úvodní pasáž metody zobrazená v následujícím příkladě.

Příklad 10:

```
boxFact.fronta.unsetObject();
boxFact.jmeno.unsetObject();
boxFact.retractall(&conn);
```

Nejprve nastavíme oba atributy metody jako neobsahující hodnotu, to pro případ, že by jim byla na jiném místě hodnota už nastavena. Kdyby totiž byla metoda *retractall* zavolána na faktu, který má nastaven své parametry, nebyly by odstraněny všechny fakty tohoto jména, ale jen všechny fakty tohoto jména s těmito parametry.

Poté mohu všechny fakty tohoto jména odebrat z databáze pomocí zmíněné metody *retractall.*

V další části metody vložím do Prologu data o vstupní a výstupní lince. Tyto linky nejsou v C++ modelovány skutečnými linkami, v Prologu s nimi však takto počítám. Proto jim oběma nastavím hodnotu fronty na 0, viz Příklad 11:

Příklad 11:

```
boxFact.jmeno = "linka0";
boxFact.fronta = 0;
boxFact.insert(&conn);
boxFact.jmeno = "linka100";
boxFact.fronta = 0;
boxFact.insert(&conn);
```

Poté Prologu předložím data i o dalších skutečných linkách.

Příklad 12:

```
for(int i = 0;i<boxy.size(); ++i)
{
      boxFact.jmeno = boxy[i]->Name();
      boxFact.fronta = (int)(boxy[i]->QueueLen());
      boxFact.insert(&conn);
}
```

Prolog má v tuto chvíli dost informací o celé síti. Přechody mezi jednotlivými linkami jsou zapsány v souboru, který je sestaven s tímto programem a aktuální informaci o situaci na linkách jsem mu právě předala.

Nyní stačí Prologu říci, odkud kam má vyhledat cestu.

Příklad 13:

```
result.start = start_box;
result.end = "linka100";
```

```
FactList * list;
FactList::iterator it;


list = result.call(&conn);
it = list->begin();

result.copy(**it);
Fact::destroyList(&list);
```

Na začátku této pasáže je na objektu *result* třídy **Resclass**, nastaveno počáteční a koncové místo cesty. Poté je vytvořen list, do kterého se budou výsledky dodané Prologem ukládat, a na predikátu, který objekt *result* reprezentuje, je spuštěno volání Prologu.

Výsledky volání lze procházet pomocí iterátoru pojmenovaného *it*. Protože Prolog by měl vrátit jen nejvýhodnější cesty (které jsou stejně hodnoceny), můžeme použít libovolnou z nich. Vybereme tedy hned první výsledek uložený na začátku seznamu. Jeho hodnoty pomocí metody *copy* předáme do objektu *result*. Z tohoto objektu jsou již hodnoty dostupné jako datové typy C++ a můžou být dále zpracovány pro výběr linky, kterou má výrobek zabrat.

Aby byla uvolněna paměť alokovaná pro seznam faktů, které jsme získali z Prologu, uvolníme ji pomocí metody *destroyList(Factlist **list)*.

3.4.3 Výsledky testování

Testovány byly všechny tři přístupy k řešení. V případě, kdy síť nebyla vůbec využita (bez UI), byl minimální čas průchodu 50 časových jednotek - stejně jako při průchodu za pomocí algoritmu vyhledávajícího nejlepší cestu. Tento výborný čas je dosahován prvními výrobky, které ještě nemusí čekat ve frontách a tak není nutné použít algoritmus, který rozhoduje, do které z front se má výrobek zařadit. Stejně tak, pokud síť není naplněna a výrobky prochází bez čekání ve frontách, není žádný rozdíl mezi případem, kdy algoritmus byl a nebyl použit.

Jakmile se začnou před linkami tvořit fronty, začíná se projevovat algoritmus vyhledávající nejlepší cestu. Zatímco bez použití tohoto algoritmu sítí projde maximálně 995 výrobků, s jeho použitím to může být až dvakrát tolik.

Čas aktivace výrobků	Celkový čas simulace	bez UI				Bez aktualizace cesty			
		Min. čas	Max. čas	Průměr. čas	Počet zprac. výrobků	Min. čas	Max. čas	Průměr. čas	Počet zprac. výrobků
100	10000	50	59,81898	50,42528	**109,6**	50	59,28821	50,43924	**97,7**
90	10000	50	59,81141	50,57302	**120,5**	50	59,68102	50,61392	**113,2**
80	10000	50	64,26075	50,74615	**126,5**	50	60,37958	50,68493	**127,3**
70	10000	50	62,64345	50,54081	**147**	50	66,47648	50,75446	**140**

		bez UI				Bez aktualizace cesty			
60	10000	50	65,47893	50,94401	**162,3**	50	68,017	50,97207	**164,7**
40	10000	50	70,99536	51,68292	**235,7**	50	77,33574	51,59459	**247,7**
30	10000	50	73,13178	52,49862	**330,1**	50	81,3966	52,3657	**323,2**
20	10000	50	94,5055	54,69228	**495,9**	50	95,78757	54,66605	**495,6**
10	10000	50	425,367	198,8691	**969,5**	50	137,1497	63,76685	**1000,3**
5	10000	50	4991,987	2534,89	**995**	50	494,1777	237,8134	**1948,6**
1	10000	50	8984,452	4519,641	**995**	50	9172,575	3866,742	**1153**
0,9	10000	Neměřeno	Neměřeno	Neměřeno	Neměřeno	50	9122,947	3960,724	**1196**
0,85	10000	Neměřeno	Neměřeno	Neměřeno	Neměřeno	50	9028,4	4030,492	**1245,2**
0,8	10000	Neměřeno	Neměřeno	Neměřeno	Neměřeno	50	9136,838	4075,201	**1209,1**
0,75	10000	Neměřeno	Neměřeno	Neměřeno	Neměřeno	50	9104,323	4210,411	**1284,2**
0,65	10000	Neměřeno	Neměřeno	Neměřeno	Neměřeno	50	9291,113	4516,009	**1394,2**
0,6	10000	Neměřeno	Neměřeno	Neměřeno	Neměřeno	50	9385,276	5014,449	**1513,2**
0,55	10000	Neměřeno	Neměřeno	Neměřeno	Neměřeno	50	9403,152	4877,528	**1469,3**
0,5	10000	Neměřeno	Neměřeno	Neměřeno	Neměřeno	50	9512,344	5244,24	**1444,5**
0,45	10000	Neměřeno	Neměřeno	Neměřeno	Neměřeno	50	9568,354	5193,872	**1245,2**

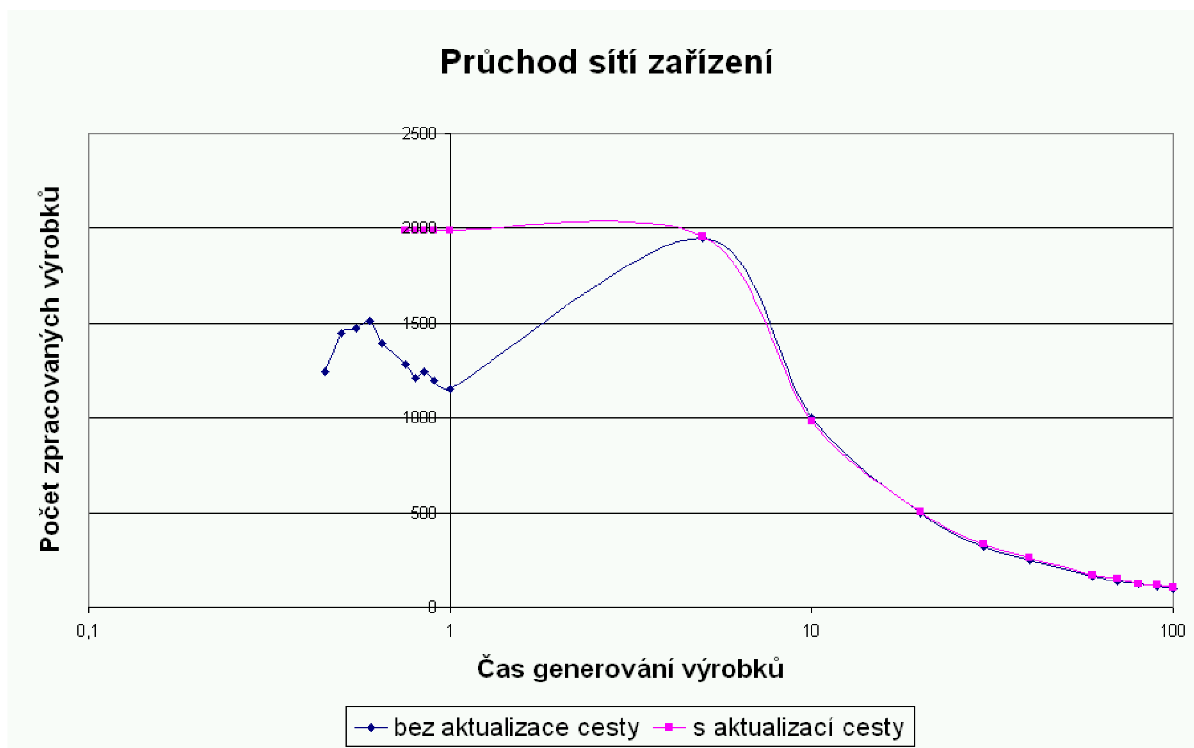
tabulka 1:Výsledky testů - průchod sítí zařízení I.

V následující tabulce jsou zaneseny výsledky testování druhého a třetího typu řešení – tedy pokusy, kdy je cesta vypočítána na začátku života výrobku a touto cestou se již definitivně řídí, a pokusy, kdy je cesta přepočítána kdykoli výrobek opustí nějakou linku.

