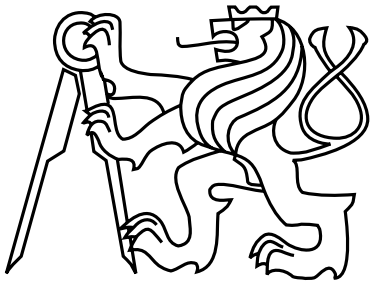
Na tomto místě bude oficiální zadání vaší práce

- Toto zadání je podepsané děkanem a vedoucím katedry,
- musíte si ho vyzvednout na studiijním oddělení Katedry počítačů na Karlově náměstí,
- v jedné odevzdané práci bude originál tohoto zadání (originál zůstává po obhajobě na katedře),
- ve druhé bude na stejném místě neověřená kopie tohoto dokumentu (tato se vám vrátí po obhajobě).

České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů



Diplomová práce

Dokončení projektu Migdb

Bc. Martin Lukeš

Vedoucí práce: Ing. Ondřej Macek

Studijní program: Otevřená informatika, strukturovaný, navazující Magisterský

Obor: Softwarové inženýrství a interakce

13. prosince 2014

Poděkování

Chtěl bych poděkovat vedoucímu své diplomové práce Ing. Ondřeji Mackovi za pomoc s vypracovávánáním této práce. Dále bych chtěl poděkovat svým kolegům z týmu Migdb, obzvláště Martinu Mazanci, kteří svými připomínkami napomáhali ke zkvalitnění této práce a zahlazení některých nepřesností. V neposlední řadě bych chtěl poděkovat firmě CollectionsPro s.r.o, jež přišla s původní myšlenkou, která vedla k vytvoření Migdb týmu.

Prohlášení

Prohlašuji, že jsem práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.
Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 22. 5. 2014 .....

Abstract

This work is concerned with specifying the contract and implement the transformation changes of the application model into changes of the database model. It also deals with the automation of the derivation of the changes applied to one model leading to another without loss or with minimal loss of stored data.

Abstrakt

Tato práce se zabývá upřesněním kontraktu a realizací transformací změn aplikačního modelu na změny modelu databázového. Dále se zabývá automatizací odvození změn vedoucích z jednoho modelu k druhému bez ztráty či s minimální ztrátou uložených dat.

Obsah

1 Úvod **1**
1.1 Motivace . . . 1

2 Projekt Migdb **3**
2.1 Framework Migdb . . . 3
2.2 Moduly frameworku . . . 5
2.3 Metamodely . . . 6
2.4 Aplikační metamodel . . . 9
2.5 Metamodel struktury aplikace . . . 9
2.6 Databázový metamodel struktury . . . 10
2.7 Relační model . . . 13
2.8 Operace nad aplikačním modelem . . . 14
2.8.1 Seznam aplikačních operací . . . 14
2.8.2 Rozdělení aplikačních operací . . . 17
2.8.3 Vlastnosti operací . . . 18
2.9 Databázové operace . . . 20
2.10 QVTo . . . 22
2.10.1 Ukázka kódu . . . 24
2.11 ORMo (ORM operací) . . . 25

3 Dokončení projektu Migdb **29**
3.1 Historický vývoj operací . . . 29

4 Popis problému, specifikace cíle **31**
4.1 Řešení problému rozpoznávání operací . . . 31
4.1.1 Diff a delta notace . . . 31
4.1.2 Rozpoznávání operací . . . 34
4.1.3 Obecné principy model matching . . . 34
4.1.4 Graph matching . . . 35

5 Vytvořené algoritmy rozpoznávání operací **37**
5.1 Stavový algoritmus . . . 37
5.1.1 Energie modelu . . . 39
5.2 Návrh ze studia článků . . . 44
5.3 Základní implementace . . . 44
5.4 Složitější implementace . . . 45

5.4.1 Diff elementy 46

6 Testování projektu Migdb **47**

7 Ukázka zdrojového kódu práce **49**

8 Závěr **53**
8.1 Další poznámky 55
8.1.1 České uvozovky 55

9 Seznam použitých zkratek **57**

10 UML diagramy **59**

11 Instalační a uživatelská příručka **61**

12 Obsah přiloženého CD **63**

Literatura **65**

Seznam obrázků

2.1 Rozdělení vrstev softwaru - převzato z [Maz14] 4
2.2 Rozdělení vrstev softwaru - převzato z [Tar12] a částečně upraveno 5
2.3 Aplikační metamodel v počátku vývoje obrázek převzat z [Luk11] 7
2.4 Aplikační metamodel v průběhu vývoje obrázek převzat z [Jez12] 8
2.5 Rootové elementy aplikačního modelu 9
2.6 Struktura aplikačního metamodelu 10
2.7 Struktura databázového metamodelu 11
2.8 Struktura databázového metamodelu v průběhu vývoje, obrázek převzat z [Tar12] 12
2.9 Struktura databázového metamodelu v průběhu vývoje, obrázek převzat z [Luk11] 13
2.10 Ukázka operace AddParent 19

4.1 Diff model převzatý dle [Cic08] 33
4.2 Seznam diff elementů 33
4.3 Typy graph matchingu 36

5.1 Počáteční model 41
5.2 Cílový model 41

12.1 Seznam přiloženého CD — příklad 63

Seznam tabulek

Kapitola 1

Úvod

1.1 Motivace

V průběhu poslední dekády je vyvíjeno více nového softwaru než kdy předtím a současně je i stávající software stále více a častěji modifikován, ať už je to zapříčiněno existencí rozsáhlého legacy systému, špatného návrhu či upravováním funkcionality softwaru. Dá se předpokládat, že díky masivnímu rozšíření informačních technologií, obzvláště mobilních tento trend nejenže bude pokračovat, ale bude i dále na vzestupu.

Díky nutnosti zpracování a ukládání velkého množství dat se již od padesátých let dvacátého století prosazovaly myšlenky vedoucí k vytvoření speciálních systémů k těmto účelům určeným - tento software se v české odborné literatuře nazývá systém řízení báze dat (SŘBD).

Kvůli nutnosti dokumentace a komunikace mezi vývojáři vznikají různé typy modelů. Objektový model aplikace popisuje strukturu aplikace a je doplněn modelem databázovým popisujícím stav databáze. Nejrozšířenějším typem databáze jsou v nynější době databáze relační, které uspořádávají data podle relačního modelu.

Aby byla aplikace funkční, je nutné zajistit konzistenci mezi databázovým a aplikačním modelem. Tento problém byl již vyřešen a jeho řešení bývá v odborných kruzích nazýváno objektově relační mapování(ORM) [wc14c]. Dnešní implementace ORM jsou schopny nejen transformovat aplikační model na model databázový, ale také vyjádřit změnu v struktuře aplikace pomocí skriptů Data definition Language (DDL), podmnožiny jazyka SQL. Tyto skripty pozmění model databázový tak aby odpovídal modelu aplikačnímu. Tato konzistence je zaručena automatickou transformací aplikačního modelu na model databázový, což vývojářům softwaru šetří čas strávený vývojem softwaru.

Problém nastává, jakmile zahrneme do zachování nejen strukturu dat, ale i samotná data. Změnit strukturu dat a zároveň transformovat data tak, aby měla stejnou vyjadřovací schopnost jako původní data je zatím problémem nevyřešeným. Považujme změny aplikačního modelu jako smazání třídy, atributu apod za změny zachovávající informaci.

Práce obsahuje tyto kapitoly :

Kapitola 2

Projekt Migdb

Tato dipomová práce byla napsána v rámci projektu Migdb, který vznikl díky spolupraci akademické sféry se sférou komerční v roce 2011. Komerční sféra byla v tomto případě zastoupena firmou CollectionsPro, s.r.o (dále jen CP) a akademická potom Katedrou počítačů fakulty Elektrotechnické na pražském Českém vysokém učení technickém. Ambiciózním cílem tohoto projektu bylo od pořátku projektu definování ucelené množiny změn, tj. operací, kterými mohou vývojáři změnit model aplikace a transformovat tyto změny do spustitelného SQL skriptu, který změní strukturu databáze a přesune data do odpovídajících elementů z modelu aplikace.

Tato diplomová práce se zabývá se zkoumáním změn aplikačního modelu, jejich popisem a rozpoznáváním změn vedoucích od jednoho aplikačního modelu k druhému. Dále pak dokončuje a upřesňuje kontrakt takzvaných operací nad aplikačním modelem, popisuje jejich transformaci na změny modelu databázového a následným vygenerováním SQL příkazů spustitelných nad relační databází PostgreSQL. Dalším tématem této diplomové práce je automatizace odvození množiny změn ze vstupních dvou modelů.

V rámci Migdb bylo v posledních letech vytvořeny a obhájeny 4 bakalářské práce a 2 diplomové práce členů Migdb.

Jednalo se o tyto bakalářské práce - bakalářská práce mé osoby [Luk11], jež pojednávala o problematice mapování aplikačního modelu na model databázový, bakalářské práce Jiřího Ježka [Jez12] popisující aplikační model a jeho transformace, bakalářská práce Petra Taranta [Tar12] popisující databázový model a jeho transformace a poslední bakalářskou prací je práce popisující testování projektu [Luk13].

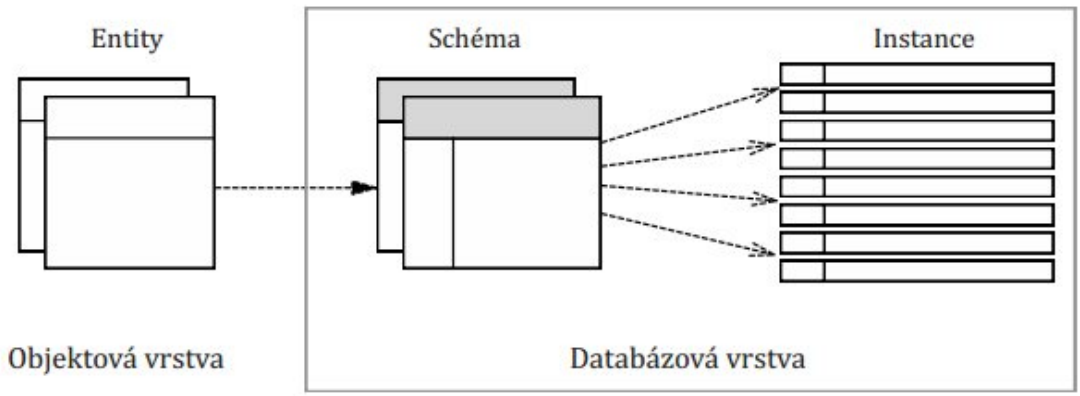
V roce 2014 byla obhájena diplomová práce Petra Taranta [Tar14] formálně specifikující aplikační operace a diplomová práce Martina Mazance [Maz14] definující doménově specifikující jazyk aplikačních operací.

Projekt Migdb byl započat v spolupráci se společností Collections Pro Výsledky dosavadní práce byly v roce 2012 prezentovány jako case-study na prestižní modelové konferenci Code Generation 2012 [PMH12] v Anglickém Cambridge.

2.1 Framework Migdb

Tato sekce je věnována krátkému představní frameworku Migdb. Projekt Migdb se věnuje evolučnímu procesu v průběhu vývoje software, konkrétně jen dvěma částmi - aplikační

a databázovou vrstvou. Rozdělení vrstev software je zobrazeno na obr. 2.1 převzatého z [Maz14]. Objektová vrstva každé aplikace je reprezentována jejím modelem obsahujícím entity aplikace. Databázová vrstva zajišťující perzistenci dat obsahuje jednak schéma definující strukturu databáze, ale také samotné perzistované instance dat.

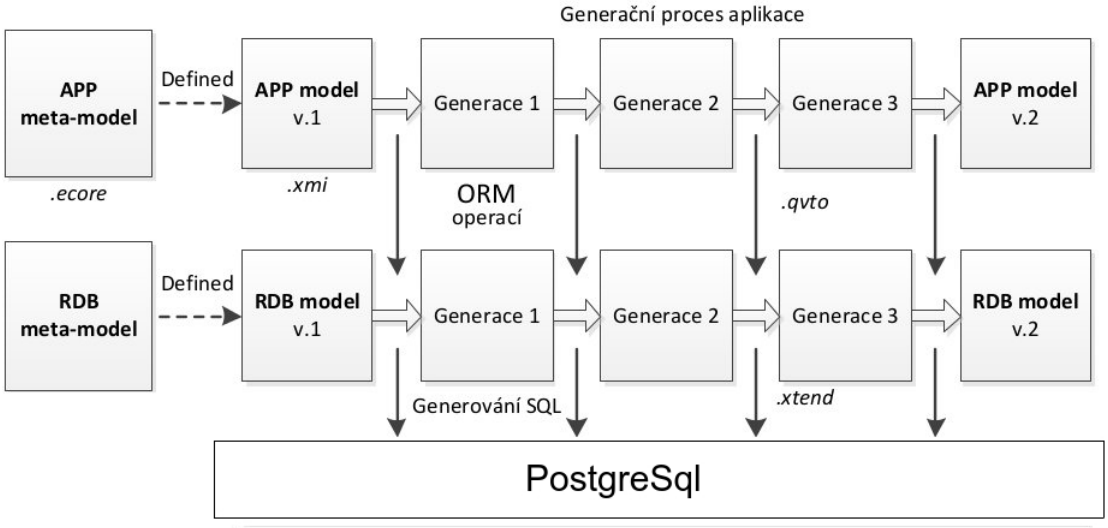


Obrázek 2.1: Rozdělení vrstev softwaru - převzato z [Maz14]

Proces evoluce změn v životním cyklu je definován v [Jez12] následovně:

1. redefinice entit na objektové vrstvě
2. opětovná tvorba databázového schématu dle redefinovaných entit
3. sběr potřebných informací pro migraci dat a příprava migračního skriptu
4. ověření proveditelnosti migračního skriptu
5. migrace dat

Po ověření nadbytečnosti vrstvy ověřující proveditelnost migrovaných skriptů byl tento koncept přepracován do formy zobrazené na obrázku 2.2. Na obrázku vidíme aplikační a databázový metamodel.



Obrázek 2.2: Rozdělení vrstev softwaru - převzato z [Tar12] a částečně upraveno

Instancemi aplikačního APP metamodelu jsou jednotlivé generace aplikačního modelu. Instancemi databázového RDB metamodelu jsou potom generace RDB modelů. Změny, které se provádějí v aplikaci jsou transformovány na databázové operace aplikovatelné na RDB model. Databázové operace jsou potom transformovány na SQL scripty. V původní představě byl do frameworku zapojen i exekutor těchto skriptů nad databází, ale vzhledem ke znovupoužitelnosti SQL souborů byl z frameworku vypuštěn.

V rámci projektu jsem vytvořil ve své bakalářské práci [Luk11] ORM transformaci aplikační struktury na databázovou a následné vygenerování SQL skriptu vytvářející strukturu aplikace. Tento nástroj není ve frameworku použit při nasazení, ale byl použit při testování frameworku - viz kapitola 6. Framework Migdb pracuje oproti původní sekvenční představě iteračně viz obr. 2.1. V první iteraci je první aplikační operace aplikována na aplikační model, transformována do databáze, kde je její obraz (sekvence databázových operací) aplikován na databázový model. Po provedení první iterace prochází tímto cyklem druhá operace, potom třetí ...