Čas aktivace výrobků	Celkový čas simulace	S aktualizací cesty				Bez aktualizace cesty			
		Min. čas	Max. čas	Průměr. čas	Počet zprac. výrobků	Min. čas	Max. čas	Průměr. čas	Počet zprac. výrobků
100	10000	50	62,55448	50,57432	**104,6**	50	59,28821	50,43924	**97,7**
90	10000	50	67,87774	50,79926	**115,6**	50	59,68102	50,61392	**113,2**
80	10000	50	63,46114	50,51467	**122,5**	50	60,37958	50,68493	**127,3**
70	10000	50	67,95142	50,89244	**153,2**	50	66,47648	50,75446	**140**
60	10000	50	67,61948	50,98133	**171,4**	50	68,017	50,97207	**164,7**
40	10000	50	76,88378	51,86282	**260**	50	77,33574	51,59459	**247,7**
30	10000	50	75,30199	52,51873	**335,8**	50	81,3966	52,3657	**323,2**
20	10000	50	79,61257	54,03328	**502,1**	50	95,78757	54,66605	**495,6**
10	10000	50	91,74206	59,14305	**979,7**	50	137,1497	63,76685	**1000,3**
5	10000	50	339,5896	180,1795	**1954,1**	50	494,1777	237,8134	**1948,6**
1	10000	50	7997,401	4030,179	**1989**	50	9172,575	3866,742	**1153**
0,9	10000	50	8217,301	4137,454	**1989**	50	9122,947	3960,724	**1196**
0,85	10000	50	8292,492	4168,321	**1988,9**	50	9028,4	4030,492	**1245,2**
0,8	10000	50	8416,009	4237,503	**1989**	50	9136,838	4075,201	**1209,1**
0,75	10000	50	8514,098	4283,237	**1989**	50	9104,323	4210,411	**1284,2**

tabulka 2: Výsledky testů - průchod sítí zařízení II.

Jak je z tabulky patrné, nejvíce výrobků bylo zpracováno linkou s aktualizací cesty - 1989. Při zvedání četnosti příchodu výrobků jsem ale narazila na zajímavou věc. Hodnoty, zejména u neaktualizované cesty, poměrně kolísají. Tuto skutečnost si vysvětluji tím, že v době, kdy se cesta počítala (tj. na začátku života výrobku) mohla být některá z linek výrazně volnější než jiné a tak všechny výrobky, které vznikly za dané situace, mají ve své cestě tuto linku naplánovanou, čímž ji zahltí. Proto se tato vlastnost projevuje více u dlouhodobě přepočítané cesty, protože systém s aktualizací cesty dokáže tuto vlastnost částečně eliminovat. Kmitání lze dobře pozorovat na následujícím grafu.



Obrázek 2:Ukázka kmitání výkonu při průchodu sítí zařízení

3.4.4 Možná vylepšení

Hlavním vylepšením tohoto příkladu, které by vylepšilo bilanci prvního typu řešení, tedy bez umělé inteligence, by bylo zavést náhodu. Pokud by totiž výrobky náhodně volily linku, do jejíž fronty se zařadí (samozřejmě linku, do které vede přechod z linky, ve které se právě nachází), mohly by dosáhnout výrazně lepšího výsledku.

Dalším vylepšením, které by umožňovalo dynamicky měnit síť (například v případě, že by některá z linek byla nevyužita), je přesunutí informací o topologii sítě z části napsané v Prologu do části napsané v C++. V tuto chvíli jsou informace o topologii pevně stanoveny v programu pro Prolog a během simulace tedy nedovolují žádné změny. Pokud by byla topologie určována z jazyka C++, bylo by výhledově možné i testovat, zda přidání linky na určité místo zvýší průchodnost celé sítě, a to dynamicky za běhu programu.

Vylepšení, které by uvedený příklad mohlo posunout blíže k tzv. anticipativním systémům (více na [9]), je zavedení předvídání v oblasti velikosti front. Lze totiž zjistit, kolik a které výrobky se mohou řadit do stejných front jako výrobek, jehož cestu právě plánujeme. Pokud bychom dokázali odhadnout (a pomocí použitého algoritmu, který je napsán v Prologu i můžeme), pro kterou linku se ostatní výrobky rozhodnou, mohl by být průchod sítí ještě rychlejší.

4 Kruhový dopravník

Stěžejním příkladem použití knihovny je problém kruhového dopravníku. V tomto příkladu chci dokázat, že je výhodnější zapojit do otáčení dopravníku umělou inteligenci, která bude rozhodovat o jeho otočení, něž nechat dopravník otáčet o konstantní počet polí v určitém rytmu.

Kruhový dopravník je posuvný pás, na který v jednom místě klademe neopracované výrobky a na jiném místě výrobky, které již opracovány byly, odebíráme. K dopravníku jsou v určitých místech připojeny linky, které opracovávají výrobky. Tyto linky mohou být různých typů, můžeme je např. nazvat lakovnou, hoblovnou atd. K dopravníku může být připojeno více exemplářů jednoho typu linky.

Výrobek, který na dopravník položím, je „inteligentní“, tedy ví, kterými linkami se má nechat opracovat, má svůj výrobní plán. Pokud se výrobku nepodaří „nechat se opracovat“ všemi potřebnými linkami během jednoho průchodu dopravníkem, pokračuje dalším „kolečkem“.

Pokud výrobek prošel všemi potřebnými linkami, je přesunut na pole, které slouží k odebírání výrobků, a je z pásu odebrán.

4.1 Analýza

4.1.1 Použité termíny

Pro snazší pochopení dalšího textu, uvedu několik základních termínů, které jsou v tomto příkladě použity:

- **Dopravník** – posuvný pás, jehož začátek a konec jsou spojeny. Tato vlastnost umožňuje předmětům, které se po něm pohybují, cestovat po dopravníku více jak jedenkrát. Na každém místě na dopravníku může být umístěn maximálně jeden výrobek.
- **Výrobek** – předmět, který je posunován po dopravníku za účelem opracování. Výrobek má na sobě zaznamenán svůj výrobní plán. Výrobek je postaven základech tzv. „procesu“, jak je chápán v prostředí simulační knihovny SIMLIB/C++. Na ilustracích jsou výrobky zobrazovány trojúhelníky s pořadovým číslem.
- **Výrobní plán** – seznam typů linek, kterými musí být výrobek opracován. Na konci každého výrobního plánu musí být zmíněna linka pro výstup výrobků z linky, jinak by mohlo dojít k „zmizení“ výrobku z dopravníku.
- **Typ linky** – výrobek má být opracován určitým způsobem, např. lakováním. Pro zrychlení výroby může být na dopravníku umístěno více lakoven. V takovém případě, musí výrobek pouze vědět, že má být nalakován, nikoli, že má být např. nalakován

v lince 2. Proto je do výrobního plánu zanášen jen typ linky, nikoli konkrétní instance linky.

- **Instance linky** – konkrétní linka připojená k dopravníku. Linka má své přesné místo, kde je připojena k dopravníku. Pouze výrobky, které jsou na tomto místě, mohou do linky vstoupit. Linka (s výjimkou linky vstupní) nemá žádnou frontu. Pokud chce být výrobek opracován, musí ji zastihnout v okamžiku, kdy je volná, jinak pokračuje po dopravníku dál. Pokud chce linka výrobek, který již opracovala, vrátit na dopravník, musí vyčkat, dokud její místo na dopravníku není uvolněno (nemůže umístit výrobek na místo na dopravníku, kde již jiný výrobek je). Linka je postavena na základech tzv. „zařízení", jak je chápáno v prostředí simulační knihovny SIMLIB/C++. Na ilustracích jsou linky zobrazovány obdélníky.
- **Vstupní linka** – druh linky, která slouží k umisťování výrobků na dopravník. Jako jediná má vstupní frontu. Pokud chce linka uvolnit výrobek a umístit jej tak na dopravník, musí mít na daném místě dopravníku volno a také dopravník nesmí být již plně obsazen (viz 4.1.2 Princip dopravníku).
- **Výstupní linka** – linka, která slouží k odebírání výrobků z dopravníku. Po uvolnění této linky se výrobek již nevrací na dopravník a jeho život končí.

4.1.2 Princip dopravníku

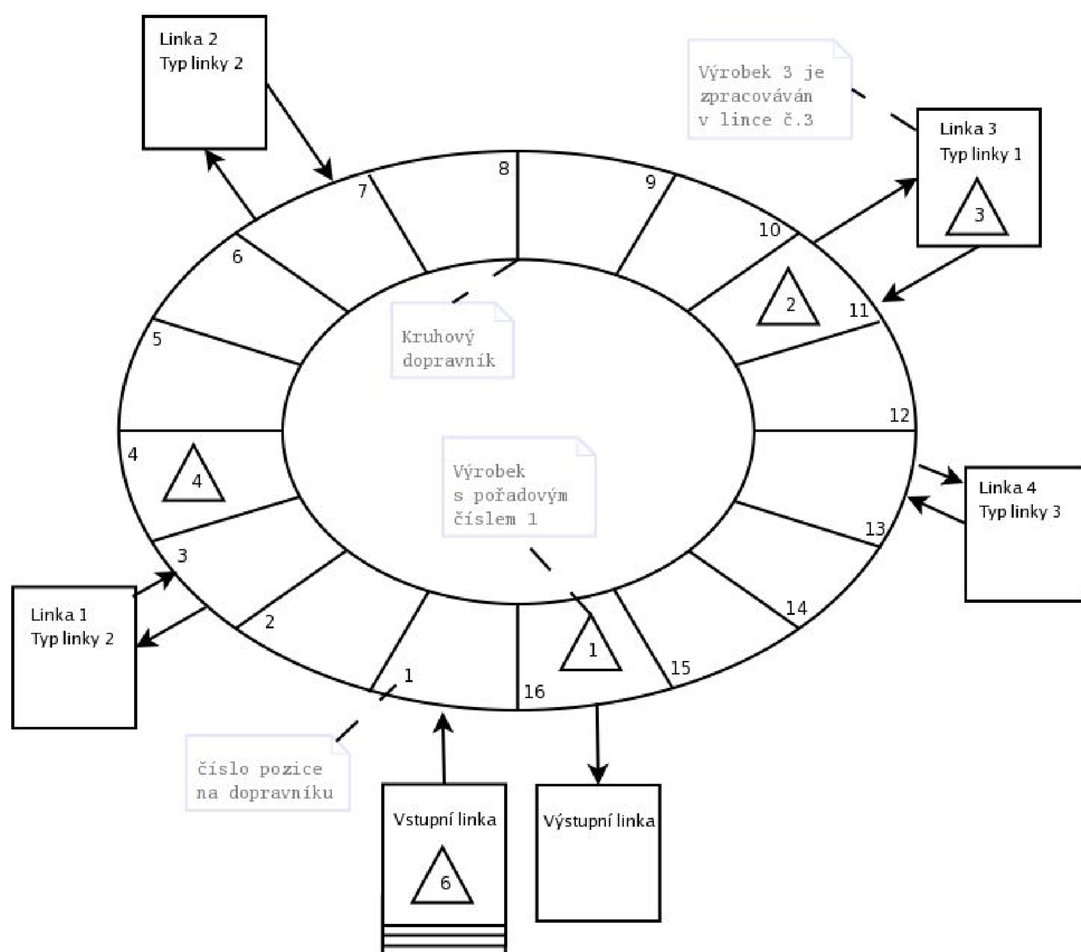
Princip dopravníku, jak je navržen v testovacím příkladě, bude vysvětlen na praktickém příkladu.

Na obrázku (Obrázek 3: Kruhový dopravník) je zobrazen kruhový dopravník, který má 16 pozic pro výrobky. Pozice 1 slouží pro umisťování nových výrobků a pozice 16 pro odebírání zpracovaných výrobků. Na pozicích 3, 7, 11 a 13 jsou umístěny běžné linky pro opracování výrobků.

Aby nedošlo k přeplnění dopravníku, je linka pro vstup výrobků koncipována jako linka s frontou. Do této fronty se řadí výrobky, které chtějí na dopravník vstoupit. Pokud je vstupní linka volná, může ji výrobek obsadit. Opustit ji ale může jen v případě, že je místo na dopravníku před linkou prázdné a dopravník není plně obsazen.

Plně obsazený dopravník chápu tak, že na něm je jen tolik prázdných míst, kolik je k němu připojeno linek. Tento předpoklad samozřejmě platí pouze pro případ, kdy počet linek je výrazně menší než počet polí v dopravníku a existuje více druhů linek. Kolik musí zůstat prázdných polí na dopravníku, aby byl ještě zajištěn jeho chod, závisí na konkrétním dopravníku.

Další specifickou linkou je linka pro odebírání výrobků z dopravníku. Pokud výrobek projde touto linkou, není již vracen na dopravník, končí jeho „život" a je ze systému odstraněn.



Obrázek 3: Kruhový dopravník

4.1.2.1 Posun dopravníku

Pokud má být otočeno dopravníkem, jsou požádány všechny výrobky a linky, aby informovaly o tom, jaké posuny jsou pro ně v příštím kroku vhodné. Z těchto informací se pak vybere takový posun, který vyhovuje nejvíce výrobkům a nejvíce linkám a dopravník je otočen.

Jako protiklad této možnosti dávám k dispozici i takovou implementaci dopravníku, kdy je dopravníkem otáčeno o konstantní počet polí (zpravidla 1).

Při návrhu dopravníku vyšlo najevo, že systém, který hodlám použít pro posun výrobků na dopravníku, tedy „demokratická volba výrobků a linek“, není vhodný pro všechny typy dopravníků. Například, pokud budu popisovat dopravník, který bude mít stále plnou kapacitu a bude se otáčet poměrně rychle, pak je výkon dopravníku, který se otáčí o jedno místo v každém cyklu srovnatelný s výkonem dynamicky se otáčejícího dopravníku (v dalším textu je též užito sousloví „inteligentní“ dopravník).

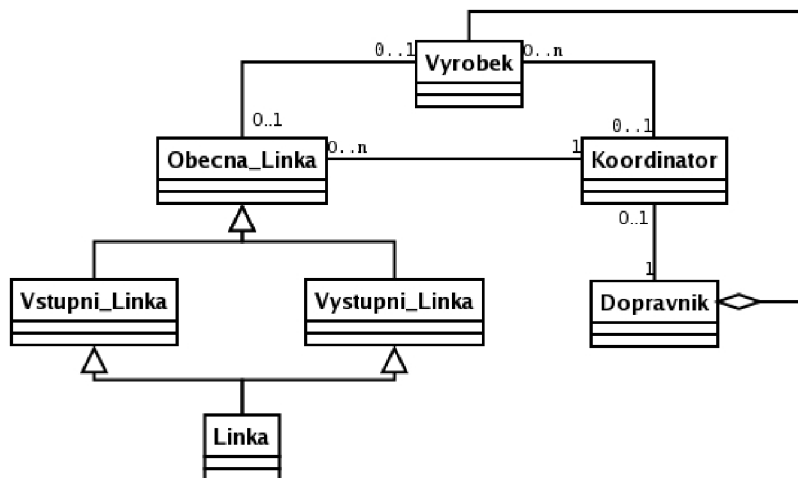
4.2 Implementace

4.2.1 Objektový návrh

Jedním z hlavních úkolů bylo správně rozdělit úkoly jednotlivým subsystémům, aby celkový výsledek byl co nejefektivnější a zároveň dobře implementovatelný. Část implementovaná v jazyce C++ obsahuje všechny objekty, které se týkají samotné simulace (dopravník, výrobky a linky) a část implementovaná v Prologu obsahuje řízení činnosti dopravníku (o kolik bude dopravníkem otočeno).

Část, která je zpracována v jazyce C++, využívá pro řízení simulace simulační knihovnu SIMLIB/C++.

Při vytváření návrhu bylo třeba myslet na synchronizaci všech složek, které se procesu účastní. Linky a výrobky musí v pravý čas dát vědět, jaké jsou jejich další záměry, a teprve po jejich zpracování se dopravník může otočit. Pro udržení celého systému v chodu jsem proto navrhla třídu **Koordinator**. Úkolem této třídy je synchronizovat práci všech ostatních tříd a komunikovat s Prologem.



Obrázek 4: Objektový návrh

Dále se v systému nachází třída **Dopravnik**, která reprezentuje samotný kruhový dopravník, třída **Vyrobek**, reprezentující výrobky na dopravníku a třídy, které představují linky pro zpracování výrobků.

Předkem všech linek v systému je třída **Obecna_Linka**. V rámci této třídy jsou definovány všechny potřebné atributy a většina metod. Od této třídy dědí třída **Vstupni_Linka** a třída **Vystupni_Linka**, které představují linky pro pokládání výrobků na dopravník a jejich odebírání z dopravníku. Ve třídě **Vstupni_Linka** byla přetížena metoda *Release(Entity *e),* aby bylo možno po ukončení zpracování vložit výrobek na dopravník, ve třídě **Vystupni_Linka** byla naopak přetížena metoda *Seize(Entity *e, ServicePriority_t sp)*, aby bylo možné výrobky na zpracování odebrat z dopravníku (když je výrobek zpracováván, nemůže být umístěn na dopravníku). Společným

potomkem těchto dvou tříd je třída **Linka**, která potřebuje výrobky z dopravníku jak odebírat (pomocí metody *Seize*), tak je na něj znova vracet (pomocí metody *Release*).

4.2.1.1 Vztahy mezi třídami

Protože třída **Koordinator** celou simulaci řídí, je nutné, aby měla všechny potřebné informace. Proto komunikuje se všemi ostatními třídami, aby získala informace o obsazení dopravníku a potřebách linek a výrobků. Tyto informace pak posílá do Prologu, kde jsou vyhodnoceny a zpět získává informaci o kolik se má dopravník posunout, aby nastala situace, která vyhovuje nejvíce výrobkům. Tuto informaci pak musí předat třídě **Dopravnik**.

Další vztah byl identifikován mezi výrobkem a linkou. Výrobek dostává informaci, která z instancí linky, kterou potřebuje být opracován, na něj vyšla a kterou má tedy obsadit.

Posledním vztahem je vztah dopravníku a výrobku, který se na něm pohybuje. Byl zvolen vztah kompozice, neboť výrobek, který není umístěn na dopravníku, nebo není právě zpracováván linkou, nemá v systému význam.

4.2.2 Třída Dopravnik

Třída **Dopravnik** slouží k reprezentaci kruhového dopravníku. Samotný dopravník je tvořen oboustrannou frontou (deque). Tento abstraktní datový typ byl zvolen z toho důvodu, že k jeho prvkům lze přistupovat jak od začátku, tak od konce (což se hodí při překládání výrobků z „konce“ dopravníku na „začátek“) a rovněž se dá efektivně indexovat. V této frontě jsou pak umístěny ukazatele na výrobky, které se na dopravníku nacházejí.

Většina metod této třídy slouží pouze ke zjišťování a nastavování výrobků na dopravníku. Implementačně zajímavou je pouze metoda *otoc(unsigned pocet).* V této metodě nejprve přesunu požadovaný počet prvků z konce fronty na její začátek, čímž provedu otočení dopravníku a poté zkoumám možnosti linek po otočení dopravníku, konkrétně, zda bude možné tuto linku v následujícím kroku uvolnit.

Pokud je zjištěno, že před linkou se nachází volné místo, je linka o tomto stavu informována. V případě, že linka zpracovává nějaký výrobek, předá mu linka zprávu, že pokud by ji chtěl v tomto okamžiku uvolnit, může tak učinit.