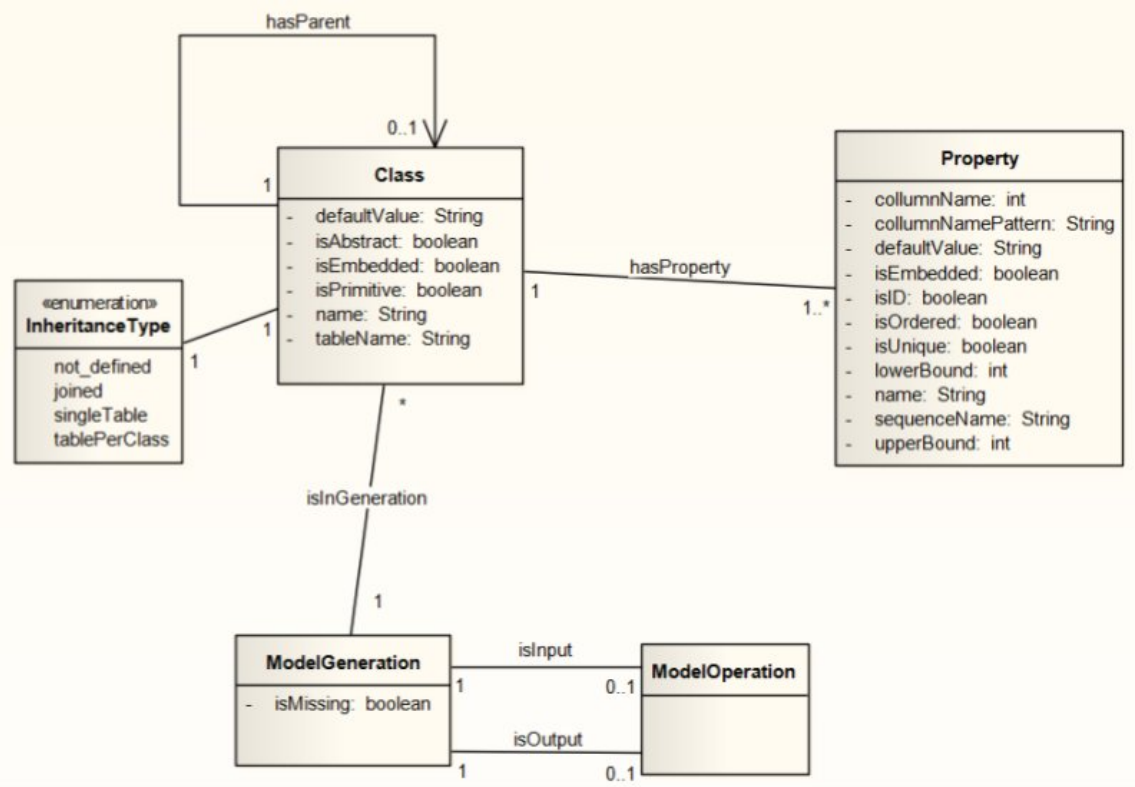
2.2 Moduly frameworku

Framework byl od začátku vývoje používán v rámci Eclipse IDE [Fou14a]. Nad vrstvou aplikační byly vytvořeny 2 moduly. Modul Operations a modul aplikační evoluce. Modul Operations umožňující napsat vyjídřit vstupní operace pomocí textového souboru zapsaném v DSL jazyce. Kromě zápisu tento modul transformuje textově zapsané operace do XMI [OMG14b] souboru pomocí model to text transformačního jazyka Xtend [Fou14b]. Tento modul je podrobně popsán v [Maz14] a není v této práci blíže rozebírán. Modul aplikační evoluce definuje pro každou operaci, jaký vliv bude mít provedení této operace na daný apli-

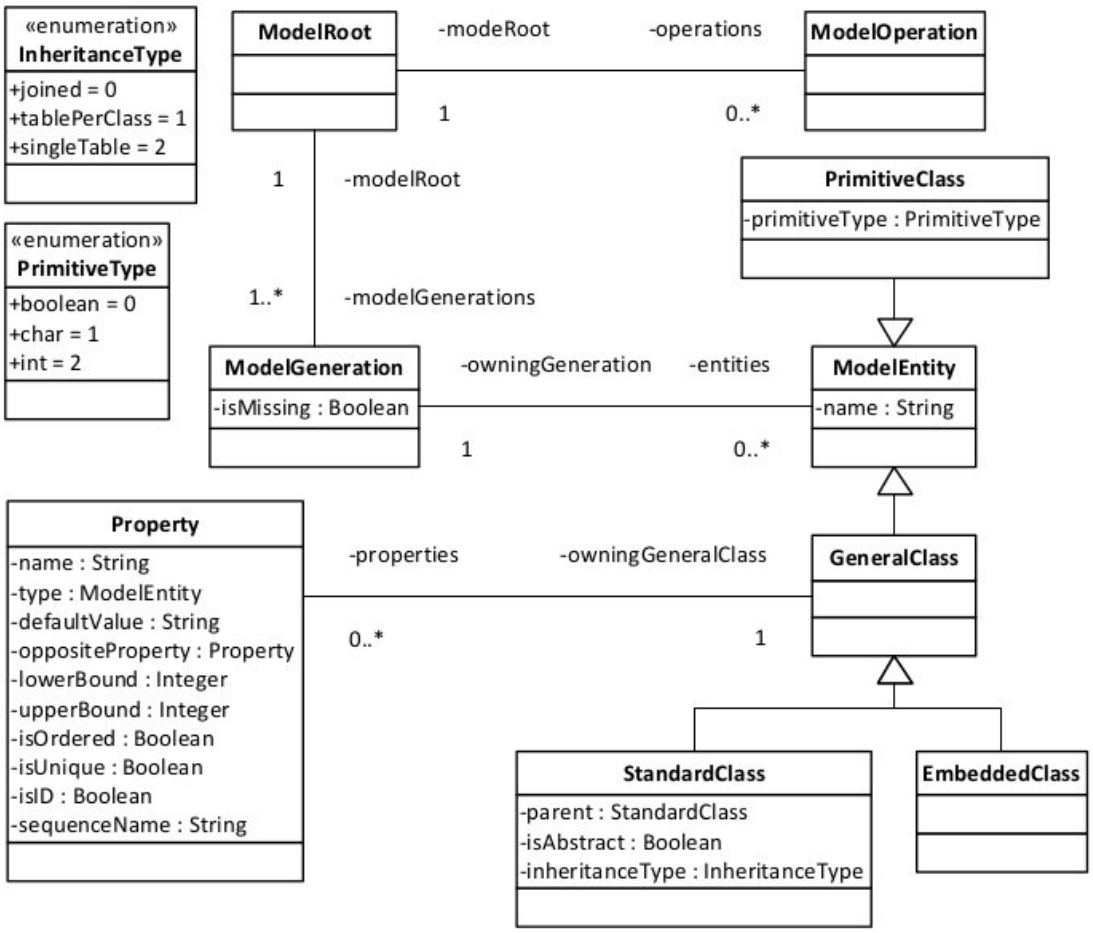
kační model a jaké podmínky musí daný aplikační model splňovat, aby se tato operace dala úspěšně provést. Více v sekci 2.8. Nad databázovou vrstvou je definován modul Databázové evoluce, který pro každou databázovou operaci definuje, jaký bude mít tato operace vliv na databázový model a jaké podmínky musí daný model splňovat, aby se tato operace mohla úspěšně provést. Více v sekci 2.9.
Spojnici mezi aplikačním a databázovým modulem tvoří dva moduly - modul ORM a modul ORMo. Modul ORMo. Modul ORM transformuje model struktury aplikace na model struktury databáze. Tento modul byl prezentován v mé bakalářské práci [Luk11] a není více v této diplomové práci rozebírán. Modul ORMo transformuje seznam aplikačních operací na seznam operací databázových. Tento modul byl popsán v [Jez12] a [Tar12]. Tento modul tvoří motor celého frameworku a jeho popisu je věnována sekce 2.11.
Výstupem frameworku Migdb není databázový model, ale textový soubor s SQL příkazy. Textové výstupy vytvářejí dva moduly generátorů kódu. SQL generátor generuje ze souboru operací výstupní upgrade skript zajišťující samotnou migraci dat. Schema generator potom generuje ze souboru databázové struktury SQL skript vytvářející strukturu databáze. Ani jedním z těchto modulů se nebudu v této práci blíže zabývat. Oba dva generátory byly napsány v jazyce Xtend.
Nejnovějším modulem frameworku je modul OPS recognition. Tento modul se snaží ze dvou vstupních aplikačních modelů odvodit seznam aplikačních operací, které byly provedeny nad zdrojovým modelem a transformovaly ho do modelu výstupního. Tento modul je samostatným nástrojem a pracuje nezávisle na ostatních modulech. Je mu věnována sekce rozpoznávání operací 4.1.
Moduly Aplikační evoluce, databázové evoluce, ORM, ORMo a OPS recognition byly napsány v jazyce QVT Operational (QVTo) [OMG14a]. Za účelem ověření správné funkcionality byl vytvořen projekt Migdb.testing.run, který je zmíněn v kapitole 6.

2.3 Metamodely

Frameworku MigDb je postaven na konceptu MDA [OMG14c] a pro popsání jednotlivých vrstev software zavádí pojem metamodel. Metamodel definuje strukturu popsaných modelů stejně jako model definuje strukturu dat odpovídajících tomuto modelu. Ve frameworku Migdb jsou popsány dva metamodely - aplikační metamodel a databázový metamodel. Aplikační metamodel definuje elementy tvořící strukturu aplikace, množinu aplikačních operací a diff entity použité při rozpoznávání operací.



Obrázek 2.3: Aplikační metamodel v počátku vývoje obrázek převzat z [Luk11]



Obrázek 2.4: Aplikační metamodel v průběhu vývoje obrázek převzat z [Jez12]

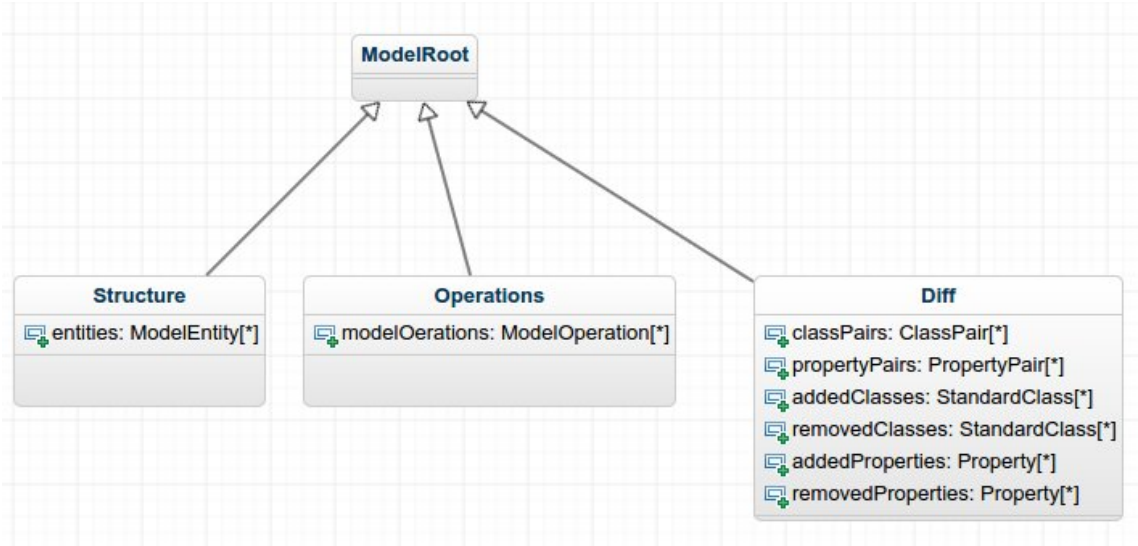
2.4 Aplikační metamodel

Aplikační metamodel byl vytvořen již v ranných fázích projektu Migdb, kdy obsahoval jen elementy tvořící strukturu aplikace a její vztah k aplikačním operacím. Evoluce byla v tehdejší době reprezentována modelově jako sekvence generací. Každá operace měla přiřazenou jednu vstupní generaci a jednu výstupní.

Aplikační metamodel z ranné fáze vývoje je zobrazené na obr. 2.3 viz [Luk11].

Postupem času byl aplikační model měněn viz. 2.4 [Jez12] a dočasně byly přidány entity podporující EmbeddedClass, které v nynější době v modelu již znovu nejsou.

V nynější chvíli došlo k oddělení struktury aplikace od seznamu aplikačních operací a přibyl kořenový element Diff. Na obrázku 2.5 jsou znázorněny kořenové elementy nynějšího aplikačního modelu - každý aplikační model musí obsahovat jeden container - potomka třídy ModelRoot.



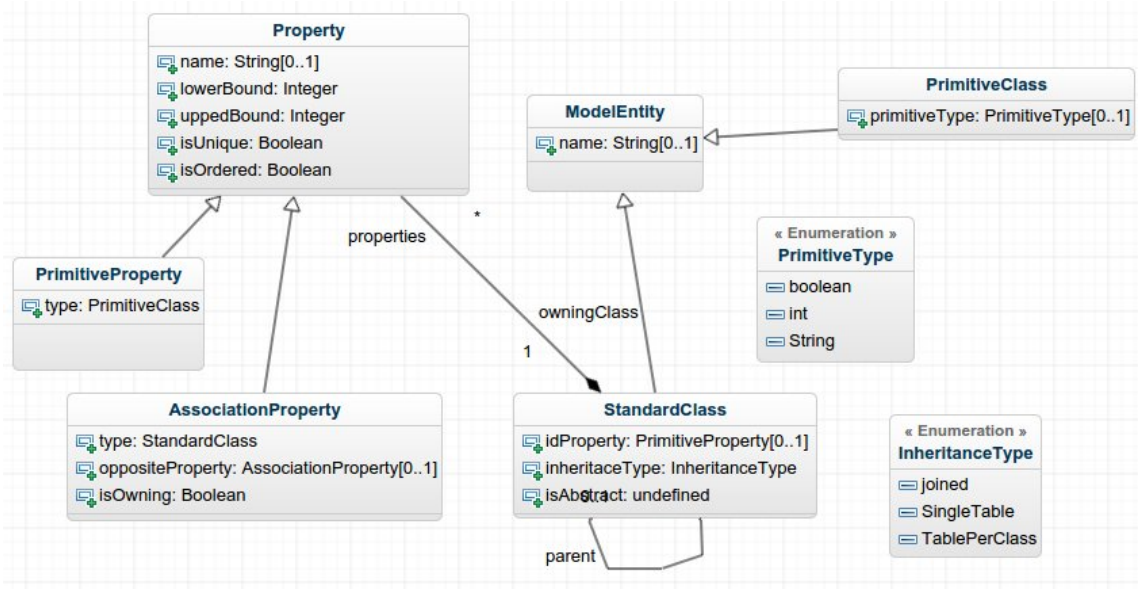
Obrázek 2.5: Rootové elementy aplikačního modelu

2.5 Metamodel struktury aplikace

Metamodel struktury aplikace zachycuje vztahy mezi jednotlivými objekty tvořícími aplikaci. Na obrázku 2.6 jsou zobrazeny elementy patřící do Structury aplikačního modelu.