Pokud je ovšem zjištěno, že na dopravníku před linkou se nachází výrobek, pak je linka informována o tom, že není možné, aby ji výrobek, pokud nějaký zpracovává, opustil. Tato zpráva je poté linkou rovněž předána výrobku.

4.2.3 Třída Vyrobek

Třída **Vyrobek** je potomkem třídy **Process** knihovny SIMLIB/C++. Tato třída se stará výhradně o život jednotlivých výrobků. Při vzniku objektu je mu přiřazen tzv. výrobní plán, tedy seznam typů linek, které jej musí opracovat a jednoznačné ID.

Celý životní cyklus výrobku je zachycen v metodě *Behavior()*. Po vzniku výrobku se zařadí do fronty ke vstupní lince. Až se výrobku podaří tuto linku zabrat, čeká na příhodné podmínky, za kterých by mohl linku opustit (před linkou je volno a dopravník není obsazen). Po jejím opuštění se dostává do cyklu, ve kterém se snaží zabrat jednotlivé linky. Pokud má výrobek zabrat nebo uvolnit některou z linek, je o tom informován metodou *set_ready(Obecna_Linka * i)*. Pokud dojde ke změně a linka již nemůže být zabrána, výrobek o tom rovněž dostane zprávu (metoda *unset_ready()*). Po zpracování linkou je seznam linek, kterými má být ještě opracován, zkrácen (metoda *odeber_linku()*). Pro zjištění, který typ linky má nyní výrobek opracovat, je použita metoda *dalsi_linka()*. Pokud již v seznamu nezbývá žádná linka, je život výrobku ukončen a jeho údaje jsou zachyceny pro pozdější statistické vyhodnocení (mohou být prezentovány pomocí metody *statistika()*).

4.2.4 Třídy typu linka

Společným předkem všech linek, které jsou použity v tomto příkladu, je třída **Obecna_linka**, jež je potomkem třídy **Facility**. Na této třídě jsou definovány všechny atributy a metody, které jsou pro ostatní třídy společné, jako např. *pozice_na_dopravniku()*, která vrací umístění linky na dopravníku, nebo metoda *typ_linky()*, která vrací typ linky.

Další metodou, která je implementována na této třídě, je *volny_dopravnik(bool volno)*. Tato metoda slouží k informování výrobku, který je v danou chvíli linkou obsluhován, že před linkou je volné místo a pokud ji chce uvolnit, tak má v danou chvíli možnost.

4.2.4.1 Třída Vstupni_Linka

Třída **Vstupni_Linka** má překrytou metodu *Release(Entity * e)*. Ta navíc oproti metodě z třídy **Facility** umisťuje výrobky, které jsou uvolněny z linky, na dopravník.

4.2.4.2 Třída Vystupni_Linka

Třída **Vystupni_Linka** má překrytou metodu *Sieze(Entity * e, ServicePriority_t sp)*, která oproti metodě původní (taktéž z třídy **Facility**) umožňuje navíc odebrat výrobek z dopravníku před začátkem zpracování.

4.2.4.3 Třída Linka

Třída **Linka** dědí od tříd **Vstupni_Linka** a **Vystupni_Linka**. Běžná linka potřebuje jak výrobky z dopravníku odebírat, tak i na dopravník po zpracování vracet, proto dědí právě od těchto dvou tříd.

4.2.5 Třída Koordinator

Třída Koordinátor se stará o komunikaci s programem, který je napsán v Prologu, o synchronizaci a řízení celého systému. Stejně jako třída **Vyrobek** je potomkem třídy **Process**. Protože koordinátor by měl být v každém systému jen jeden, vytvořila jsem tuto třídu podle návrhového vzoru Singleton. Protože instance koordinátoru je potřebná i v jiných třídách (neboť koordinátor vidí na všechny části systému) vytvořila jsem rovněž metodu, která vrací ukazatel na koordinátor – *instance()*.

Aby nedošlo k nepříznivému stavu zpracování procesů (každý výrobek je procesem), má proces koordinátoru vyšší prioritu než procesy jednotlivých výrobků.

Důležitým atributem této třídy je statický atribut *HODINY*, který určuje, jak často se bude testovat stav systému a ověřovat, zda je vhodné otáčet dopravníkem.

Za pozornost stojí metoda *registruj(Obecna_Linka * linka)*, kterou jsem vytvořila dle návrhového vzoru Observer a při vzniku linky informuje koordinátor, že vznikla nová linka.

Zcela stěžejní funkci pro celý systém má metoda *Behavior()*. V rámci této metody se odesílají všechna potřebná data do Prologu, kde jsou zpracována, a výsledek je vrácen do C++ zpět této metodě. Poté jsou vyzvány výrobky, které stojí na místě, ke kterému je připojena linka, aby linku, pokud je volná a pokud ji zabrat chtějí, zabraly.

4.2.6 Program napsaný v jazyce Prolog

Program v jazyce Prolog má mnoho vstupních dat. Je potřeba zjistit aktuální pozici výrobků na dopravníku, jaký typ linky by který z výrobků chtěl zabrat, jestli jsou linky na dopravníku volné nebo právě opracovávají výrobek, kolik linek v systému je, na jakých pozicích se linky nachází, jak velký je dopravník a jaká linka je jakého typu.

Všechny tyto informace jsou v podobě faktů uloženy do databáze Prologu. Poté jsou postupně zpracovávány. Nejprve jsou zpracovány potřeby linek, tedy posuny dopravníku tak, aby linky, které by mohly vypustit výrobek zpět na dopravník, měly před sebou volné místo. Není hledán jen první (nejmenší) možný posun, ale všechny posuny, které vedou ke kýženému cíli. Poté jsou zpracovány výrobky, které aktuálně leží na dopravníku, nejsou tedy opracovávány žádnou z linek. Pro výrobky jsou opět hledány všechny vhodné posuny, tak, aby se ocitly před linkou, kterou chtějí zabrat a která je aktuálně prázdná. Seznam možných posunů pro každý výrobek či linku, tedy seznam typu [0,1,3], je poté převeden do podoby seznamu, který označuje četnost výskytu posunů (tedy při velikosti dopravníku 5 by převedený seznam vypadal takto [1,1,0,1,0,0]). Tyto seznamy jsou vhodné pro sčítání a nalezením maxima je pak lehce objeven ideální posun dopravníku. Pokud je maximum nalezeno pro nulový posun, zkusím, jestli jej nenaleznu i pro jinou hodnotu posunu. Zabráním tak při plnějším dopravníku nepříjemným situacím, kdy se dopravník nepohybuje, i když by to mnoha výrobkům vyhovovalo.

4.3 Identifikované problémy

Základním problémem při analýze systému a později i při implementaci byla verifikace systému. Vzhledem k tomu, že příklad nemá žádné grafické rozhraní a veškeré výstupy jsou jen ve formě textu, bylo velmi obtížné zjistit, zda dopravník pracuje dle zadání a např. neztrácí výrobky. Konkrétně k tomuto problému docházelo, pokud poslední linkou ve výrobním plánu nebyla linka výstupní. Výrobek tak prošel všemi linkami, kterými měl být opracován, pak byl ale ukončen jeho život bez ohledu na to, na jakém místě dopravníku se nacházel.

Testování těchto chyb, kdy celý systém zdánlivě fungoval, ale neplnil svoji funkci stoprocentně, bylo asi nejvíce časově náročné.

Dalším problémem, na který jsem narazila, byla synchronizace jednotlivých akcí. Pro systém je totiž nezbytně důležité, aby během jednoho tiku hodin proběhly všechny potřebné akce v patřičném pořadí. Tedy aby se nejprve linky uvolnily a teprve potom aby se znovu plnily (samozřejmě by pořadí akcí mohlo být i obráceno, ale je důležité, aby byly tyto akce odděleny).

4.4 Testování

Kruhový dopravník jsem testovala z několika pohledů. Prvním z nich bylo chování systému při zvětšujícím se dopravníku. Testovala jsem systém s dopravníkem od 10 do 50 pozic. Tento typ testu byl zařazen z toho důvodu, aby se ověřil předpoklad, že výrobky, které putují po dopravníku konstantní rychlostí, budou mít podstatně delší dobu života než výrobky, které mohou putovat o libovolný počet polí.

Druhým typem testů bylo chování a doba života výrobků na dopravnících s různým časem prodlevy mezi jednotlivými posuny. Zde jsem očekávala, že čím menší bude prodleva mezi jednotlivými posuny, tím menší bude rozdíl mezi oběma typy dopravníku. Tento předpoklad zdůvodňuji tím, že výrobek má při malých prodlevách mezi posuny dostatek času se i po jednotlivých krocích octnout na vzdálenějším místě za dobu zpracování jiného výrobku linkou. Nevýhodou, která se však projevuje ve všech testech dopravníku s konstantním otáčením, je fakt, že výrobek může projít kolem linky, kterou potřebuje obsadit dříve, než je tato linka volná a nemůže ji tudíž obsadit.

Dalším typem testu, který jsem se rozhodla provést na základě poznatků při provádění jiných testů, bylo prověření závislosti mezi rozestavěním linek a výkonností dopravníku. Během předchozích testů jsem narazila na zvláštní jevy. V některých případech se jevil jeden z dopravníků výrazně výkonnějším, skokově se však v jednom testovaném případě propadl a v dalším případě byl opět na dobré úrovni výkonu, jak je vidět v následující tabulce, kde dva vyznačené řádky označují zmiňovaný případ. Jedná se zde o dopravníky, jejichž odlišnost je jen v rozmístění linek. Jak je vidět, u dynamického dopravníku bylo dosaženo více než 70 % nárůstu.

V tabulce nejsou zachyceny jednotlivé pokusy, ale vždy průměr 10 provedených pokusů (stejně tomu tak je i u všech dalších prezentovaných tabulek). Jednotlivé pokusy je možné nalézt na přiloženém CD.

Po zjištění této skutečnosti jsem proto testovala různé kombinace rozestavění linek na obou typech dopravníku.

Velikost dopravníku	Čas otočení	Čas generování výrobků	Celkový čas simulace	Dopravník s dynamickým posunem				Dopravník s konstantním posunem			
				Min. čas	Max. čas	Průměr. čas	Počet zprac. výrobků	Min. čas	Max. čas	Průměr. čas	Počet zprac. výrobků
10	5	3	10000	95,082	8704	4411,8	**424,1**	182,4946	9197,707	4687,899	**256,4**
20	5	3	10000	147,5	5954,6	3069,5	**218,6**	370,2501	9368,003	4975,398	**200,7**
30	5	3	10000	158,43	6792	3559,6	**207,9**	584,449	9488,633	5213,51	**152,6**
40	5	3	10000	141,89	3926,8	2136,6	**94,9**	716,7068	9602,281	5627,778	**116,5**
50	5	3	10000	111,99	5295,6	2871,4	**115,7**	921,0914	9712,05	6055,541	**86,3**
40	5	3	10000	177,82	6930,5	3677,4	**167,5**	778,2788	9585,533	5504,227	**122,5**

tabulka 3: Závislost rozestavění linek na výkonu

Velkým problémem je fakt, že rozestavění linek, které je vhodné pro jednu prodlevu mezi otočením dopravníku, nemusí být vhodné pro jinou prodlevu mezi otočeními. Tento problém se projevil při provádění prvního typu testu, kdy jsem se pro co největší vypovídací hodnotu snažila dodržet stejné rozestavění linek na dopravníku pro různé doby otáčení dopravníku. Proto výsledky tohoto typu testu nemusí být, zejména pro krátké doby otáčení, naprosto v souladu se závěry, které byly z testování odvozeny. Při vyvozování závěrů jsem se snažila přihlédnout vždy ke všem aspektům, které k danému výsledku mohly vést.

Posledním typem pokusů byla různá doba výskytu výrobků v systému. Systém by měl být chráněn proti zahlcení (při určitém počtu výrobků na dopravníku systém nevpustí další výrobek na dopravník), takže by neměla nastat situace, kdy při velkém počtu výrobků na dopravníku bude výkon menší než při menším počtu.

Při všech pokusech jsem využila možností simulační knihovny SIMLIB/C++, která poskytuje funkčnosti pro sběr informací o průběhu simulace. Všechny provedené pokusy jsou zaznamenány v na přiloženém CD.

4.5 Výsledky

Testování tohoto příkladu přineslo řadu zajímavých poznatků. Ve většině tabulek, které jsou zde prezentovány, byl kvůli větší přehlednosti vynechán sloupec s počtem a rozestavěním linek. Linky byly zpravidla používány 4 a jejich konkrétní rozestavění je možné zjistit ze záznamu z pokusů, který je umístěn na přiloženém CD.

Testování na závislost výkonu a velikosti dopravníku byl poznamenáno již zmíněnou náchylností na rozmístění linek. Přesto však přinesl několik zajímavých zjištění. Při testování s prodlevou mezi otáčením 10 časových jednotek (což se rovná zpracování jednoho výrobku libovolnou linkou) držel „inteligentní" dopravník poměrně stabilní výkon ve všech měřených kritériích, zatímco výkon dopravníku s konstantním výkonem klesl až na nulu, jak je patrné z následující tabulky.

Velikost dopravníku	Čas otočení	Čas generování výrobků	Celkový čas simulace	Dopravník s dynamickým posunem				Dopravník s konstantním posunem			
				Min. čas	Max. čas	Průměr. čas	Počet zprac. výrobků	Min. čas	Max. čas	Průměr. čas	Počet zprac. výrobků
10	100	1000	10	112,76	128,9	117,23	**7**	261,0727	418,9523	332,3743	**8**
20	100	1000	10	116,31	135,56	124,74	**6,4**	261,5161	469,3224	372,9542	**6,7**
30	100	1000	10	117,19	137,37	127,07	**6,4**	369,669	683,224	474,298	**3**
40	100	1000	10	111,2	125,11	116,5	**8**	467,236	880	622,0135	**3**
50	100	1000	10	111,78	150,38	124,93	**8,9**	0	0	0	**0**

tabulka 4: Výkon systému na různě velikém dopravníku

Protože se mi konstantní výkon zdál poměrně zvláštní, zkusila jsem zvýšit počet výrobků, které se dostanou na linku. „Inteligentní" dopravník přestal mít konstantní výkon, i když byl výkonnější než dopravník s konstantním posunem. Konstantní výkon byl způsoben pravděpodobně tím, že více výrobků se na dopravník nedostalo a systém je i při zvětšujícím se dopravníku dokázal všechny zpracovat.

Vzhledem k tomu, že výrobků bylo průměrně jen 7, prodloužila jsem v dalších pokusech také čas simulace, aby rozdíly mezi jednotlivými pokusy nebyly pouze v jednotkách.

Další pokusy tedy byly prováděny v systému, který generoval výrobky exponenciálně každých 100, 50, 20, 10, 5, 3, nebo jednu jednotku, abychom dostali co nejucelenější obraz. Do textu jsem zařadila ukázky pro generování každé 3 časové jednotky. Bohužel, vzhledem k závislosti na rozestavění linek, nelze říct, o kolik je dopravník s dynamickým posunem výkonnější, protože by to znamenalo vyzkoušet všechny možné kombinace rozestavění linek, aby byly nalezeny ty nejvýhodnější pro oba dopravníky.

Velikost dopravníku	Čas otočení	Čas generování výrobků	Celkový čas simulace	Dopravník s dynamickým posunem				Dopravník s konstantním posunem			
				Min. čas	Max. čas	Průměr. čas	Počet zprac. výrobků	Min. čas	Max. čas	Průměr. čas	Počet zprac. výrobků
10	10	3	10000	163,57	9255,6	4724	**248,1**	337,9433	9536,471	4958,133	**140,8**
20	10	3	10000	240,06	7267,6	3808,3	**137,3**	602,0555	9597,12	5328,036	**111,9**
30	10	3	10000	224,96	7709,3	4128,7	**121,5**	810,358	9700,839	5668,083	**79,1**
40	10	3	10000	183,16	6919,1	3745,5	**92,9**	1027,553	9708,825	6140,902	**54,3**
50	10	3	10000	194,36	7393,5	4019,3	**82,7**	2268,092	9794,4	6761,886	**34,2**

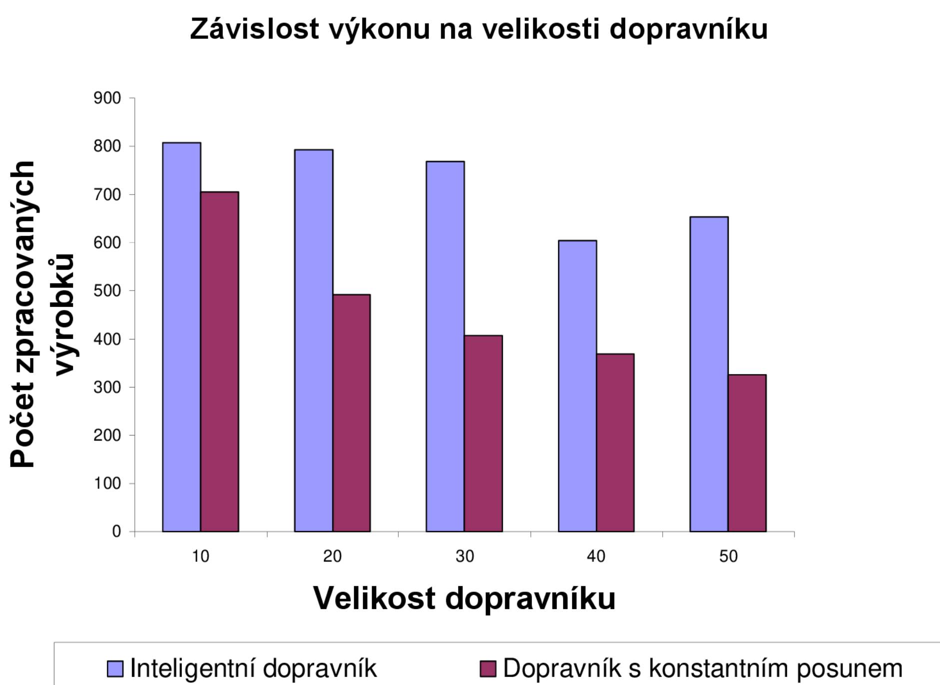
tabulka 5:Výkon systému na různě velikém dopravníku - pro otáčení po 10 časových jednotkách

Pokus jsem opakovala i pro časy prodlev 5 a 1 časová jednotka. I zde byl patrný vliv rozestavění linek. Výsledky pro prodlevu 5 jsou zobrazeny v již použité tabulce (tabulka 3: Závislost rozestavění linek na výkonu), výsledky pro prodlevu (tik hodin) 1 časovou jednotku jsou zobrazeny zde.

Velikost dopravníku	Čas otočení	Čas generování výrobků	Celkový čas simulace	Dopravník s dynamickým posunem				Dopravník s konstantním posunem			
				Min. čas	Max. čas	Průměr. čas	Počet zprac. výrobků	Min. čas	Max. čas	Průměr. čas	Počet zprac. výrobků
10	1	3	10000	46,52	7581,1	3820,4	**808,3**	66,79273	7833,845	3950,793	**706,4**
20	1	3	10000	57,896	6858,5	3444,4	**702**	92,75843	8451,987	4293,14	**493,5**
30	1	3	10000	57,059	7739,2	3906,8	**766,2**	113,3661	8872,594	4440,224	**409,2**
40	1	3	10000	49,194	7476,8	3785	**681,1**	132,6652	9056,042	4454,85	**348,5**
50	1	3	10000	48,385	6771,5	3412,4	**584**	170,4528	9418,74	4651,741	**310**

tabulka 6: Výkon systému na různě velikém dopravníku - pro otáčení po 1 časové jednotce

V následujícím grafu jsou pro ilustraci zobrazeny výsledky pro tik hodin každou jednu časovou jednotku, ale pro čas generování výrobků každých 5 časových jednotek.