Structura aplikačního modelu obsahuje množinu elementu *ModelEntity* obsahuje svůj identifikátor name. Primitivní typy programovacího jazyka jsou reprezentované elementem *PrimitiveClass* potomkem ModelEntity. PrimitiveClass obsahuje jen *primitiveType*. Druhým potomkem ModelEntity je *StandardClass* obsahuje specifikaci svého *inheritancetType*, příznak *isAbstract*, referenci na seznam *Property*, referenci na svou *idProperty* a referenci na *parent* StandardClass. Podporujemem pouze jednoduchou dědičnost, proto může mít každá třída maximálně jednu rodičovskou třídu. Element *Property* obsahuje svůj *name*, *lowerBound*

a *upperBound* specifikující svou aritu, pro kolekce důležité atributy *isUnique* a *isUnique*. Potomek *PrimitiveProperty* rozšiřuje svou rodičovskou třídu jen o svůj *type*. *AssociationProperty*, druhý potomek Property, obsahuje také *type* [StandardClass], referenci na *oppositteProperty* pro bidirectional vazby a atribut isOwning, který je přidán z implementačních důvodů.



Obrázek 2.6: Struktura aplikačního metamodelu

Oproti aplikačnímu metamodelu [Jez12] byly odstraněny entity EmbeddedClass a její předek GeneralClass, dále byla zjednodušena třída Property, u níž ubyly atributy defaultValue, sequenceName a atribut isId. Atribut isId byl nahrazen přímou referencí na idProperty ve třídě StandardClass který byl nahrazen referencí.

Koncept generace modelů byl zachován, ale tyto generace nejsou obsaženy z implementačních a testovacích důvodů v jednom souboru, ale ve více souborech.

2.6 Databázový metamodel struktury

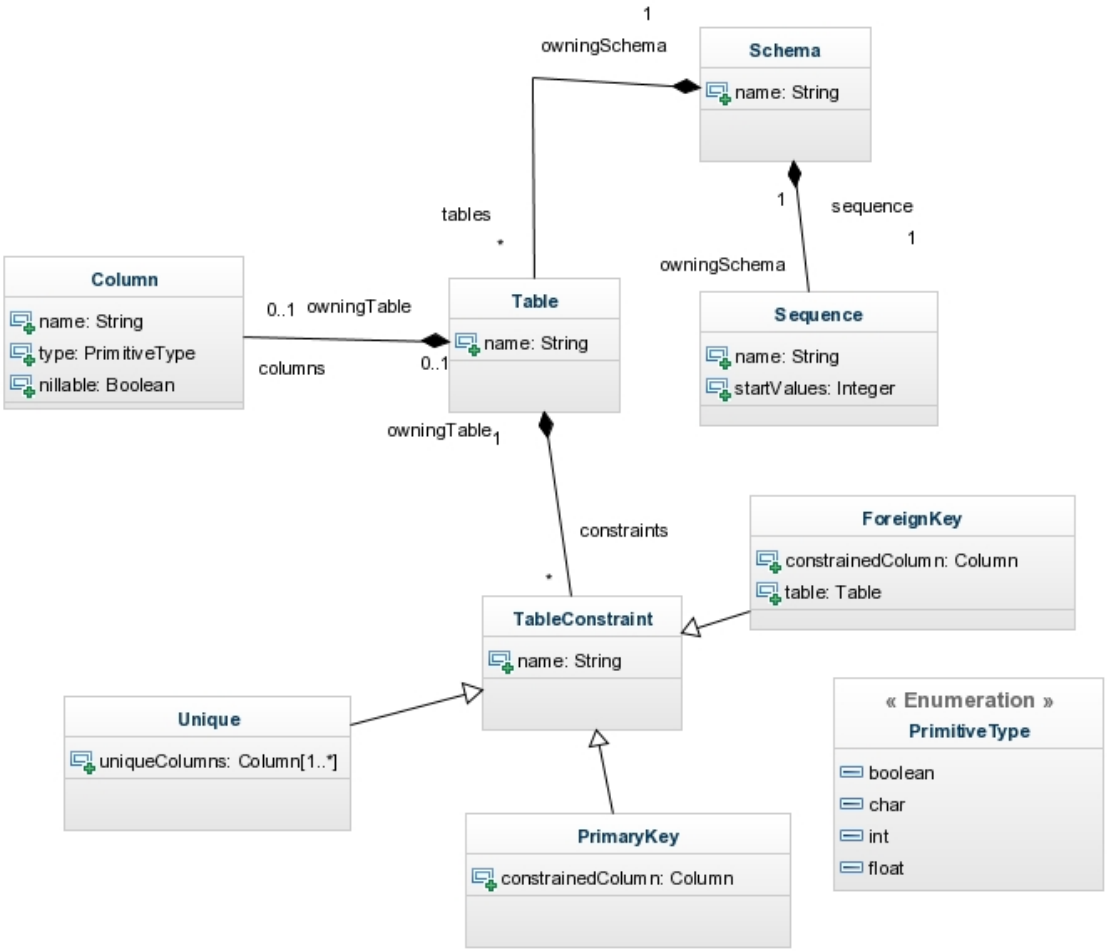
Databázový metamodel od počátku vývoje specifikuje elementy nutné k specifikaci struktury databáze a databázové operace. Základním databázovým konstruktem je *Schema*, které je jednoznačně identifikované svým *name*, obsahuje sezname tabulek a seznam Sequencí. Sekvence je databázový element potřebný k postupnému generování čísel a je identifikovaná pomocí svého *name* a musí obsahovat své startValue. Každá *Table* má své *name*, obsahuje seznam Column a seznam TableConstraint. Každý *Column* má své *jméno*, atribut *nillable*, který povoluje či zakazuje NULL hodnoty v tomto sloupci, *type* [PrimitiveType] a odkaz na vlastnickou tabulu *ownintTable*.

Potomci TableConstraint jsou jednotlivé IO a mají společné *name* zpřístupňující Constraint. TableConstraint *Unique* obsahuje seznan unikátních sloupců a odkaz na vlastnickou

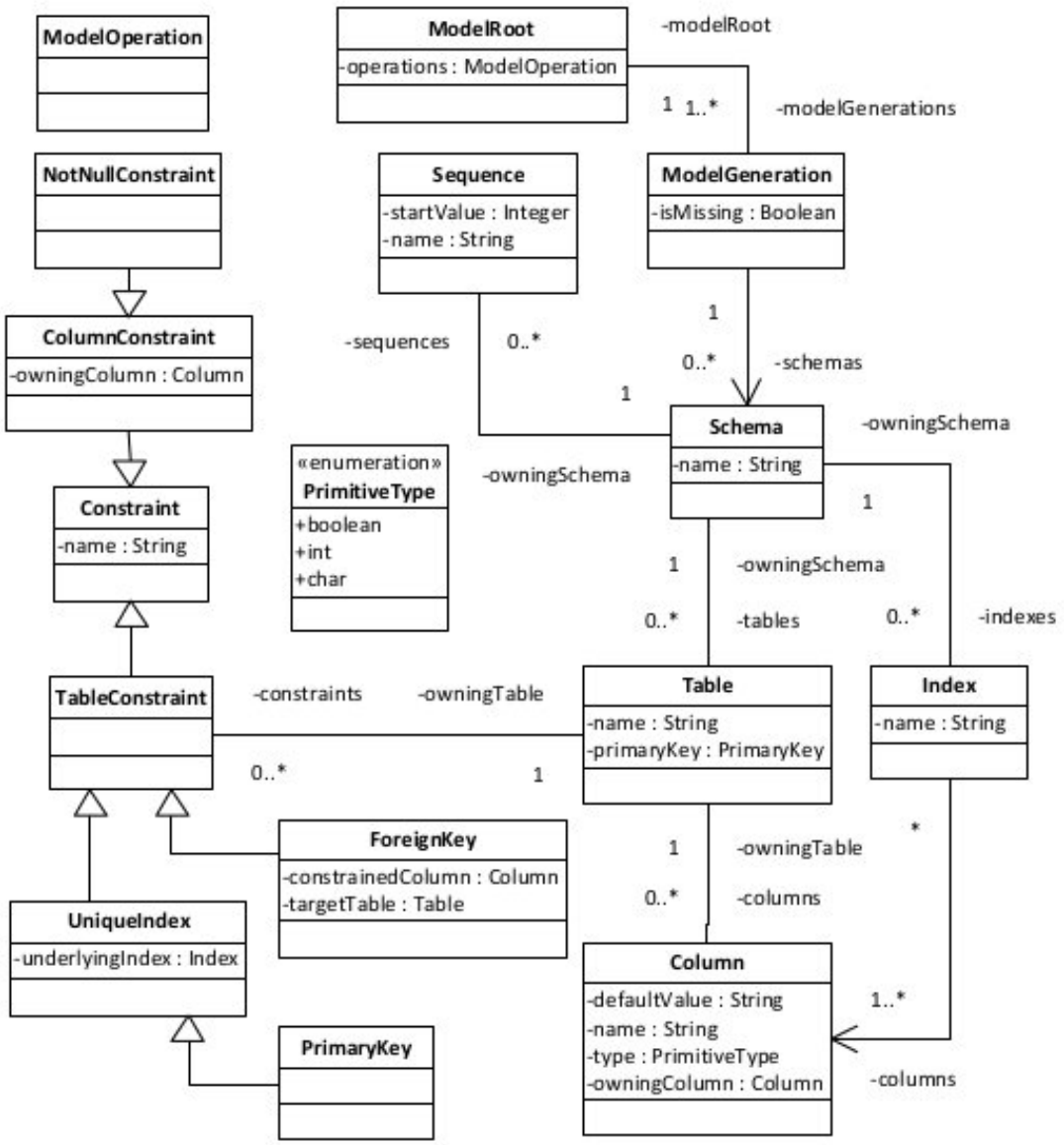
tabulu *ownintTable*. Omezili jsme si *PrimaryKey* tak, aby ho bylo možné definovat jen nad jedním sloupcem *constrainedColumn*. *ForeignKey* je poslední TableConstraint a podobně jako PrimaryKey má omezen počet constrainedColumns na 1 a obsahuje také odkaz na referencovanou tabulku.

Metamodel databázové struktury se ukázal jako celkem dobře definovaný a proto nedocházelo k zásadním změnám.

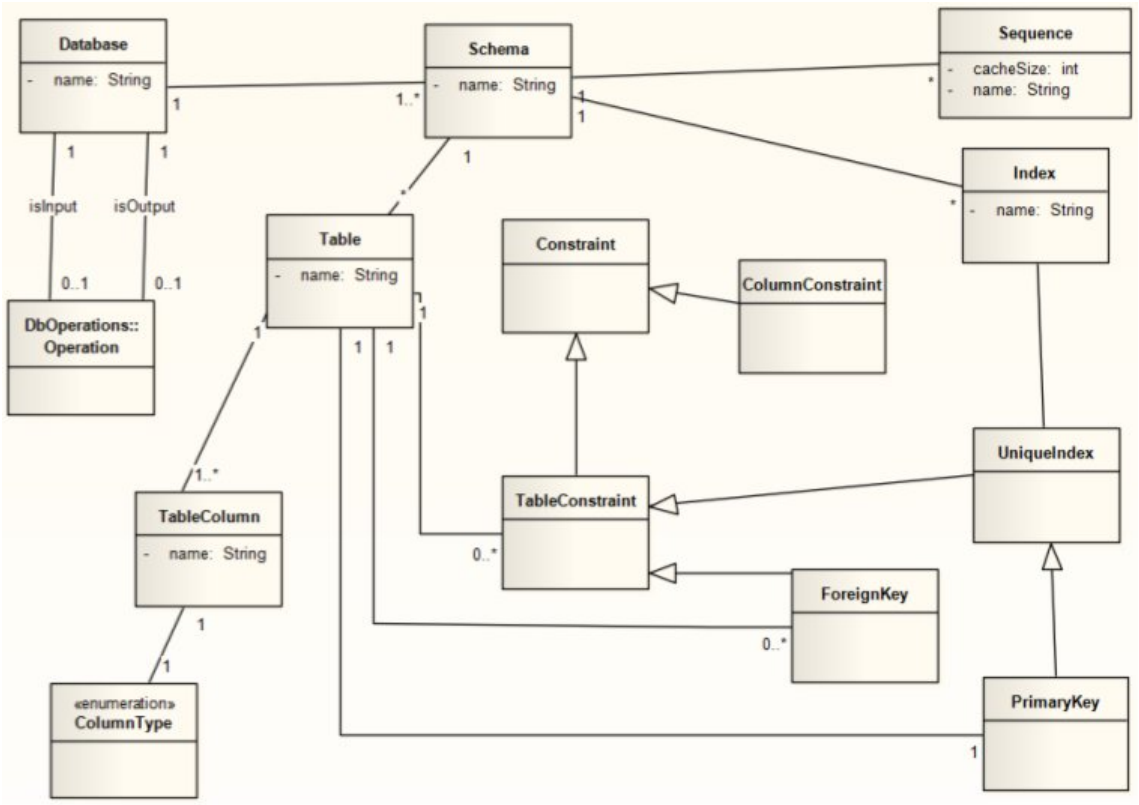
Na obr. 2.7 vidíme aktuální matamodel databázové struktury, na obr. 2.9 vidíme model na počátku vývoje převzatý z [Luk11] a na obrázku 2.8 model v pozdější fázi vývoje převzatý z [Tar12].



Obrázek 2.7: Struktura databázového metamodelu



Obrázek 2.8: Struktura databázového metamodelu v průběhu vývoje, obrázek převzat z [Tar12]



Obrázek 2.9: Struktura databázového metamodelu v průběhu vývoje, obrázek převzat z [Luk11]

Nejvýraznější změnami databázového metamodelu struktury jsou - stejně jako v metamodelu aplikační struktury odstranění generace modelů, odstranění elementu UnderlyingIndex, odstranění elementu ColumnConstraint, nahrazení elementu NotNullConstraint atributem boolean v Column a snížení kardinality Sequence obsažených ve schematu z * na 1.

Myšlenka generace modelů byla zachována, ale jejich případné uchovávání bylo zvoleno ve více oddělených souborech. Element UnderlyingIndex byl shládán nadbytečným, stejně jako element ColumnConstraint. Neatributový element NotNullConstraint byl shledán příliš informačně chudým a byl nahrazen atributem isNillable zachovávajícím stejnou informační hodnotu jako element v původních modelech.

2.7 Relační model

Relační model je dle [Val14a] formální abstrakce využívající relaci jakožto jediný konstrukt. Relace je uspořádaná n-tice souvislých dat R(A1:D1, A2:D2, . . . An:Dn), přičemž "R"je název relace, Ai jsou názvy atributů a Di jsou k nim přidružené typy. K zajištění konzistence jsou v SŘDB používána Integritní omezení (IO). IO jsou dle [Val14b] tvrzení vymezující korektnost DB, stupeň souladu datového obrazu s předlohou (jaká data v databází mohou být a jaká již ne).

Relační algebra definuje pomocí relací a integritních omezení strukturu databáze. Dotazovacím aparátem pro data definovaná pomocí relační algebry je relační algebra. Relační algebra využívá operátorů $\bigcup$ (sjednocení), $\bigcap$ (průnik), $\setminus$ (množinový rozdíl), $\times$ (kartézský součin), selekce značená R(φ) = { u | u ∈ R a $\varphi(u)$} a projekce značená R[C] = u[C] | u ∈ R a operace přirozené spojení T(C) = R * S ={u | u[A] ∈ R a u[B] ∈ S}. [Val14a] dále definuje relačně úplný jazyk jako takový, který umožňuje realizovat relační algebru. Takovým jazykem je například jazyk SQL (Structured Query Language). Relační databáze implementuje relaci tabulkou, její atributy Ai jednotlivými sloupci s typy Di. V dotazovacím jazyce SQL nahrazuje projekci R[a1, . . . , an] operací SELECT a1, . . . an FROM R, operaci selekce R[a = 1] klauzulí where v dotaze select SELECT * FROM R WHERE a = 1;

2.8 Operace nad aplikačním modelem

2.8.1 Seznam aplikačních operací

Operace jsou uvedeny v následujícím seznamu. Kromě regulérních operací, které může vytvořit uživatel jsou v tabulce zde uvedeny i virtuální operace DistributeProperty, MergeProperty a operace ExportProperty, které jsou používány jako pomocné v implementaci složitějších reduktivních a expanzivních operací a manipulují s Property v rámci dědičné struktury tříd. Tyto operace mohou narozdíl od nevirtuálních operací být aplikovány na model, který je z nějakého hlediska nevalidní. Například operace MergeProperty počítá s hierarchií s kolizní property v třídě předka a potomka. Hlavním přínosem virtuálních operací je zabránění duplikace v definici ORMo mapování a zjednodušení kódu.