Obrázek 5: Závislost výkonu na velikosti dopravníku

Výsledky druhého testu, tedy změny času prodlevy mezi jednotlivými posuny dopravníku, jsou rovněž zobrazeny v tabulkách tabulka 3, tabulka 5, tabulka 6 a tabulka 7. Předpoklad, že by se výkonnost dopravníku s konstantním posunem mohla přiblížit výkonnosti dynamického dopravníku, se potvrdil a rozdíl výkonnosti se snižujícím se časem kroku dopravníku opravdu zmenšuje. Přesto zůstává dynamický dopravník většinou výkonnější a to hlavně u větších dopravníků. V následující

tabulce je vidět výkonnost obou dopravníků při nejčastějším otáčení, které bylo testováno (0.1 časové jednotky).

Velikost dopravníku	Čas otočení	Čas generování výrobků	Celkový čas simulace	Dopravník s dynamickým posunem				Dopravník s konstantním posunem			
				Min. čas	Max. čas	Průměr. čas	Počet zprac. výrobků	Min. čas	Max. čas	Průměr. čas	Počet zprac. výrobků
10	0,1	3	10000	39,681	9012,4	4526,3	**982,2**	35,7553	9026,863	4544,623	**974,7**
20	0,1	3	10000	51,047	8889	4470,5	**965,2**	48,40436	9019,316	4572,274	**967,2**
30	0,1	3	10000	46,457	8341,8	4206,7	**894,7**	52,95534	9174,172	4581,805	**831,2**
40	0,1	3	10000	63,847	8444,3	4278,1	**886,7**	51,46305	9178,488	4600,476	**827**
50	0,1	3	10000	77,091	9064,8	4614	**947,5**	66,52583	9076,382	4694,135	**935,2**

tabulka 7: Výkon systému na různě velikém dopravníku - pro otáčení po 0,1 časové jednotky

Třetí test - prověření závislosti mezi rozestavěním linek a výkonností dopravníku – jsem prováděla na dopravníku o velikosti 10 pozic obsahujícím 4 linky, při čase výskytu prvku v systému každých 50 časových jednotek, době jednoho kroku 10 časových jednotek a celkové době simulace 10 000 časových jednotek. Prováděla jsem testy na různých pozicích z dopravníku: na pozicích 2, 4, 6, 8, pozicích 3, 5, 7, 9, (rovnoměrně rozložené po lince), pozicích 3, 4, 7, 8, (dvě skupiny linek) a pozicích 2, 3, 4, 5 (všechny linky na jednom místě). Pozice 1 a 10 jsou vyhrazeny pro vstupní a výstupní linku. Pro každou z těchto pozic existuje 24 možných kombinací[4]. Výsledky testů pro pozice 3, 5, 7, 9 jsou zobrazeny v následující tabulce včetně souhrnných informací (souhrnné informace se týkají celého provedeného testu).

Rozestavění linek	Dopravník s dynamickým posunem				Dopravník s konstantním posunem			
	Min. čas	Max. čas	Průměr. čas	Počet zprac. výrobků	Min. čas	Max. čas	Průměr. čas	Počet zprac. výrobků
3 5 7 9	110,3854	470,9008	190,682	183,1	264,8014	1945,275	997,19	160,8
3 5 9 7	110,6286	726,3856	250,9811	181,8	266,3304	2377,406	1141,58	153,5
3 7 5 9	111,8005	706,7207	235,0398	178,4	162,5947	1998,174	849,5235	160,6
3 7 9 5	111,4134	615,4157	224,8724	162,7	276,1222	2553,975	1288,361	150,7
3 9 5 7	113,4704	508,1976	231,9486	102,5	191,525	2438,145	1251,071	152,7
3 9 7 5	110,2878	462,5923	182,1923	182,7	187,6603	3197,955	1556,051	142,9
5 3 7 9	116,2634	517,4529	203,2816	140,7	272,0915	2270,259	1125,283	162,4
5 3 9 7	110,8615	609,0239	224,8284	189	266,7958	2494,262	1302,305	152,9
5 7 3 9	110,2772	596,9351	212,1115	181,3	168,1378	2546,588	1294,926	169,5
5 7 9 3	110,4156	565,9008	218,3876	179,5	350,665	3400,756	1718,826	143,1
5 9 3 7	110,4178	704,4179	220,6241	195,2	166,6934	3213,064	1542,314	142,1
5 9 7 3	110,4915	641,8502	244,8846	201,4	164,6792	2968,735	1450,162	153

[4] Permutace ze 4 prvků (2,4,6,8) se spočítá jako 4!, což je 24, kde permutace n prvků je skupina všech n prvků, které jsou uspořádány v jakémkoliv možném pořadí, tzn. výběr prvků závisí na pořadí.

Rozestavění linek	Dopravník s dynamickým posunem				Dopravník s konstantním posunem			
7 3 5 9	110,6185	547,1489	195,5907	197,2	165,9337	2657,526	1322,034	163,9
7 3 9 5	123,7085	449,0788	202,5295	126,6	277,4167	2569,88	1334,824	152,8
7 5 3 9	110,4085	543,4944	204,5143	167	165,419	2258,403	1165,486	161,4
7 5 9 3	110,2327	439,9106	189,4622	166,7	312,4193	2848,788	1586,818	140
7 9 3 5	111,1102	576,8408	228,6282	188,2	269,2501	3641,413	1714,926	136,5
7 9 5 3	111,0008	613,8866	190,9957	191,5	269,2422	2113,743	1023,49	156,6
9 3 5 7	110,9099	700,3318	227,055	189,9	163,1462	2417,298	1289,237	154,1
9 3 7 5	111,5697	567,5005	211,9538	164,1	188,9644	2935,525	1371,121	153,4
9 5 3 7	110,2437	624,9691	203,9519	185,3	172,5124	3161,383	1524,7	141,1
9 5 7 3	110,3297	516,2047	202,636	195,2	191,3815	3171,954	1537,303	158,5
9 7 3 5	110,8267	429,5708	197,8626	144,8	267,5854	3741,683	1944,371	136,4
9 7 5 3	110,2579	518,606	211,2339	193,5	266,7259	3194,308	1669,944	159,5
Min	110,1349	415,4668	177,4707	**102,5**	160,279	1541,662	825,326	**136,4**
Max	123,7085	726,3856	255,9492	**212,8**	350,665	3767,385	1944,371	**170,3**
Průměr	111,0745	560,1332	209,7728	**177,2773**	227,4112	2717,52	1384,232	**152,4365**
Rozdíl	13,5736	310,9188	78,4785	**110,3**	190,386	2225,723	1119,045	**33,9**

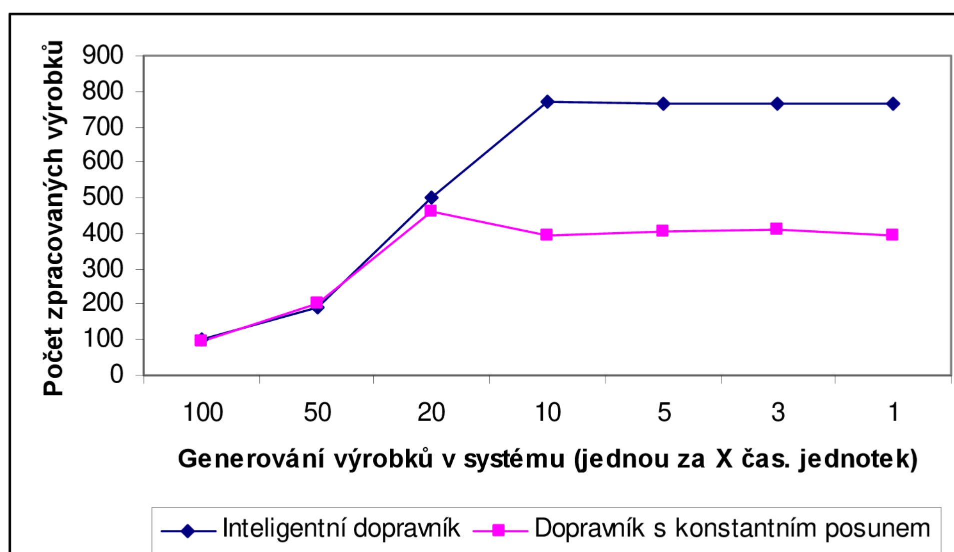
tabulka 8: Vliv rozestavění linek na výkonnost dopravníku

V prvním sloupci vidíme rozestavění linek, kde na první a druhé pozici jsou umístěny linky typu 2, na třetí pozici linka typu 1 a na poslední pozici linka typu 3. Dvě linky typu 2 byly zvoleny proto, že se ve výrobních plánech, tak jak byly vytvořeny, vyskytují nejčastěji.

Další sloupce již obsahují hodnoty zjištěné při pokusech. Z tabulky je patrné, že při použití dopravníku s konstantním posunem je rozptyl zpracovaných výrobků výrazně menší než u „inteligentního“ dopravníku. Na druhou stranu je nutno zdůraznit, že u dopravníku s konstantním posunem je podstatně delší průměrná délka života výrobku.

Pouze v několika málo z testovaných příkladů ale rozestavění linek vyhovovalo oběma druhům dopravníku zároveň – např.4, 6, 2, 8 nebo 5, 7, 3, 9 (zvýrazněno v tabulce).

Posledním typem pokusů byla různá doba výskytu výrobků v systému. Testování probíhalo při otáčení jednou za jednu časovou jednotku na dopravníku s 30 pozicemi a simulace trvala 10 000 časových jednotek. Výsledky tohoto testu jsem zobrazila v následujícím grafu.



Obrázek 6: Závislost výkonu dopravníku na výskytu výrobků v systému

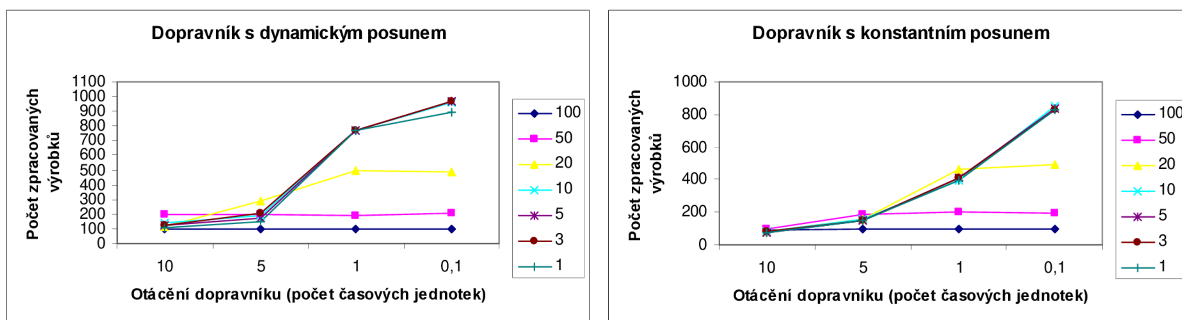
Jak je vidět, pro méně časté výskyty výrobků v systému jsou oba přístupy stejně výkonné. Jakmile ale přibude v systému více výrobků, začíná být dopravník s dynamickým posunem výkonnější. Protože systém má ochranu proti přeplnění, není možné zvyšovat jeho výkon do nekonečna, ale při naplnění kapacity dopravníku již výkon dále nestoupá.

4.6 Diskuze výsledků

Jak již bylo zmíněno v předchozích kapitolách, celé testování bylo poznamenáno vlivem rozestavění linek na výkon celého systému. Přesto jsem se pokusila provést dostatek pokusů, aby výsledky byly co nejvíce vypovídající o obecném výkonu systému s dopravníkem s dynamickým posunem.

Hlavním kritériem byl počet výrobků, které dokáže linka zpracovat. Dalšími důležitými kritérii, která jsou rovněž udávána u všech provedených testů, byla nejkratší a nejdelší doba, po kterou musel výrobek pobývat v systému. Při sledování těchto dodatečných informací je možné zjistit, že průchod výrobku systémem je závislý nikoli na velikosti dopravníku, ale na prodlevě mezi otáčením. Při pohledu na jednotlivé experimenty (pokud pomineme jednotlivé pokusy, které selhaly kvůli špatnému rozložení linek) mohu konstatovat, že u dynamického dopravníku zůstávají časy pro různé velikosti dopravníku poměrně stabilní i při zvětšování dopravníku. U dopravníku s konstantním otáčením je vidět podstatně větší diferenci. Další závěry se vztahují již k hlavnímu kritériu, tedy k počtu výrobků, které prošly systémem a byly opracovány.

Bylo prokázáno, že pokud se dopravník otáčí jednou za dlouhý čas, pak je výkon dopravníku s dynamickým otočením srovnatelný s dopravníkem s konstantním otočením. Při častějším otáčení je lehce výkonnější dopravník „inteligentní“ (např. při otáčení každou jednu časovou jednotku), viz následující grafy: (každá řada představuje pokusy s jiným množstvím generovaných výrobků)



Obrázek 7:Srovnání dopravníků s přihlédnutím k času otáčení

Dále bylo zjištěno, že rozestavění linek má menší vliv na dopravník s konstantním posunem než na dopravník s dynamickým posunem. Na druhou stranu, dopravník s dynamickým posunem měl celkově vyváženější výsledky u doby setrvání v systému.

4.7 Možná vylepšení

Už v průběhu testování tohoto příkladu jsem nalezla spoustu drobností, jejichž vylepšení, případně odstranění, by mohlo výkonnosti dopravníku ještě pomoci, případně by přispěly k transparentnosti celého programu.

Současná implementace trpí zcela zjevným nedostatkem v tom případě, že existuje typ výrobku, který má zcela odlišné potřeby než všechny ostatní výrobky. Pokud se totiž ve svých potřebách otočení dopravníku neshodne s dalšími výrobky, nemá šanci se k lince, kterou potřebuje zabrat, dopravit. Tento nedostatek by se dal ošetřit přidáním priorit. Každý výrobek, který určitou dobu „krouží" po dopravníku, by získal vyšší priority než výrobek, který je na dopravníku krátce. Tato priorita by se pak zohlednila při vytváření jeho seznamu výskytů posunů - např. násobením jeho prioritou. Tato úprava by však znamenala výrazný zásah do kódu programu, proto nebyla realizována.

Dalším možným vylepšením by mohl být třetí stav linky. V tuto chvíli jsem schopna rozlišit pouze, zda je linka obsazena, nebo zda je linka volná. Ke všem obsazeným linkám pak přistupuji jako k potenciálním zájemcům o umístění výrobku na dopravník. Kdybych ale měla možnost rozlišit mezi linkou, která zatím jen výrobek zpracovává a linkou, která se již chystá výrobek položit na dopravník, odfiltroval by se další zbytečný posun.

Další, spíše realizační, vylepšení by mohlo spočívat v upravení života výrobku a odstranění výstupní linky z výrobního plánu a jeho zapracování přímo do metody *Behavior()* výrobku.

Vzhledem k širokému záběru tohoto výzkumu by si taktéž zasloužil i mnohem mohutnější a komplexnější testování, než je možné provést v rámci diplomové práce.

5 Metodika pro další projekty

Velmi důležitým krokem, bez kterého by celý systém mohl výrazně ztratit na kvalitě, je rozdělení jednotlivých funkčností mezi subsystémy modelu. Který problém je lépe řešit v C++ a který v Prologu? Při své práci jsem došla k závěru, že není výhodné Prologu nechávat sebemenší prostor k řízení simulace a je výhodné jej volat jen pro konkrétní průchody stavovým prostorem, příp. jiné výpočty. Výsledky těchto výpočtů samozřejmě mohou sloužit k rozhodnutí, jaký bude další krok simulace, ale toto rozhodnutí již ponecháme na programu napsaném v C++.

Jakmile se totiž pokusíme přenechat část řízení na Prologu, začne být průběh simulace nečitelný a obtížně odladitelný. Prolog totiž při svém běhu používá několik vláken a to, co se v nich děje, je velmi těžko zjistitelné (já jsem k ladění používala nástroj gdb). Naproti tomu řízení simulace v C++ lze velmi snadno odladit a pokud volání Prologu vhodně zapouzdříme do metod, je velmi pravděpodobné, že poměrně brzy odhalíme, zda se chyba nachází v Prologu nebo v C++.

Pokud již máme jasné, jakou funkčnost budeme realizovat ve kterém programovacím jazyce, doporučuji začít implementovat výkonnou část v Prologu. Při psaní algoritmu odhalíme, jaké jsou vstupní parametry, co všechno budeme potřebovat Prologu dodat. Ač jsme provedli dobrou analýzu, může se stát, že pro výpočet v prologu budeme potřebovat nějaký další parametr a pokud si nejprve navrhneme strukturu objektů v C++, může být obtížné tento nový parametr zařadit do již existující struktury. Pokud ale začneme od programu napsaného v Prologu, zjistíme, které parametry je nutné dodat pro výpočet a jak se nám nejvýhodněji bude výsledek vracet.

Poté může přistoupit k návrhu programu v C++. Je nutné pamatovat na to, které informace budeme dodávat do Prologu, a podle toho vhodně volit datové typy. Např. je dobré počítat s tím, že čísla, která posíláme do Prologu, jsou datového typu integer. Protože knihovna obsahuje i metody určené pro vnitřní použití, které jsou typu term_t , neprobíhá konverze mezi typy unsigned a integer, ale mezi unsigned a term_t, což způsobí chybu programu. Z tohoto důvodu je nutné používat správné datové typy a nespoléhat se na implicitní konverze.

Jak již bylo zmíněno v kapitole pojednávající o vytvořené knihovně, pokud v programu budou použity fakty nebo pravidla se stejnými typy parametrů, je možné vytvořit jednu třídu, která bude sloužit pro všechny takovéto fakty, resp. pravidla. Je ale dobré, už jen kvůli čitelnosti programu, vytvářet nové objekty této třídy pro každé použití. Výjimkou snad mohou být případy, kdy objekt použijeme pro volání Prologu, kdy chceme znát výsledek dotazu a tento výsledek si pak uložíme do tohoto objektu. V jiných případech ale můžeme zapomenout odstranit staré hodnoty z původního objektu a Prolog pak nemusí vrátit korektní výsledky tak, jak jsme je očekávali, neboť měl nekorektní již vstup (zůstala tam data z předchozího použití).

Abychom se co nejvíce vyvarovali takovýchto chyb, je dobré si vytvořit ladící výstupy, které ukáží, co bylo ve skutečnosti do Prologu vloženo. Tyto výstupy nesmí odrážet informace, které máme

k dispozici a vkládáme je do Prologu, ale informace, které získáme přímo z Prologu, po vložení. Na následujícím příkladu vidíme praktickou ukázku kontroly správného vložení. Jedná se fragment z kontroly v kruhovém dopravníku, kdy zjišťujeme, jaké jsme vložili fakty o výrobcích na dopravníku a jejich potřebách linek. Jelikož vstupujeme doprostřed kontrol, jsou již veškeré objekty připraveny k použití.