Operace I: AddStandardClass(name, isAbstract, inHeritanceType)

- Validační podmínky - neexistuje třída s jménem nově vznikající
- Operace vytvoří novou třídu a její id odvozené z názvu třídy

Operace II: RenameEntity(name, newName)

- Validační podmínky - existuje třída s původním jménem, neexistuje třída s novým jménem
- Operace změní název třídy na nový

Operace III: SetAbstract(name, isAbstract)

- Validační podmínky - existuje třída s daným jménem
- Operace nastaví třídě atribut abstract na danou hodnotu

Operace IV: RemoveEntity(name)

- Validační podmínky - existuje třída s daným jménem, neexistuje link na tuto třídu, třída neobsahuje žádné property, neexistuje pro tuto třídu žádný potomek
- Operace odstraní entitu (standardní třídu) z modelu

Operace V: AddProperty(owningClassName, name, typeName, lowerBound, upperBound, isOrdered, isUnique)

- Validační podmínky - zadané bounds jsou validní, v hierarchii dědičnosti neexistuje kolizní property se stejným jménem
- Operace vytvoří v dané třídě novou property se zadanou horní mezí, dolní mezí, typem, seřaditelností a unikátností

Operace VI: RenameProperty(owningClassName, name, newName)

- Validační podmínky - existuje přejmenovaná property v dané třídě, nexistuje property v dané třídy nového jména
- Operace změní název property v dané třídě ze starého na nový

Operace VII: RemoveProperty(owningClassName, name)

- Validační podmínky - musí exitovat vlastnická třída property a v ní odstraňovaná property
- Operace odstraní property z dané třídy

Operace VIII: SetBounds(upperBound, lowerBound)

- Validační podmínky - bounds musí být validní a musí existovat daná tríd a property
- Operace nastaví horní a dolní mez property na nové hodnoty

Operace IX: SetOrdered(owningClassName, name, isOrdered)

- Validační podmínky - musí existovat daná tríd a property
- Operace nastaví property seřaditelnost

Operace X: SetUnique(owningClassName, name, isUnique)

- Validační podmínky - musí existovat daná třída a property
- Operace nastaví property unikátnost

Operace XI: AddParent(className, parentClassName)

- Validační podmínky - musí existovat rodičovská třída a třída potomka, třída potomka nesmí mít nastaveného rodiče
- Operace nastaví třídě předka a přesune namerguje kolizní atributy do rodičovské třídy

Operace XII: RemoveParent(className, parentClassName)

- Validační podmínky - musí existovat třída child a mít nastavenou rodičovskou třídu
- Operace odstraní třídě rodičovskou třídu a rozdistribuje (použije virtuální operaci distribute property) property z rodičovské třídy do třídy původního potomka

Operace XIII: ExtractClass(sourceClassName, extractClassName, associationPropertyName, oppositePropertyName, propertyNames)

- Validační podmínky - musí existovat zdrojová třída, neexistuje property s jménem linku na nově vzniklou třídu, existují exportované property
- Operace vytvoří novou třídu, kterou napojí na původní třídu, exportuje(využije virtuální operaci export property) do nově vzniklé třídy vyjmenované property

Operace XIV: InlineClass(targetClassName, associationPropertyName)

- Validační podmínky - musí existovat cílová třída a musí existovat asociační property typu Inlinované třídy, která má upper bound 1
- Operace exportuje všechny property z inlinované třídy do cílové třídy přes specifikovanou unidirectional asociaci

Operace XV: ChangeUniToBidir(className, associationPropertyName, oppositePropertyName)

- Validační podmínky - v dané třídě musí existovat asociační property s daným jménem a nesmí mít nastavenou opposite property
- Operace vytvoří nový zpětný link s oppositePropertyName k property targetClassName a nastaví správně data do opepositePropertyName

Operace XVI: ChangeBiToUnidir(className, associationPropertyName)

- Validační podmínky - v dané třídě musí existovat asociační property s daným jménem a musí mít nastavenou opposite property
- Operace odstraní opoziční property

Operace XVII: CollapseHierarchy(superClassName, subClassName, isIntoSub)

- Validační podmínky - musí existovat subclass a superclass, subclass musí mít nastavenou superclass jako parenta
- Operace exportuje (aplikuje virtuální operaci) všechny property z jedné třídy do jejího předka a třídy spojí, upraví dědičné vazby

Operace XVIII: ExtractSubClass(sourceClassName, extractedClassName, extractedPropertyNames)

- Validační podmínky - musí existovat třída s name sourceClassName a nesmí existovat třída s jménem extractedClassName, v třídě sourceClass musí existovat property s názvy z kolekce extractedPropertyNames
- Operace vytvoří třídě nového potomka a exportuje(aplikuje virtuální operaci export property) do něj vyjmenované property

Operace XIX: ExtractSuperClass(sourceClassesName, extractParentName, propertyNames)

- Validační podmínky - musí existovat třída s name sourceClassName a nesmí existovat třída s jménem extractedParentName, v třídě sourceClass musí existovat property s názvy z kolekce propertyNames
- Operace vytvoří třídě nového předka a přesune do něj vyjmenované property, pokud měla původní třída předka nastaví tohoto předka rodičem nově vzniklé třídě

Operace XX: PullUpProperties(childClassName, pulledPropertiesNames)

- Validační podmínky - musí existovat childClass a mít nastavenou rodičovskou třídu, v třídě potomka musí existovat properties z kolekce pulledPropertiesNames, v okolních subhierarchiích nesmí existovat properties z této kolekce
- Operace exportuje(aplikuje virtuální operaci export property) property do rodičovské třídy

Operace XXI: PushDownProperties(childClassName, pushedPropertiesNames)

- Validační podmínky - musí existovat class s childClassName a mít nastavenu parentClass, v třídě potomka musí existovat properties z kolekce pushedPropertiesNames
- Operace exportuje(aplikuje virtuální operaci export property) vyjmenované property do třídy potomka a přesune JEN data potomka

Operace XXII: ExportProperty(exportedPropertyName, className)

- virtuální operace
- Operace přesune property a data v ní obsažená v rámci hierarchie do cílové třídy

Operace XXIII: DistributeProperty(distributedPropertyName, className)

- virtuální operace
- Operace zduplikuje strukturu v rámci hierarchie dané property do cílové třídy a přesune data přiřazená této třídě

Operace XXIV: MergeProperty(mergedPropertyName, className)

- virtuální operace
- Operace přesune data zdrojové property do cílové property a smaže strukturu původní property

V průběhu vývoje existoval koncept ComposedOperation a AtomicOperation, kdy každá operace byla buď Composed nebo Atomic, každá Composed operace byla na aplikační vrstvě nejdříve dekomponována na set atomických, které se později vykonaly a mapovaly přes ORMo na databázové operace. Tento koncept jsme zavrhli, protože jsme nedokázali dekomponovat správně některé operace a obzvláště pořadí ORMo obrazů nám dělalo problémy. Nicméně v nynější chvíli by se tento koncept mohl navrátit po přidání konceptu mirrored operations, ale asi by bylo nutné předefinovat či přidat některé aplikační operace.

2.8.2 Rozdělení aplikačních operací

Operace nad aplikačním modelem je možné dělit podle dvou kritérií - 1. nad jakým typem modelové entity pracují, 2. jaký je charakter/význam pro tito entity daná operace má. Operace byly rozděleny podle obou kritérií spíše formálně. Všechny operace jsou potomkem generické operace ModelOperation. Rozdělení podle druhého kritéria vzniklo až po přidání funkcionality rozpoznávání operací.

První kritérium dělí aplikační operace na operace pracující s třídami a operace pracující pouze s properties daných tříd. Příkladem operací pracujících s třídami jsou operace AddStandardClass, AddParent a RemoveEntity. Příkladem operací pracujících s properties jsou operace AddProperty, RemoveProperty, SetAbstract.

Podle druhého kritéria je možné rozdělit operace nad aplikačním modelem do 5 skupin - konstruktivní, destruktivní, expanzivní, reduktivní a modifikační operace. Konstruktivní operace jsou takové, které po své aplikaci vytvoří 1 novou entitu v výsledném modelu, která nemá žádné vazby na jiné entity. Příklady aditivní operace je operace AddClass.

Destruktivní operace je opak konstruktivní, v vstupním modelu existuje entita a ta je aplikaci destruktivní operace odstraněna. Příkladem destruktivní operace je operace RemoveProperty.

Operace expanzivní přídává do výstupního modelu jednu entitu, čímž se podobá operaci konstruktivní, nicméně zároveň je vázána na jinou entitu stejného typu a zmenšuje její obsah. Příkladem expanzivní operace je ExtractClass.

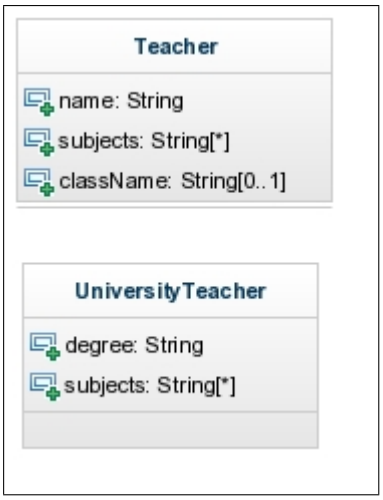
Reduktivní operace entitu z vstupního modelu odstraní a zároveň entitě, která je pro operaci řídící změní obsah. Příkladem této operace je InlineClass. Reduktivní operace jsou inverzní k operacím expanzivním.

2.8.3 Vlastnosti operací

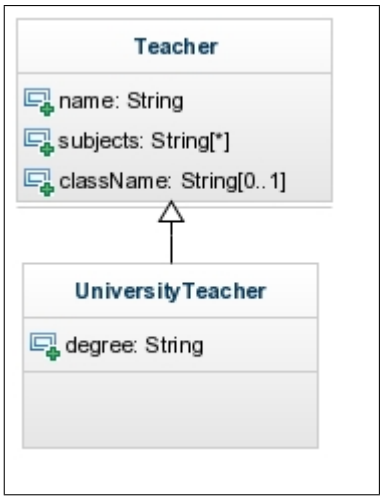
V [Cic08] Antonio Cincetti popisuje některé specifické vlastnosti jako je invertovatelnost a rozložitelnost operací. Je nutné říci, že operace zmiňované v literatuře pracují jen se strukturou dat, nikoliv s daty samotnými a jsou kontextově nezávislé - tyto operace jsou tvořeny téměř výlučně konstruktivními a destruktivními operacemi. Cincetti rozděluje zmiňuje, že v minulosti byly operace aplikovatelné na jeden konkrétní model (intensional) a modernější differenční modely jsou tzv. extensional - je možné je aplikovat na jakýkoliv model, například na paralelní vývojové větve.

V projektu Migdb jsou operace invertovatelné se znalostí původního modelu. Například u operace RemoveParent(childClass) nezískáme parentClass přímo z operace, ale musíme ho dopočítat ze vstupního modelu.

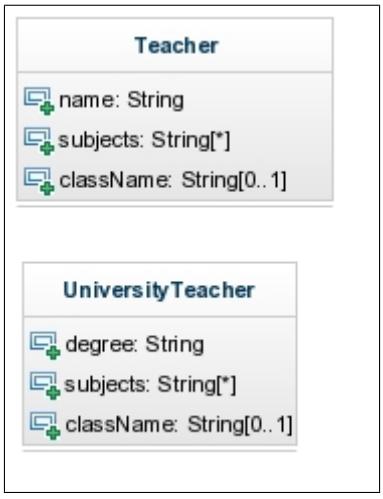
Problémem je, že i po odvození inverze nemusí vést aplikace operace do stejného vstupního modelu. Pokud například na stav 2.10a aplikujeme operaci AddParent(Teacher, UniversityTeacher), získáme cílový stav 2.10b. Pokud se chceme vrátit zpět z 2.10b do výchozího stavu, měli bychom aplikovat operaci RemoveParent(UniversityTeacher), nicméně aplikace této operace nepovede do výchozího stavu 2.10a, ale do stavu 2.10c. Po aplikaci operace RemoveParent bude v třídě UniversityTeacher navíc property class. Tento stav je zapříčiněn vývojem operace AddParent - v původní verzi operace nemohla být použita na jakékoliv třídy s kolizními atributy, ale shledali jsme tuto operaci nepoužitelnou - většinou přidáváme supertyp třídě, pokud je třída potomka speciálním typem třídy rodičovské, což se ale v drtivé většině případů projevuje kolizními atributy. Možným odstraněním tohoto problému by bylo přidání informací o distribuovaných properties do operace RemoveParent, čímž bychom se nicméně odklonili od cílu minimalizovat operace.



(a) AddParent(Teacher, UniversityTeacher)



(b) Výsledek po aplikaci operace AddParent



(c) Výsledný stav po aplikaci RemoveParent

Obrázek 2.10: Ukázka operace AddParent

2.9 Databázové operace

Evoluce na databázové vrstvě nemá vliv na vygenerování SQL kódu a je jen jakousi simulací změn, které se provedou na databázové vrstvě, tudíž se může jevit nadbytečnou. Tato simulace může odhalit chyby ve vygenerovaném SQL skriptu podstatně rychleji oproti generování databáze a aplikaci migračních skriptů, které jak z praxe víme často probíhají v řádu desítek minut až několika hodin.

Databázové operace reprezentují změny proveditelné na úrovni databáze. Mělo by z nich být možné generovat SQL kód jednoduchou Model-to-text transformací zajišťovanou modulem Generator.xtend.