Příklad 14:

```
std::cout<<"vyrobek"<<std::endl;
fakt_vyrobek.typ_linky.unsetObject();
fakt_vyrobek.vyrobek.unsetObject();

list =  fakt_vyrobek.call(&conn);
for(iter = list->begin();iter!= list->end();++iter)
{
      fakt_vyrobek.copy(**iter);
      fakt_vyrobek.printMap();
      std::cout<<std::endl;
}
Fact::destroyList(&list);
```

Všimněme si, že jsem použila již jednou použitý objekt a vymazala jsem jeho proměnné, abych zaručila jeho prázdnost. Je to další možnost, jak pracovat s objekty a zajistit jejich korektní vstupy (jak je zmíněno výše), ale vyžaduje koncentraci, abychom nikdy nezapomněli tyto „mazací" metody použít.

Dále na tomto prázdném objektu zavoláme volání Prologu, který vrátí všechny výskyty tohoto faktu v databázi do seznamu *list*. Tento je poté vytištěn a nakonec zrušen, aby nezabíral místo v paměti.

Kontrola uvolňování paměti je rovněž důležitým hlediskem. Většinu paměti alokované pro komunikaci s Prologem dokáže knihovna sama uvolnit. Existují však situace, kdy je uvolnění na uživateli – například při volání Prologu je nutné po ukončení práce se seznamem výsledků tento seznam zrušit. Může se to zdát naprosto malicherné, avšak pokud počítáme se statisíci procesy, což se při simulaci může stát, může být každý neuvolněný byte důležitý.

Závěr

Cílem této práce bylo ukázat některé možnosti nabízené heterogenním modelováním. Byla vytvořena knihovna, jež dovoluje propojit dva modelovací přístupy na objektové úrovni. S využitím této knihovny, napojené na simulační knihovnu SIMLIB/C++, byly vytvořeny dvě případové studie (case-study), pomocí nichž jsem se snažila demonstrovat výhody, které může heterogenní modelování přinést. Současně s tím bylo i ukázáno, jak by se modelovaným inteligentním řízením mohla zvýšit produktivita práce v modelovaných situacích.

Navržená koncepce modelování vychází z dřívějších publikovaných úvah (např. [3]). Tyto teoretické úvahy se mi podařilo prakticky implementovat a umožnit jejich ověření v simulační praxi. Jedná se o jeden z prvních úspěšných pokusů o uvedení těchto myšlenek do simulační praxe. Podle mého soudu dosahuje navržená knihovna vysokého stupně abstrakce, který bychom mohli požadovat od moderních metod modelování. Současně s knihovnou byla předložena i jistá modelovací metodologie, která byla popsána a užita při popisu simulačních studií – v obou případech se jedná o výrobní procesy, tedy o problémy s přímou návazností na realitu. V těchto studiích je ukázán způsob použití knihovny, jenž technologům výroby může pomoci lépe optimalizovat výrobní procesy, což je dáno zavedením jisté formy inteligentního rozhodování v procesu výroby.

Na případu kruhového dopravníku je ukázán celkový přínos mojí práce. Jedná se o poměrně komplexní studii reálného problému, která pohlíží na předložený problém z mnoha hledisek. Studii dopravníku lze považovat za metodologické doporučení pro podobné projekty a to zejména v oblasti rozdělení celého systému na jednotlivé subsystémy a také v oblastech jejich komunikace.

Po vyhodnocení výsledků simulačních studií byly identifikovány nedostatky týkající se specifikace obou studií uvedených v této práci a navržena možná rozšíření a vylepšení. Uvedená vylepšení by mohla tento výzkum ještě významně rozšířit a výsledky by mohly ještě lépe odrážet význam heterogenního modelování ve spojení s umělou inteligencí.

Průběžné výsledky tohoto výzkumu byly s úspěchem prezentovány na studentské konferenci EEICT (viz [10]), kde má prezentace byla oceněna druhým místem v dané kategorii.

Literatura

[1] Kindler, E.: Simulační programovací jazyky, Praha, SNTL 1980

[2] Wikipedia, http://wikipedia.cz (prosinec 2006)

[3] Hrubý, M.: Prostředí pro modelování heterogenních systémů [Disertační práce], VUT Brno

[4] Hrubý, M.: přednášky pro předmět Modelování a simulace, VUT Brno

[5] Peringer, P.: FIT VUT, Brno. SIMulation LIBrary for C++, http://www.fit.vutbr.cz/~peringer/SIMLIB (prosinec 2006)

[6] Kelemen, J., Ftáčník, M., Kalaš, I., Mikulecký P..: Základy umelej inteligencie, Bratislava, ALFA 1992

[7] Lukasová, A.: Formální logika v umělé inteligenci, Brno, Computer Press, 2003

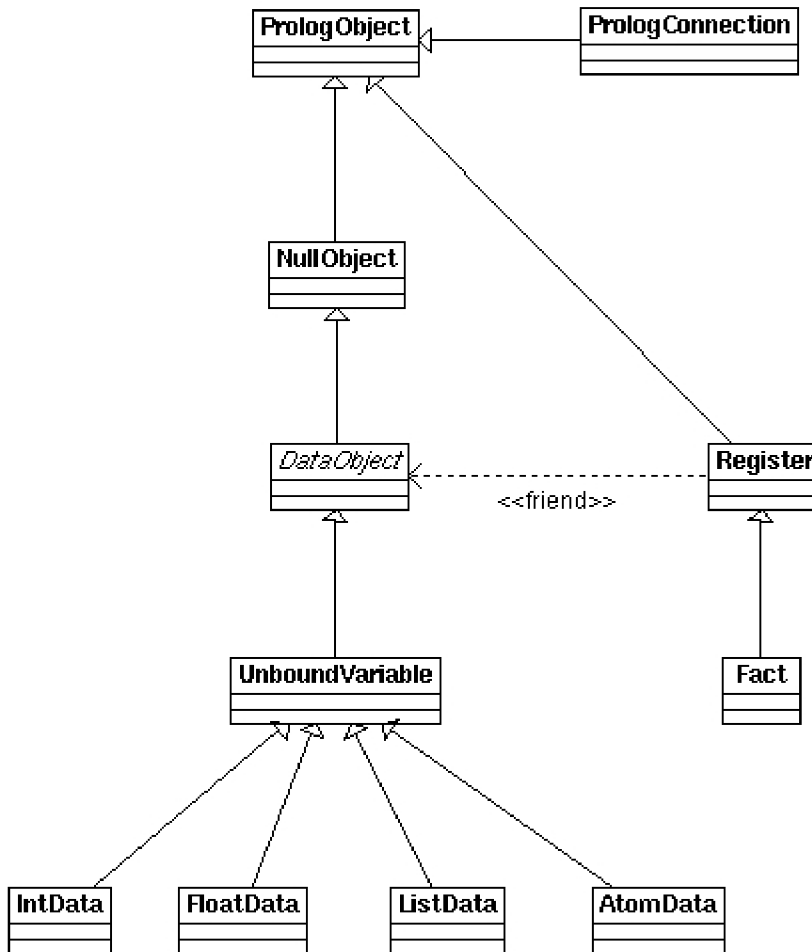
[8] Wielenmaker, J.: Dept. Of Social Science Informatics, Amsterdam. SWI-Prolog Reference Manual http://www.cse.psu.edu/~catuscia/teaching/prolog/Manual/Title.html (prosinec 2006)

[9] Anticipation, http://www.anticipation.info/ (prosinec 2006)

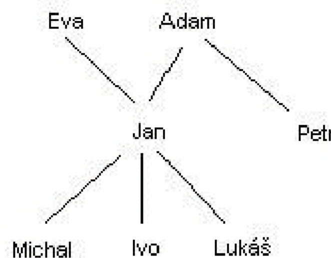
[10] Hrabcová P., The Prolog with C++ Interconnection Library, In Proceedings of the 12th Conference STUDENT EEICT 2006, Volume 2, VUT Brno, 2006, str. 251

Přílohy

I. Objektový návrh knihovny



II. Rodokmen použitý v ukázkovém příkladu



III. Zdrojový kód programu rodokmen

```
#include "PrologLib.h"

class MyFact: public Fact
{
	public:
		MyFact(std::string aName): Fact(aName),
		osoba1(*this, 0),osoba2(*this,1){ }
		AtomData osoba1;
		AtomData osoba2;
};

int main(int argc,char *argv[])
{
	PrologConnection conn("database",argc,argv);

	MyFact otec("otec");

	otec.osoba1="adam";
	otec.osoba2="jan";

	otec.insert(&conn);

	otec.osoba1="adam";
	otec.osoba2="petr";

	otec.insert(&conn);

	otec.osoba1="jan";
	otec.osoba2="michal";

	otec.insert(&conn);

	otec.osoba1="jan";
	otec.osoba2="ivo";

	otec.insert(&conn);

	otec.osoba1="jan";
	otec.osoba2="lukas";

	otec.insert(&conn);

	MyFact matka("matka");
	matka.osoba1="eva";
```

```
	matka.osoba2="jan";

	matka.insert(&conn);

	std::cout<<"Otec a syn jsou: ";
	otec.osoba1.unsetObject();
	otec.osoba2.unsetObject();
	otec.osoba2="ivo";

	FactList * list;
	FactList::iterator it;

	list = otec.call(&conn);

	for(it = list->begin();it != list->end();++it)
	{
		(*it)->printMap();
	}
	std::cout<<std::endl;
	std::cout<<"Sourozenci Michala:"<<std::endl;
	MyFact bratr("sourozenec");

	bratr.osoba1="michal";
	bratr.osoba2="jeronym";

	bratr.insert(&conn);

	bratr.osoba2.unsetObject();

	Fact::destroyList(&list);

	list = bratr.call(&conn);

	for(it = list->begin();it != list->end();++it)
	{
		bratr.copy(**it);
		bratr.osoba2.print();
		std::cout<<std::endl;
	}

	return 0;
}
```