Seznam operací s jejich validačními podmínkami:

Operace I: AddSchema(name)

- Vytvoří nové schéma s zadaným jménem
- Nesmí existovat schéma s zadaným jménem

Operace II: AddSequence(owningSchemaName, name, startValue)

- Vytvoří v cílovém schématu sekvenci s zadanou startovní hodnotou
- Musí existovat schéma, do kterého se vkládá, v něm nesmí existovat sequence s jménem name

Operace III: AddTable(owningSchemaName, name)

- V daném schématu vytvoří tabulku, id sloupec této tabulky a primární klíč odvozený z jména tabulky
- Musí existovat dané schéma, v němž nesmí existovat tabulka s jménem name

Operace IV: AddColumn(owningSchemaName, owningTableName, name, type)

- Vytvoří v daném schématu a tabulce column s zadaným primitivním typem
- Musí existovat daní schéma, tabulka a v dané lokaci nesmí existovat daný sloupec

Operace V: AddPrimaryKey(owningSchemaName, owningTableName, constrainedColumnName, name)

- Vytvoří v daném schématu a tabulce nad constrainedColumn Primární klíč s daným jménem
- Musí existovat dané schéma, daná tabulka, daná column, nesmí existovat constraint s daným jménem

Operace VI: AddForeignKey(owningSchemaName, owningTableName, constrainedColumnName, name, targetTableName)

- Vytvoří v daném schématu a tabulce cizí klíč s daným jménem, který referencuje IdColumn cílové tabulky
- Musí existovat dané schéma, daná tabulka, daná column, nesmí existovat constraint s daným jménem, musí existovat targetTable

Operace VII: AddUnique(owningSchemaName, owningTableName, constrainedColumnNames, name)

- Vytvoří v daném schématu a tabulce unique constraint s daným jménem nad zadanými sloupci
- Musí existovat dané schéma, musí existovat daná tabulka, musí existovat dané constrainované sloupce, nesmí existovat constraint s jménem name

Operace VIII: AddNotNull(owningSchemaName, owningTableName, constrainedColumnName)

- Nastaví v daném schématu a tabulce cílové property hodnotu notNull na true
- Musí existovat dané schéma, daná tabulka, daná column

Operace IX: RemoveNotNull(owningSchemaName, owningTableName, constrainedColumnName)

- Nastaví v daném schématu a tabulce cílové property hodnotu notNull na false
- Musí existovat dané schéma, daná tabulka, daná column

Operace X: RenameTable(owningSchemaName, name, newName)

- Změní cílové tabulce jméno na nové
- Musí existovat dané schéma, daná tabulka, nesmí existovat tabulka s novým jménem

Operace XI: RenameColumnowningSchemaName, owningTableName, name, newName

- Přenastaví v daném schématu a tabulce jméno z name na hodnotu newName
- Musí existovat dané schéma, daná tabulka, daná column

Operace XII: RemoveTableowningSchemaName, name

- Odstraní z daného schematu tabulku s jménem name
- Musí existovat dané schéma, daná tabulka

Operace XIII: RemoveColumnowningSchemaName, owningTableName, name

- Odstraní v daném schématu a tabulce column s jménem name
- Musí existovat dané schéma, daná tabulka, daná column, která nesmí na sobě mít constraint PrimaryKey, ForeignKey nebo Unique

Operace XIV: RemoveConstraintowningSchemaName, owningTableName, name

- Přenastaví v daném schématu a tabulce column s jménem name
- Musí existovat dané schéma, tabulka a column

Operace XV: RemoveSequenceowningSchemaName, name

- Odstraní v daném schématu sequence s jménem name

- Musí existovat dané schéma a daná sequence

Operace XVI: UpdateRowsowningSchemaName, sourceTableName, sourceColumnName, targetTableName, targetColumnName, selectionWhereCondition, safeWhereCondition

- v daném schématu updatuje hodnoty z tabulky sourceTable hodnoty z sourceColumns a nastaví je do taragetColumns tabulky targetTable pro instance splňující selectionWhereCondition, pozn. aby nebyly nullovány hodnoty, pro které nebyly vybrány hodnoty z sourceTable byla přidána safeWhereCondition
- Musí existovat dané schéma, v něm sourceTable, v ní sourceColumn, v dále musí v schématu existovat targetTable, v ní targetColumn, sourceColumn musí mít stejný typ jako targetColumn

Operace XVII: NillRowsowningSchemaName, tableName, columnName, whereCondition

- Nastraví sloupci v daném schematu a tabulce hodnoty null instancím splňující whereCondition
- Musí existovat dané schéma, daná tabulka, daná column

Operace XVIII: InsertRowsowningSchemaName, sourceTableName, sourceColumnName, targetTableName, targetColumnName, whereCondition

- v daném schématu zkopíruje z tabulky sourceTable hodnoty z sloupce sourceColumns instance splňující whereCondition a vloží je do taragetColumns tabulky targetTable
- Musí existovat dané schéma, v něm sourceTable, v ní sourceColumn, v dále musí v schématu existovat targetTable, v ní targetColumn, sourceColumn musí mít stejný typ jako targetColumn

Operace XIX: DeleteRowsowningSchemaName, tableName, whereCondition

- Operace smaže instance z daného schematu, dané tableName instance splňující whereCondition
- Musí existovat dané schéma, v něm daná table

2.10 QVTo

QVTo je imperativní jazyk, který je součástí standardu QVT definovaným konsorciem Object Management Group (OMG) viz. [OMG14d]. Součástí QVT je jazyk OCL (Object Constraint Language) [OCL14]. Pomocí tohoto jazyka je možné definovat model to model transformaci.

Základními konstrukty jazyka jsou mapping, helper a query. Mapping je konstrukt měníci pomocí nějakých pravidel vstupní element na výstupní. Query je dotazovací konstrukt, který získá potřebnou výstupní informaci z daného elementu. Helper je konstrukt, který má může narozdíl od query měnit své parametry.

```
import orm;
import queries_app;
```

```
import queries_rdb;
import validator_app;

modeltype APP uses "http://www.collectionspro.eu/jam/mm/app";
modeltype RDB uses "http://www.collectionspro.eu/jam/mm/rdb";
modeltype ERR uses "http://www.collectionspro.eu/jam/mm/errors";

transformation orm_run(in inStr : APP, in inOps : APP, out rdbOutOps : RDB, out errorMod

main(){
    var rdbOps : RDB::Operations := rdbOutOps.rootObjects()![RDB::Operations];
    rdbOps := _rdbOperations();
    var errorLog : ErrorLog := errorModel.rootObjects()![ErrorLog];
    errorLog := object ErrorLog{
        errors:= OrderedSet{};
    };

var appOps :APP::Operations := inOps.rootObjects()![APP::Operations];
var appStr :APP::Structure := inStr.rootObjects()![APP::Structure];

    if(canProcessTest(appStr, appOps, errorLog))then{
       appOps.operations->forEach(op){
          op.toRdb(appStr, rdbOps);
       };
       consolidate(rdbOps);
       setApplied(rdbOps);
    }endif;
    log("ending");
}

query canProcessTest(structure : APP::Structure, ops : APP::Operations, inout errorLog :
   if(inOps.rootObjects()![APP::Operations].operations->size() > 1 or
         inOps.rootObjects()![APP::Operations].operations->isEmpty())then{
      log("Unsupported input operations count:" +
         inOps.rootObjects()![APP::Operations].operations->size().repr());
      log("model invalid 1");
      return false;
   }endif;

   if(not inStr.rootObjects()![APP::Structure].isModelValid(errorLog))then{
      log("model invalid 2");
      return false;
   }endif;
   return true;
}
```

Na předchozí stránce je vidět ukázka kódu transformace. Na začátku souboru importujeme knihovny pomocí klíčového slova import, následně deklarujeme typy modelů, které bude naše transformace používat. Pomocí klíčového slova transform deklarujeme, jaké typy parametrů požaduje naše transformace. Vstupním bodem transformace je metoda main. V ní můžeme pomocí klíčového slova var deklarovat proměnné, do kterých přiřazujeme data pomocí operátoru := , na ukázce máme také definované query canProcessTest, které pro vstupní strukturu zjistí, jestli jsou vstupní data validní či ne, případně vypíše původ chyby.

2.10.1 Ukázka kódu

V následujícím kódu je ukázáno validační query pro validaci operace AddTable

```
query RDB::ops::AddTable::isValid(structure : RDB::Structure,
  inout errorLog : ErrorLog, operationIndex : Integer) : Boolean {
    var existSchema : Boolean := checkExistSchema(
                                        self.owningSchemaName,
                                        structure,
                                        errorLog,
                                        operationIndex,
                                        getEvolutionRdbTransformationId());
    var notExistTable : Boolean := checkNotExistTable(
                                        self.owningSchemaName,
                                        self.name,
                                        structure,
                                        errorLog,
                                        operationIndex,
                                        getEvolutionRdbTransformationId());
    return existSchema and notExistTable;
}

helper checkExistSchema(schemaName : String, structure : Structure,
  inout errorLog : ErrorLog, operationIndex : Integer,
  transformationId : String) : Boolean{
    var existSchema : Boolean := structure.containsSchema(schemaName);
    if(not existSchema)then{
        var errorMessage : String := "Schema " + schemaName + " doesn't exist";
        errorLog.errors += _evolutionError(
                                          operationIndex,
                                          errorMessage,
                                          transformationId);
    }endif;
    return existSchema;
}

helper checkNotExistTable(owningSchemaName : String, tableName : String,
  structure : Structure, inout errorLog : ErrorLog, operationIndex : Integer,
```

```
  transformationId : String) : Boolean{
    if(not structure.containsSchema(owningSchemaName))then{
       return false;
    }endif;
    var notExistTable: Boolean := not
                  structure.containsTableInSchema(owningSchemaName, tableName);
    if(not notExistTable)then{
        var errorMessage : String := "Table " + tableName + " exists in schema "
                     + owningSchemaName;
    errorLog.errors += _evolutionError(
                                       operationIndex,
                                       errorMessage,
                                       transformationId);
    }endif;
return notExistTable;
}
```

Je možné zaznamenat, že helpery nejen vrací hodnotu, ale i loggují chyby do errorLogu pro daný index operaxe a danou transformaci, tudíž není možné volat helper checkExistTable a invertovat navrácenou hodnotu, ale bylo nutné napsat helpery kladné i záporné.

2.11 ORMo (ORM operací)

ORMo mapování je transformace, která mapuje elementy z dómény aplikačních operací na elementy z domény operací databázových. Toto mapování mapuje 1 aplikační operaci na 1 až N operací databázových.

Ačkoliv ORM transformace vstupního aplikačního modelu funguje se všemi inheritanceTypy bylo nutné zjednodušit aplikační model tak, aby byla transformace ORMo implementovatelná, proto jsme v rámci týmu Migdb rozhodli o redukci počtu inheritanceTypů na jeden - nejvhodnější typ je joined, který je nejvíce používaným.

Operace I: AddStandardClass(name, isAbstract, inHeritanceType)

- Vytvoří tabulku, id sloupec této tabulky a primární klíč

Operace II: AddProperty(owningClassName, name, typeName, lowerBound, upperBound, isOrdered, isUnique)

Primitivní typ a UpperBoubd = 1 operace přidá do cílové tabulky sloupec pro primitivní property

Primitivní typ a UpperBound != 1 operace přidá tabulku, která je obrazem kolekce, datový sloupec do této tabulky, referenční sloupec a ForeignKey referencující tabulku, která je obrazem vlastnické třídy

Neprimitivní typ a UpperBound = 1 operace přidá do tabulky, která je obrazem vlastnické třídy property a ForeignKey odkazující na tabulku, která je obrazem třídy typu přidávané property

Neprimitivní typ a UpperBound != 1 operace vytvoří vazební tabulku pro neprimitivní property, vloží do ní referenční sloupce na tabulku, která je obrazem vlastnické třídy, a tabulku, která je obrazem třídy typu. Nad vazební tabulkou vytvoří cizí klíče na tabulku, která je obrazem vlastnické třídy, a cizí klíč na tabulku, která je obrazem třídy typu

Operace III: RenameEntity(owningClassName, name, newName)

- Operace změní název tabulky na nový, odstraní a vytvoří PK s novým jménem, odstraní všechny ForeignKey referencující obraz vlastnické třídy a vytvoří nové ForeignKey s pozměněným jménem ,

Operace IV: SetAbstract(name, isAbstract)

isAbstract = true maže data, která náleží pouze dané třídě

Operace V: RemoveEntity(name)

- operace smaže primární klíč, id property a tabulka odpovídající dané třídě

Operace VI: RenameProperty(owningClassName, name, newName)

primitivní typ a UpperBound = 1 přejmenuje property v tabulce, která je obrazem vlastnické třídy property

primitivní typ a UpperBound != 1 přejmenuje datový sloupec, odstraní a vytvoří ForeignKey s novým jménem referencujícím třídu, která je obraz vlastnické třídu property a přejmenuje tabulku obrazu kolekce

neprimititní typ a UpperBound = 1 přejmenuje sloupec v tabulce, která je obrazem vlastnické třídy property, odstraní a vytvoří ForeignKey referující tabulku, která je obrazem třídy typu

neprimitivní typ a UpperBound != 1 přejmenuje vazební tabulku s referenčními sloupci na tabulku, která je obrazem třídy vlastníka property a tabulku, který je obrazem třídy typu asociace, odstraní a vytvoří ForeignKey s novými jmény na obraz třídy vlastníka property a obraz třídy typu

Operace VII: RemoveProperty(owningClassName, name)

primitivní typ a UB = 1 odstraní sloupec z dané tabulky

primitivní typ a UB != 1 odstraní referenci na vlastnickou tabulku, sloupec z tabulky dané kolekce, datový sloupec a smaže kolekční tabulku

neprimitivní typ a UB = 1 odstraní referenci na tabulku vlastníka a referenční sloupec

neprimitivní typ a UB != 1 odstraní reference na vlastnickou tabulku a tabulku typu, datový sloupec a sloupec typu a smaže vazební tabulku

Operace VIII: SetOrdered(owningClassName, name, isOrdered)

isOrdered = true přidá sloupec ordering, přenastaví data a vytvoří unikátní constraint přes typový, referenční a orgering sloupec

isOrdered = false smaže ordering unique constraint a ordering sloupec

Operace IX: SetUnique(owningClassName, name, isUnique)

isUnique = true vytvoří unikátní constraint přes typový a referenční sloupec

isUnique = false smaže unique constraint

Operace X: AddParent(className, parentClassName)

- Aplikuje obraz operace MergeProperty na všechny kolizní property, přidá cizí klíč na rodičovskou třídu

Operace XI: RemoveParent(className)

- aplikuje obraz operace DistrubuteProperty na všechny property rodičovské třídy, odstraní cizí klíč, smaže data třídy potomka z tabulky rodiče

Operace XII: ExtractClass(sourceClassName, extractClassName, associationPropertyName, oppositePropertyName, propertyNames)

- vytvoří novou sekvenci, vytvoří novou tabulku a nové sloupce spolu se sloupcem pro opposite referenci na zdrojovou tabulku, aplikuje obraz operací exportProperty pro každou exportovanou property, vytvoří sloupec referencující nově vzniklou tabulku, updatuje mu hodnoty, smaže vygenerovanou sekvenci

Operace XIII: InlineClass(targetClassName, associationPropertyName)

- aplikuje obraz operací exportProperty, smaže association column a Inlinovanou tabulku

Operace XIV: PullUpProperties(childClassName, pulledPropertiesNames)

- aplikuje obraz operace export property do rodičovské třídy pro každou property s name z pulledPropertiesNames

Operace XV: PushDownProperties(childClassName, pushedPropertiesNames)

- aplikuje obraz exportProperty pro každou property s name z p vyjmenované property do třídy potomka a přesune JEN data potomka

Operace XVI: ExportProperty(virtuální operace)

primitivní typ a UB = 1

primitivní typ a UB !=1

neprimitivní typ a UB = 1

neprimitivní typ a UB !=1

Operace XVII: DistributeProperty(virtuální operace)

primitivní typ a UB = 1

primitivní typ a UB !=1

neprimitivní typ a UB = 1

neprimitivní typ a UB !=1

Operace XVIII: MergeProperty(propertyName, targetClassName)

primitivní typ a UB = 1 updatuje column v cílové tabulce smaže column ze zdrojové tabulky

primitivní typ a UB !=1 vloží řádky do tabulky collection, smaže FK z zdrojové collectionTable(případně odstraní UX a ORD constrainty), smaže data, reference column z zdrovové collection table a nakonec i zdrojovou collectionTable

neprimitivní typ a UB = 1 updatuje column v cílové tabulce smaže column ze zdrojové tabulky

neprimitivní typ a UB !=1 vloží řádky do tabulky collection, smaže FK z zdrojové collectionTable(případně odstraní UX a ORD constrainty), smaže data, reference column z zdrovové collection table a nakonec i zdrojovou collectionTable

Kapitola 3

Dokončení projektu Migdb

Tato diplomová práce si klade za první ze svých cílů dokončit vývoj na projektu Migdb. Tj. doimplementovat a otestovat ORM transformace vzniklé v předešlých fázích projektu, upravit a otestovat generátor SQL, případně upravit aplikační a databázový metamodel.

3.1 Historický vývoj operací

V průběhu modelování operací nad aplikačním modelem jsme se snažili, aby tyto operace byly jednoznačné (strojově zpracovatelné) v rámci daného kontextu, dále vzhledem k nutnosti textového zápisu uživatelem o minimalističnost zápisu. Tyto dva koncepty jdou obecně proti sobě, proto jsme došli k jistému jejich kompromisu uživatelské jednoduchosti zápisu a jednoznačnosti.

Operace v aplikačním modelu se vyvíjely a měnily se jejich parametry, ale současně se měnil i seznam dostupných operací nad aplikačním modelem. Z operací v první verzi modelu byly odstraněny operace MoveProperty, AddPrimitiveClass, SetOpposite a SetType. Operace AddPrimitive byla označena za nadbytečnou, protože není cílem modifikace modelu změna seznamu primitivních tříd, který bývá definován použitým programovacím jazykem a tudíž by měl tento seznam být v vstupní generaci.V průběhu analýzy operace SetOpposite bylo zjištěno, že tato operace má smysl na strukturální úrovni, ale není možné ji aplikovat na obecná data, proto byla tato operace nahrazena dvojicí operací ChangeUniToBidir a ChangeBiToUnidir, které plní nároky kladené na původní operaci a jsou aplikovatelné na datové úrovni. Operace SetType byla prozkoumána, ale nebyla exaktně popsána, nebylo nalezeno její mapování na operace v databázi ani validační podmínky nutné k úspěšné aplikaci operace na aplikační model. Předpokládáme, že by tato operace měla souviset s dědičnými hierarchiemi.

Kapitola 4

Popis problému, specifikace cíle

Dalším cílem, který jsem si před vypracováním diplomové práce stanovil bylo vytvoření a zdokumentování algoritmu generující z dvou vstupních modelů sekvenci operací, jejichž aplikací se model zdrojový transformuje na model koncový.

4.1 Řešení problému rozpoznávání operací

4.1.1 Diff a delta notace

Nejznámějším nástrojen používaným při porovnávání a zjišťování změn dvou textových souborů tzv. patchů [wc14d] je nástroj diff [wc14a] . Diff je založen na algoritmu hledání největší společné podsekvence (LCS) viz [wc14b]. Algoritmus LCS byl analyzován a nebyl shledán jako dostatečným pro námi definovaný problém, jelikož nezohledňuje doménu problému. Výstup algoritmu se dá vyjádřit pomocí delta notace.

Příklad Delta notace za pomoci linuxových nástrojů diff dvou souborů najdeme na výpisu 4.3

Listing 4.1: Man1.java

```
class Man {
        private String name;

        public Man(String name){
                this.name = name;
        }

}
```

Listing 4.2: Man2.java

```
class Man {
        private String name;

        private String surename;
```

```
        public Man(String name){
                this.name = name;
        }

        public Man(String name, String surename){
                this(name);
                this.surename = surename;
        }
}
```

Listing 4.3: Patch Man1 Man2

```
3a4,5
>       private String surename;
>
5a8,12
>       }
>
>       public Man(String name, String surename){
>               this(name);
>               this.surename = surename;
```

Ukázka kódu 4.1 zobrazuje zdrojový kód třídy Man v první verzi. Ukázka 4.2 potom zdrojový kód třídy Man po první naší editaci, kdy jsme do dané třídy přidali nový atribut a nový konstruktor. Diff v delta notaci těchto dvou souborů je ukázán v 4.3. V delta notaci vidíme, že do prvního souboru byly za 3. řádek vloženy řádky 4-5 z druhého souboru - řádek definující nový atribut surename a oddělující prázdný řádek, dále za 5. řádek byly vloženy řádky 8-12 z druhého souboru definující nový konstruktor a uzavírací závorka.

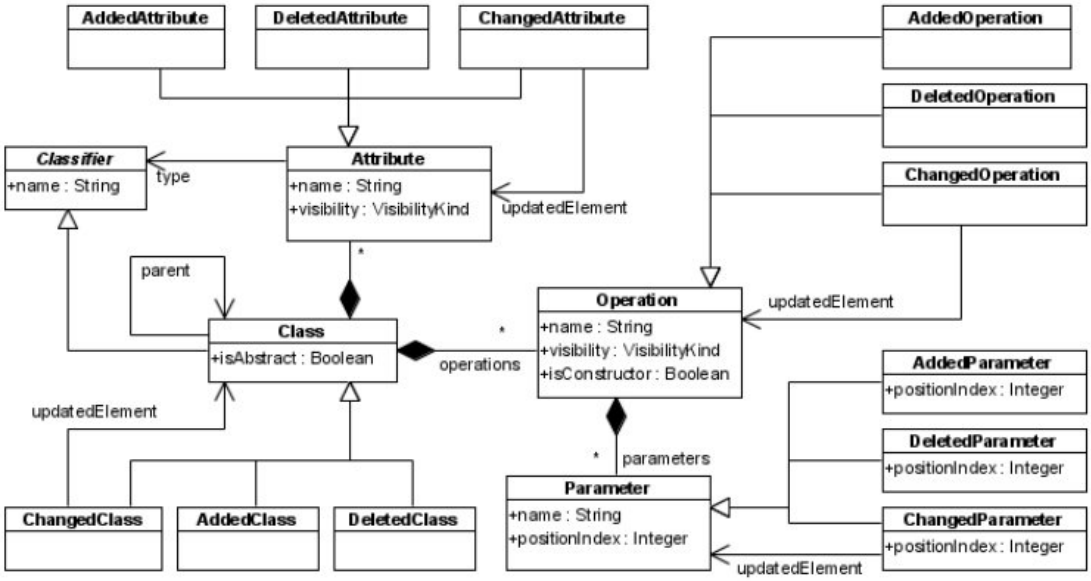
V [Cic08] jsou definovány operace pomocí delta notace. Delta notace ukazuje všechny změny mezi danými dvěma artefakty stejné úrovně abstrakce. Změnami můžeme rozumět přidáním elementu, odstraněním elementu a modifikací elementu.

Delta notace je dle autora výhodná díky snadné rozložitelnosti velkých patchů na více menších patchů. Druhá výhoda je použitelná při paralelním vývoji. Pokud vznikne více verzí souboru v různých větvích, delta notace umožňuje snadné oddělení konfliktních operací od operací nezávislých.

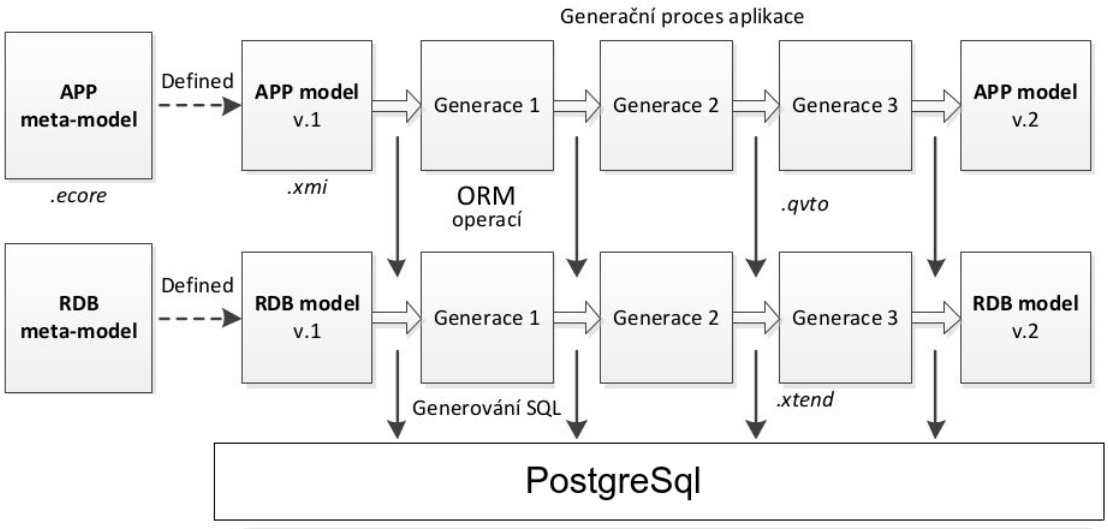
Delta notace vytvořená dle metamodelu na obr. 4.1 definuje add, delete entit a modify atributů jednotlivých entit. V našem modelu existují operace, které se nedají zařadit ani do skupiny add, delete a modify operací. Operace ExtractClass, ExtractSuperClass, InlineClass, CollapseHierarchy, PullUpProperties a PushDownProperties naopak nepatří mezi modifikační, add ani delete operace. proto jsme mohli postupovat jednou z následujících cest

- získat standardní diff dvou modelů a aplikovat na něj algoritmus rozpoznávání jakožto sadu podmínek, které musí platit, aby algoritmus rozpoznal operaci.
- vytvořit vlastní Diff metamodel, který podobně jako v 4.1 odpovídá vlastnímu seznamu operací aplikovatelých na model.

Zvolili jsme si cestu definice vlastního metamodelu.



Obrázek 4.1: Diff model převzatý dle [Cic08]



Obrázek 4.2: Seznam diff elementů

4.1.2 Rozpoznávání operací

Algoritmem pro rozpoznávání operací nazveme každý algoritmus, který nám pro každý vstupní model A a cílový model B najde uspořádaný seznam operací, jejichž postupná aplikace transformuje model A do modelu B. Tento algoritmus nemusí být deterministický.

Jedním z zajímavých faktů je poznatek, že seznam operací nemusí být jednoznačný a to i u jednoduchých změn. Pokud aplikujeme sekvenci operaci Inline A, B + Rename B ->C na model X dostaneme stejný výstup jako aplikací operací Inline B, A + Rename A->C, ještě zajímavějším poznatkem je, že nejsme schopni rozeznat rozdíl mezi aplikací sekvence operací Rename A, C + Inline C, B.

Samostatným tématem je pořadí operací a jeho permutace. Je zřejmé, že pořadí transformací v seznamu operací operujících nad hierarchiemi dědičnosti bude možné libovolně prohazovat. Také je samozřejmé, že seznam reduktivních operací stejného typu je také možné libovolně zpermutovat. Stejně tak seznam aditivních operací stejného typu. Obecný princip seřazení kolekce operací není znám.

4.1.3 Obecné principy model matching

Nejtriviálnější implementovatelný algoritmus by mohl smazat zdrojový model pomocí destruktivních operací a následně vytvořit výsledný model pomocí operací konstruktivních, případně poupravit atributy jednotlivých elementů pomocí operací modifikačních. Argumentem proti použití takového algoritmu je smazání jakýchkoliv dat, které v původní databázi byla. Takovýto algoritmus tudíže nemigruje žádná data, ale nahrazuje funkci ORM mapování integrované do většiny současných IDE. Proto se jím v této práci nezaobírám.

Jak je diskutováno v [FW14] a [DSK14] existuje několik požadavků na algoritmus řešící problém model matching. Tyto požadavky zahrnují přesnost, vysokou míru abstrakce na které je porovnávání provedeno, nezávislost na konkrétních nástrojích, doménách a jazycích, použitelnost a minimální nutnost adaptace algoritmus pro daný problém. Tyto požadavky jdou proti sobě a je nutné preferovat některé na úkor jiných, proto není možné označit za nejlepší, ale je nutné vybrat si správný algoritmus v závislosti na řešeném problému.

V [DSK14] byly popsány algoritmy pro mapování shodných entit modelů a algoritmy pro získávání rozdílu modelů. Principem těchto modelů je párování elementů vstupního modelu s elementy z modelu cílového. Autor je dělí na 4 obecné skupiny matching algoritmů.

- Párování podle statického identifikátoru páruje elementy podle perzistentního identifikátoru, který je přiřazen každé entitě v době jejího vzniku, je neměnný a unikátní. Nejzákladnějším principem model matchingu je tedy párování entit na základě shodnosti jejich identifikátorů. Tento princip má výhody jednoduchosti implementace a rychlosti. Tento algoritmus není použitelný pro modely vytvořené nezávisle jeden na druhém či u technologií nepodporujících údržba unikátních identifikátorů.

- Algoritmus signature based matching byl navržen kvůli limitaci párování podle statického identifikátoru, tento algoritmus je založen na dynamickém vypočtení nestatické signatury jednotlivých elementů pomocí uživatelem definovaných funkcí specifikovaných pomocí nějakého dotazovacího jazyka. Tento princip tedy může být použit pro modely vzniklé nezávisle na sobě. Nevýhodou je potom nutnost specifikovat query, které dopočítají signaturu.

- Algoritmus Similarity based matching používá podobně jako signature based matching podobnost sublementů jednotlivých elementů, kterou agreguje do skalární hodnoty. Tento princip se řadí mezi podtyp attribute graph [HET14] matchingu. Každá feature modelu může mít jinou váhu pro porovnávání, například u podobnosti tříd má jméno vyšší důležitost nežli abstraktnost dané třídy. Tento algoritmus musí být typicky doplněn o konfiguraci vah jednotlivých features elementů, kterou většinou píše vývojář. Zástupcem tohoto principu je framework EMF Compare, který je doplněn o defaultní konfiguraci vah. Výhodami je větší přesnost, nevýhodou je potom TRIAL ERROR (pokus omyl) metoda získávání vhodné konfigurace vah.

- Algoritmy v kategorii Custom language specific matching jsou vytvořené přímo k využití daného modelovacího jazyka. Hlavní výhodou je, že algoritmus na dané doméně může začlenit do metody similarity based matchingu sémantické detaily, což vede k přesnějším výsledkům a redukuje prohledávaný stavový prostor. Jako příklad je uváděn jazyk UMLDiff, který při porovnávání dvou UML modelů může využít faktu, že dvě třídy nebo dva datové typy stejného jména tvoří po všech praktických stránkách pár(match). Nicméně výhoda začlenění sémantických detailů konkrétní domény je vykoupeno vysokou cenou - všechny ostatní kategorie algoritmů potřebují minimální neb téměř žádné úpravy od vývojáře, pro tuto kategorii vývojáře musí napsat celý matchovací algoritmus sám.

4.1.4 Graph matching

Problém model matching je podproblémem generičtějšího problému graph matching, který studuje [Ben02] a rozděluje a popisuje algoritmy pro graph matching - algoritmus nalezení shody grafů. Problém je definován na obecné struktuře Graf, což je uspořádaná dvojice G = (V, E), kde G je množina uzlů a E je množina hran grafu, přičemž $E \subset V \times V$. Grafy mohou být orientované či neorientované, mohou mít vícenásobné hrany.

Každý graf může přidávat informace do své struktury pomocí labelu (popisku nebo čísla) do hran a vrcholů, pokud je nutné přidat více informací, je možné přidat do hran a/nebo vrcholů atributy, potom hovoříme o vertex-atributed grafech a edge atributed grafech, případně attributed grafech. V některé literatuře jsou attributed grafy označovány jako labeled grafy. Graph matching je aplikován v mnoho oborech jako je počítačové vidění, analýza scény (scene analysis), chemie a molekulární biologie. Pro tyto obory je esenciálním nalézt vzorky nalezeny v daných datech.

Problém shodnosti dvou grafů G_O (grafu originálu) a G_V (grafu vzorku), se dělí na algoritmus nalezení přesné shody vzorku v hledaném grafu či algoritmus hledání podobnosti grafu vzorku v hledaném grafu jak je zobrazeno na obr. 4.3 převzatém z [Ben02].

Algoritmus hledání přesné shody je definován následně: Mějme grafy $G_O = (V_O, E_O)$ a $G_V = (V_V, E_V)$, přičemž $\|V_O\| = \|V_V\|$, úkolem je potom najít takové prosté zobrazení $f : V_O \to V_V$, takové, že $(u, v) \in E_O$ iff $(f(u), f(v)) \in E_V$. Pokud takové mapování existuje, nazveme ho přesnou shodou.

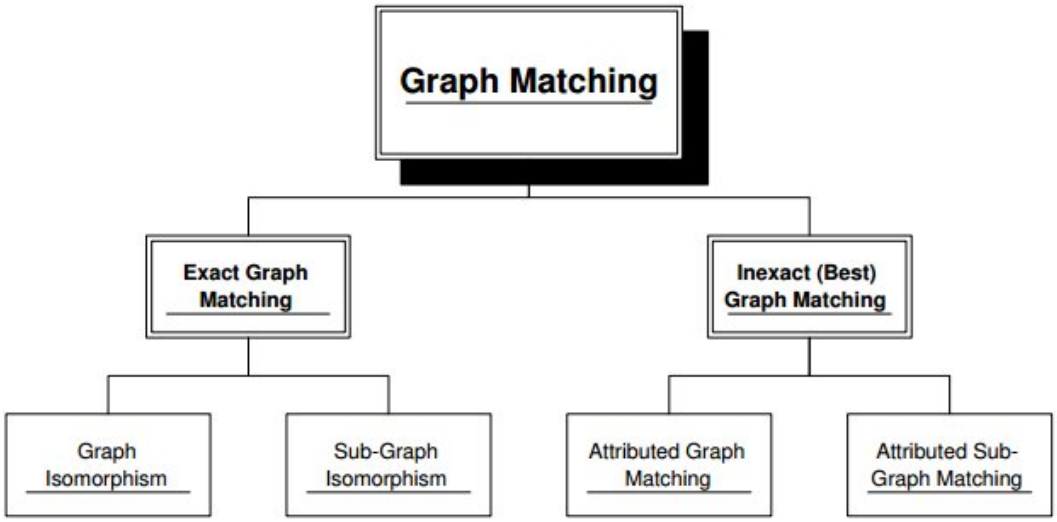
Termín Hledání podobnosti grafů (inexact macthing) aplikovaný na některé problémy týkající se shodnosti grafů vyjadřuje, že není možné nalézt izomorfismus mezi dvěma grafy, aby byly shodné. V těchto případech mají grafy rozdílné charakteristiky jako je například jiný

počet vrcholů, jiný počet hran či jiná délka nejdelší kružnice. Tedy není hledán izomorfismus dvou grafů, ale problém je upraven na hledání největší možné shody mezi grafy. Tato transformace mění problém a zařazuje ho do třídy problémů známé jako inexact graph matching. V takovém případě hledáme nebijektivní korespondenci mezi grafem vzorku a grafem originálu. V následujícím textu předpokládejme $\|V_V\| < \|V_O\|$. Inexact matching je používán v oborech kartografie, rozpoznávání znaků a medicíně. Nejlepší korespondence graph matching problému je definována jako optimum nějaké objektivní funkce, která měří podobnost mezi přiřazenými uzly a hranami. Tato funkce je nazvána fitness funkcí, případně funkcí energie.

Formálně je tedy hledání podobnosti grafů definováno takto: mějme dva grafy, $G_V a G_O$ přičemž $\|V_V\| < \|V_O\|$ a cílem je nalezení mapování $f' : V_D \to V_M e(u, v) \in E_P$ iff $(f(u), f(v)) \in E_M$.

Podtypem těchto úloh jsou problémy subgraph matching a subgraph izomorfizmu.

Při hledání složitost uváděných problémů autor [Ben02] řadí hledání přesné shody do P až NP kompletní množiny úloh, přičemž že u problémů této kategorie nebyla dokázána nejvyšší složitost NP complete. Pro problémy subgraph isomorphismu bylo dokázáno, že patří do třídy NP complete. Pro složitost nepřesného graph matchingu bylo dokázáno, že patří do třídy NP-complete.



Obrázek 4.3: Typy graph matchingu

Kapitola 5

Vytvořené algoritmy rozpoznávání operací

5.1 Stavový algoritmus

V ranné fázi jsem napsal prototyp rozpoznávacího algoritmu, který se snaží minimalizovat vzdálenost současného modelu od modelu cílového pomocí rozpoznání operací a jejich následné aplikace. Výhodou tohoto přístupu je možnost nalezení více alternativních cest, nevýhodou je potom velikost stavového prostoru zvětšující se s počtem rozpoznávaných operací. Algoritmus je popsaný v pseudokódu viz algoritmus 2

Algoritmus v prvním svém kroku inicialializuje hodnoty. Aktuálnímu modelu nastaví hodnotu modelu zdrojového. Množinu rozpoznaných operací R_{ops} inicializuje prázdnou množinou. Rozpoznanou R operaci nastaví na NULL.

Následně opakuje cyklus repeat začínající na řádce 5, který je ukončen splněním terminální podmínky uvedené v pseudokódu na řádce 20, R = NULL. Tj. algoritmus je ukončen, pokud v těle cyklu nebyla rozpoznána žádná operace.

V těle cyklu Repeat potom algoritmus nastaví hodnotu R na NULL a nastaví na zmíněném řádku 7 hodnotu aktuálního nejlepšího zlepšujícího kroku K na 0, aby zajistil přemazání dat nalezené operace v minulé iteraci cyklu.

Algoritmus spočítá v cyklu na řádcích 8-8 pro každou operaci maximální zlepšující krok této operace K_{op}. Pokud je K_{op} lepší než dosavadní maximum K, aktualizuje K_{op} a R. Následně algoritmus v případě rozpoznání nějaké operace - viz. řádek 15 nalezne parametry params pro operaci R viz metoda getParams(R, A, S) na řádku 16. Tato metoda by například pro operaci AddClass nalezla jméno třídy, která neexistuje v zdrojovém modelu, ale existuje v cílovém modelu. Algoritmus přidá operaci R s aplikovanými(params) do množiny R_{ops} a aplikuje operaci R(params) na aktuální model A.

Základními myšlenkou pro vytvoření algoritmu byla existence vzdálenosti dvou modelů. Vzdálenost dvou modelů je snižována v každém kroku o zlepšující krok a jakmile jsou dva modely shodné (či pro danou množinu operací R_{ops} velmi podobné), algoritmus již žádnou další operaci nerozezná, hodnota vzdálenosti je na minimu a algoritmus končí. Vzdálenost modelu od prázdného modelu nazveme energií modelu.

Algorithm 1 Algoritmus procházení stavů

Input: Zdrojový model S, cílový model T
Output: Seznam operací R_{ops}, po jejichž aplikaci se model S tranformuje na model T

Inicializace:
$A \leftarrow S$ ▷ Aktuální model
$R_{ops} \leftarrow \{\}$
$R \leftarrow NULL$ ▷ Rozpoznaná operace

repeat
    $R \leftarrow NULL$
    $K \leftarrow 0$ ▷ Zlepšující krok
    for all op from Ops **do** ▷ Pro každou operací z app metamodelu
        $K_{op} \leftarrow getImprovement(op, A, T)$ ▷ Spočítá zlepšení
        if $K < K_{op}$ **then** ▷
            $K \leftarrow K_{op}$
            $R \leftarrow op$
        end if
    end for
    if $R \neq NULL$ **then**
        $params \leftarrow getParams(R, A, S)$
        $R_{ops} \leftarrow R_{ops} \bigcup R(params)$
        apply(R(params), A)
    end if
until $R = NULL$ ▷ Opakuje cyklus, dokud byla rozpoznána nějaká operace

5.1.1 Energie modelu

Pokud se zaměříme na samotnou existenci tříd s daným jménem a properties s daným jménem v modelu a pomineme ostatní atributy properties a tříd můžeme si představit energii modelu jako součet existujících tříd a jejich properties.

Pro jednoduchost si můžeme reprezentovat textově s následujícími pravidly: název třídy budeme reprezentovat velkými písmeny a názvy properties malými písmeny. V zájmu jednoduchosti nereprezentujeme idProperties, která má většinou název jednoznačně odvoditelný od názvu třídy, která ji vlastní. Třída je řetězec obsahující svůj název a seznam properties. Řetězec Abcd tedy reprezentuje třídu A s properties "b","c"a "d". Řetězec Bef potom reprezentuje třídu B obsahující property "e"a "f". Tuto reprezentaci nazveme Jednoduchovou textovou reprezentací. Model M_1(Abcd, Bef) obsahuje dříve zmíněné třídy. Jeho energie je rovna součtu energie třídy A a třídy B. Energie každé třídy je rovna 1 + sumy energií jednotlivých properties. Properties mají v naší demonstrativní ukázce bez započítání jiných atributů než name energii 1.

$$E(M_1) = E(A) + E(B) = 1 + \sum_{p \in A} E(p) + 1 + \sum_{p \in B} (E(p)) = 1 + 3 + 1 + 2 = 7$$

Energie je v tomto případě rovna počtu konstruktivních operací nutných k vytvoření tohoto modelu. Energie také vyjadřuje v tomto případě počet destruktivních operacích nutných k smazání tohoto modelu. Pokud vytvoříme další model, můžeme si vyjádřit hodnotu změny těchto modelů.

Můžeme vytvořit model M_2 například tak, že z modelu M_1 odebereme z třídy A property "c", přejmenujeme property "d"na "g"a přejmenujeme třídu B na C. Tak získáme model Model M_2(Abg, Cef), který má energii rovnu hodnotě 6. Patch(M_1, M_2) můžeme definovat pomocí delta notace. Naší delta notaci můžeme zapsat ve tvaru +X pro přidanou třídu, +Xyz pro přidané atributy "y"a "z"ze třídy X, -X pro odstraněnou třídu a -Xyz pro odstraněné atributy "y"a "z"ze třídy X.

Listing 5.1: Textový diff modelů

```
+Ag          −Acd
+C           −B
+Cef         −Bef
```

Pro tento minimalistický přístup můžeme použít výpočet symetrické delty následující vzorce:

$$distance(M_1, M_2) = E(\Delta(M_1, M_2)) + E(\Delta(M_2, M_1))$$

Vzdálenost dvou modelů M_1, M_2 je potom rovna energii, množinového rozdílu M_1, M_2, sečtené s energií množinového rozdílu M_2, M_1.

$$\Delta(M_X, M_Y) = (\Delta C_{XY}, \Delta P_{XY})$$

Delta modelů X a Y je uspořádaná dvojice rozdílu tříd a rozdílu properties.

$$\Delta C_{XY} = \{c : c \in M_X \wedge c \notin M_Y\}$$

Delta tříd modelů X a Y je množina tříd, které jsou v modelu X, ale nejsou v modelu Y.

$$\Delta P_{XY} = \{p : p \in X \wedge p \notin Y \wedge p.owner \notin \Delta C_{XY}\})$$

Delta properties modelů X a Y je množina properties, které patří do modelu X, ale nejsou v modelu Y a jejich vlastnická třída není obsažena v množině ΔC_{XY}. Explicitní podmínka pro vlastnickou třídu property je, že je obsažen v X, protože i p je obsaženo v X.

Předpokládejme dále, že energie dvojice $E(C_{XY}, P_{XY}) = E(C_{XY}) + E(P_{XY})$

Původní rovnice se nám tedy přepíše na :

$$\begin{aligned} distance(M_X, M_Y) =& E(\Delta(M_X, M_Y)) + E(\Delta(M_Y, M_X)) = E(C_{XY}, P_{XY}) + E(C_{YX}, P_{YX}) = \\ =& E(C_{XY}) + E(P_{XY}) + E(C_{YX}) + E(P_{YX}) \end{aligned}$$

Na listing 5.1 máme na levé straně zobrazenou $\Delta(M_1, M_2)$ a na pravé straně $\Delta(M_2, M_1)$ pro zadaný příklad. Pro naše konkrétní modely $M_1 a M_2$ je distance tedy:

$distance(M_1, M_2) = E(C_{12}) + E_{P_{12}} + E(C_{12}) + E_{P_{12}} = 1 + 3 + 1 + 4 = 9$

Tento způsob vypočítávání Energie nicméně zanedbává energii, o kterou model obohacují atributy tříd a properties. Atributy properties a tříd dělíme do skupiny primitivních typů a skupiny referencí. Mezi primitivní atributy řadíme třídní atributy name a isAbstract a atributy lowerBound, upperBound, isOrdered, isUnique a name vlastněné properties. Mezi referenční atributy patří třídní atribut parent a atributy type a oppositteProperty vlastněné property. Zásadním rozdílem mezi skupinou referenčních a primitivních atributů je způsob jejich změny jednotlivými operacemi. Primitivní atribut můžeme změnit aplikací jedné operace na jinou hodnotu. Například property atribut isUnique změníme aplikací operace SetUnique. Výjimečné jsou atributy lowerBound a upperBound, jejichž hodnotu můžeme změnit současně jednou operací SetBounds. Hodnotu referenčních atributů nemůžeme změnit jednou operací, ale dvěma. Například na změnu parent třídy musíme rodičovskou třídu nejdříve odstranit operací RemoveParent a následně operací AddParent přidat referenci na novou rodičovskou třídu. Změna referenčních hodnot modelů je započítána dvakrát oproti změně hodnot primitivních atributů. Primitivní hodnoty je nutné do vzorce tedy započítat. Původní vzorec pro vzdálenost dvou modelů se tedy změní na:

$$\begin{aligned} distance(M_X, M_Y) =& E(\Delta(M_X, M_Y)) + E(\Delta(M_Y, M_X)) = E(C_{XY}, P_{XY}) + E(C_{YX}, P_{YX}) + \\ &+ E(PRIM_{XY}) = E(C_{XY}) + E(P_{XY}) + E(C_{YX}) + E(P_{YX}) + \\ &+ E(PRIM_{XY}) \end{aligned}$$

Přičemž $E(PRIM_{XY})$ reprezentuje energii změněných primitivních atributů a gamma reprezentuje podmínku pro započítaný atribut:

$$\gamma = A : (A.owner \notin M_X \vee A.owner \notin M_Y) \vee A_X ! = A_Y$$

$$E(PRIM_{XY}) = \sum_{\gamma} E(A)$$

Pro rozšířený výpočet energie property $p_{XY} \in P_{XY}$ je zakomponována do vzorce energie referenčních propertie:

$$E(p_{XY}) = 1 + \sum_{A \in p_X \wedge A_Y = null \vee A_Y ! = A_X}$$

Takto definovaný výpočet energie s položením $E(A) = 1$ udává maximální počet konstruktivních, destruktivních a modifikačních operací nutných k transformaci modelu M_X na model M_Y.

Zlepšující krok operace je definován jako vzdálenost, o kterou se přiblíží aktuální model A po aplikaci operace op k modelu cílovému T s nejlepšími možnými parametry. Nejlepší parametry jsou takové, které přibližují aktuální model o takovou vzdálenost, že neexistují parametry, které by zdrojový model přibližovaly k cílovému o vzdálenost větší. Není zaručena jedinečnost nejlepších parametrů. Při neexistenci jakýchkoliv vhodných parametrů vrací operace $-\infty$. Pro zlepšující krok getImprovement(op, A, T) platí:

$$distance(M_A, M_T) = distance(M_X, M_T) + getImprovement(op, A, T)$$

Ilustrujme běh algoritmu na příkladě. Na obrázku 5.1 je zobrazen ukázkový zdrojový a cílový model je zobrazený na obrázku 5.2.

Obrázek 5.1: Počáteční model

Obrázek 5.2: Cílový model

Algoritmus předpokládá, že operace přibližující aktuální stav k cílovému by měly být rozpoznány (a aplikovány) co nejdříve. Například pro počáteční model zobrazený na obrázku 5.1 a cílový model zobrazený na obrázku 5.2. Pokud použijeme základní definici vzdálenosti energie, můžeme počáteční model vyjádřit pomocí jednoduché textové formy se zkrácením názvů na počáteční písmeno a dolním indexem pro property s neunikátním názvem "M_1(Cscp)"a cílový model jako $M_2(Cpc_{on}, Acsa)$ vzdálenost těchto dvou modelů vyjádřit pomocí delta notace zobrazené v seznamu 5.2.

Listing 5.2: Textový patch modelů pro ukázkový příklad v delta notaci

```
+Cc_on   −Csc
+A
```

+Asca

$$distance(M_1, M_2) = E(C_{12}) + E_{P_{12}} + E(C_{12}) + E_{P_{12}} = 0 + 2 + 1 + 4 = 7$$

Algoritmus v prvním průchodu rozpozná, že je možné aplikovat operace AddClass, RemoveProperty a ExtractClass. Nejlepším parametrem pro AddClass je třída Address. AddClass zlepšuje vzdálenost o 1. Nejlepším parametrem pro operaci RemoveProperty je jedna z odstraněných property, takže například Street. RemoveProperty zlepšuje vzdálenost také o 1. Vhodnými parametry pro operaci ExtractClass jsou třídy zdrojová třída Address, extrahovaná třída Contact, referenční property contact, její oppositum address a nějaký seznam properties obsažených v třídě address. Nejlepšími parametry pro ExtractClass jsou dvě zmiňované třídy s asociačními properties a seznam všech properies v nově vzniklé třídě Contact. ExtractClass má hodnotu zlepšujícího kroku 7. Z těchto operací vyhodnotí jako nejlepší ExtractClass a následně algoritmus skončí, protože po aplikaci se současný model dostal do cílového stavu a již nerozpozná další operaci.

Algoritmus dává stejnou váhu atributům jako třídám, což nemusí být správné. Aby algoritmus lépe splňoval požadavky uživatele, tj. priorizoval některé operace, je možné modifikovat výpočet energie a místo konstanty 1 v předešlých vzorcích použít konfiguraci vah jednotlivých atributů. Všechny atributy jsou vyjmenovány v seznamu 5.1.1.

- Váhy atributů třídy

  W(classname) jméno třídy

  W(parent) reference na rodičovskou třídu

- Váhy atributů všech property

  W(propertyName) jméno property

  W(type) reference na typ property

  W(isOrdered) boolean značící ordered kolekci

  W(isUnique) boolean značící unikátní kolekci

  W(LowerBound, UpperBound) horní a dolní mez hodnot property

- Váhy atributů asociačních property

  W(oppositteProperty) reference na opozitní property u bidirectional vazby

Příklad konfigurace:

- classname = 1
- parent = 0
- propertyName = 1
- type = 0

- isOrdered = 0
- isUnique = 0
- LowerBound + UpperBound = 0
- oppositteProperty = 0

Tato konfigurace redukuje získávání energie na původně zmiňovaný způsob definice Energie.

Další možnou modifikací je úprava seznamu R_{ops} použitého v algoritmu na řádku 8. Algoritmus je schopný rozpoznat jednotlivé operace, ale může mít problémy s rozpoznáním některých dvojic operací. Problematickým stavem je například, pokud jediným rozdílem je primitivní typ property odlišný ve vstupním a výstupním modelu. Algoritmus vyhodnotí vzdálenost > 0 a musí buď aplikovat operaci RemoveProperty, která smaže property a má záporný zlepšovací krok. Bylo by nutné upravit zisk zlepšujícího kroku, značně zesložitit implementaci metody getImprovement pro operaci RemoveParent, upravit inicializaci zlepšujícího kroku tak, aby se na řádku 7 nainicializovala hodnotou $-\infty$ a upravit terminální condition na řádku 20, tak, aby algoritmus neběžel do nekonečna. Druhou možností vyřešení tohoto problému je definovat seznam R_{ops} nikoliv jako seznam operací v aplikačním metamodelu, ale jako seznam sekvencí operací. Konverze původního seznamu operací na jednoprvkové sekvence a rozšíření seznamu R_{ops} o sekvenci operací RemoveProperty a AddProperty, s metodou getImprovement získávání zlepšujícího kroku pouze pro případy, kdy existuje property, která má v výstupním modelu jiný typ než ve vstupním.

Přes počáteční slibné výsledky testované na konstruktivních operacích nebyl tento algoritmus shledán jako příliš efektivní. První nevýhodou je, že algoritmus v každém kroku hledá nejlepší zlepšující krok i pro všechny operace nalezené v iteraci minulé. Rozpoznané operace z minulého kroku by bylo možné zapamatovat, ale nebyl nalezen způsob hledání konfliktních operací vyjma případů, kdy dvě operace pracují nad jinou hierarchickou strukturou. Proto se nedá jednoduše analyzovat, pro které operace je nutné hledat nový zlepšující krok znovu, kterých operací se aplikace rozpoznané operace nedotkla, a které operace již nemusí mít zlepšující krok shodný a mohou mít jiné parametry.

Druhou nevýhodou je šířka prohledávaného stromu - algoritmus testuje v každé iteraci všechny operace, jejichž seznam (či v modifikaci zmiňovaná sekvence operací) R_{ops} může nabývat velkých hodnot. Při všech váhách nastavených na jedna při vzdálenosti dvou modelů distance $distance(M_1, M_2) = d$, velikosti $R_{ops} = n$, složitosti aplikace operací op_{appply}, složitosti zisku zlepšujícího kroku operace $op_{getImprovement}$ a složitosti zisku parametrů $op_{getParams}$ má algoritmus algoritmickou složitost $O(d) = O((Max(op_{getImprovement})^n * op_{getParams} * op_{appply})^d)$. V každé z maximálně d iterací je nutné v nejhorším případě získat zlepšující krok pro každou operaci a pro operaci s nejvyšším zlepšujícím krokem potom nalézt pro tuto operaci nejlepší parametry a aplikovat ji. Složitost operace $op_{getImprovement}$ se různí v závislosti na typu operace. Pro c tříd ve vstupním modelu a výstupním modelu má složitost hledání parametrů operace AddClass $O(AddClass_{getImprovement}) = c^2$ a, protože hledáme třídu, která není ve vstupním modelu, ale je ve výstupním, tedy procházíme c tříd ve vstupním modelu a c tříd v modelu výstupním. Složitost $O(ExtractClass_{getImprovement}) = c^2 * p * r * p$ příčemž c je počet tříd ve vstupním a výstupním modelu, p je maximální počet operací ve třídě a r je počet extrahovaných property. Hledáme třídu, ve které existuje asociační property v cílovém modelu, která neexistovala v modelu zdrojovém, jejíž součet energie

properties, které jsou "extrahovány"do třídy typu asociace je největší. Po položení r = p dostáváme $O(ExtractClass_{getImprovement}) = c^2 * p^3$. Vzhledem k standardním modelům, které obsahují hodně malých tříd můžeme za předpokladu $c >> p$ položit $O(op_{getImprovement}) = c^2$. U většiny operací hledáme třídu ze vstupního modelu, ke které hledáme její protějšek ve výstupním modelu se zadanými vlastnostmi, proto berme $O(op_{getImprovement}) = c^2$. Stejnou složitost získáme ekvivalentním postupem pro funkci getParams. Funkce $op_{appply})^d)$ hledá jen třídu z vstupního modelu, na které provede nějaké změny, takže můžeme aproximovat její algoritmickou složitost hodnotou c.

$O((Max(op_{getImprovement})^n * op_{getParams} * op_{appply})^d) = O(c^2 * c^2 * c)^d = O(c^{5*d})$

Stavový algoritmus měl definován krok pro operace AddPrimitiveClass, AddStandardClass, RenameEntity, RemoveEntity, RenameProperty.

5.2 Návrh ze studia článků

Kvůli přílišné obecnosti algoritmů pro graph matching nebyly tyto algoritmy shledány za vhodné k použití pro problém hledání sady aplikačních operací. První 3 popsané algoritmy model matchingu (1 párování podle statického identifikátoru, 2. signature based matching, 3. similarity based matching) nejsou vhodné k použití z důdodu, že k rozpoznání popsaných expanzivních a reduktivních operací je nutné rozpoznat 2 třídy, které se mapují na jednu třídu pro reduktivní operace a naopak jednu operaci, která se mapuje na 2 třídy. Problém rozpoznávání operací je tudíž nadskupinou problému model matchingu, protože matching páruje 1 ku 1, ale algoritmus řešící problém rozpoznávání operací musí řešit matching M entit ku N entitám.

Zmiňované algoritmy mě inspirovaly k vytvoření Custom language specific matching algoritmu pro tento problém, který si z zmiňovaných algoritmů v sekci 4.1.3 bere hlavně poznámku u algoritmu UMLDiff zmiňovaného pod Custom Language skupinou 4.1.3 - ze všech praktických důvodů považujeme třídy se stejným jménem jako shodné.

Vznikly dvě implementace párovacích algoritmů.

5.3 Základní implementace

První jednodušší používá základu z UMLDiffu a očekává, že dvě třídy se stejným jménem jsou shodné, páruje tedy třídy podle jména. Následné rozdíly řeší rozpoznáním konstruktivních a destruktivních, případně některých operací modifikačních, ať už tyto operace pracovali s třídami nebo s property.

První algoritmus pracuje podle tohoto pseudokódu:

1. Pro každou třídu S z vstupního modelu, která má stejné jméno jako třída T obsažená v modelu výstupním
   (a) Pro každou property P_S z S, která má stejné jméno jako P_T Property z T
      i. odstraň P_S z SProps
      ii. odstraň P_T z TProps
   (b) odstraň S z SClses
   (c) odstraň T z TClses

Algorithm 2 Algoritmus procházení stavů

Input: Zdrojový model S, cílový model T
Output: Seznam operací R_{ops}, po jejichž aplikaci se model S tranformuje na model T

1: **Inicializace:**
2: $SClses \leftarrow S.classes$ ▷ Nenamatchované třídy z vstupního modelu
3: $TClses \leftarrow T.classes$ ▷ Nenamatchované třídy z cílového modelu
4: $SProps \leftarrow S.classes.properties$ ▷ Nenamatchované properties z vstupního modelu
5: $TProps \leftarrow T.classes.properties$ ▷ Nenamatchované properties z cílového modelu
6: $OPS \leftarrow \{\}$
7: **for all** $C_S \in S : \exists C_T \in T\ C_T.name = C_S.name$ **do** ▷ b
8: **end for**

2. Výsledný seznam SClasses transformuj na operace RemoveCls a přidej do seznamu Ops

3. Výsledný seznam TClasses transformuj na operace AddCls a přidej do seznamu Ops

4. Výsledný seznam SProps transformuj na operace RemoveProperty a přidej do seznamu Ops

5. Výsledný seznam TProps transformuj na operace AddProperty a přidej do seznamu Ops

6. Pro případně změněné atributy (isAbstract, parent, isUnique, isOrdered . . .) přidej do seznamu Ops dané modifikační operace

Tento algoritmus má schopnost rozeznat jen konstruktivní a destruktivní operace. Tudíž zachovvává data, jejichž identifikátor se nemění. Ostatní data naopak nekompromistně maže. Tento algoritmus není schopný rozpoznat operace RenameClass, RenameProperty, které jsou dle [Luk13] nejčastěji používanými operacemi. Výhodou tohoto algoritmu je, že nepotřebuje definovat jiný diff nežli delta notaci.

5.4 Složitější implementace

Složitější implementace algoritmu páruje stejně jako jednodušší v první fázi shodné elementy - modely se mění, ale některé třídy jsou zachovány. Shodné elementy potom tvoří jakési pilíře pro operace konstruktivní a destruktivní, které se vážou na rozpoznané páry. Závislost rozpoznání

Algoritmus rozpoznávání operací byl napsán se snahou o zachovávání co největšího množství dat. Ačkoliv triviální algoritmus pro přechod z modelu A k modelu B by mohl pomocí subtraktivních operací zničit model A a následně složit model B pomocí aditivních operací je zřejmé, že tento algoritmus neuchová žádná data. Proto byly zavedeny Konstruktivní a destruktivní operace.

5.4.1 Diff elementy

Kvůli nutnosti rozpoznávat operace vznikly v aplikačním metamodelu nové elementy. Kořenovým elementem diff modelu je Diff element. Tento element obsahuje kolekce elementů classpairs typů ClassPair, propertyPairs typu PropertyPair, a dále pak addedClasses a removedClasses typu DiffClass a addedProperties a removedProperties typu DiffProperty. Element ClassPair shlukuje zpárované zdrojové (source) a obrazové (reflection) třídy, dále pak referenci owningDiff na Diff element, v kterém jsou obsaženy a která je důležitá pro implementaci algoritmu a v neposlední řadě underlyingPairs - shodné páry Properties typu EqualPropertyPair, které jsou detekované danou operací. Podobně jako operace jsou i páry rozděleny do několika skupin.

Konstruktivní a destruktivní operace nemají svůj obraz v diff metamodelu jako ClassPair ani PropertyPair, protože tyto operace nemapují element ze vstupního modelu na element z výstupního modelu. Konstruktivní operace by jinak mapovaly prázdný vstup na element a destruktivní obráceně element na prázdný výstup.

Oproti jednodušším konstruktivním a destruktivním operacím jsou operace expanzívní, konstruktivní a modifikační v Diff modelu zobrazeny jako ExpansiveClassPair, ReductiveClassPair a ModifyingClassPair, mapují element vstupního modelu na element cílového modelu. Aby bylo možné rozpoznat specifický pár závislý na jiném páru, byla přidána třída ReplacingClassPair - nahrazující pár, který se používá jako pivot pro hledání expansivních, reduktivních a modifikačních párů. Od elementu ReplacingClassPair dědí elementy EqualClassPair - třída, která si uchovala jméno z původního modelu a element ReplacingClassPair - reprezentující třídu, která si neuchovala jméno, ale má změněný název. Podmínky získávání konkrétních typů párů a jejich pořadí specifikuje konkrétní rozpoznávací algoritmus.

Projevem subtraktivních a aditivních operací jsou elementy DiffClass a DiffProperty, které zaobalují třídy a property tak, aby bylo možné referencovat na jiný objekt než element Structure .

Kapitola 6

Testování projektu Migdb

V průběhu vývoje byl za účelem ověření správné funkcionality vytvořen projekt Migdb.testing.run, který spouští jednotlivé testy kompoment aplikace.

V rámci projektu byly vytvořeny testy komponent Workflow obsažené v packagi migdb.testing.components.run, testy aplikační evoluce inkludované do Workflow test_app_atomic.mwe2 v packagi migdb.testing.app.atomic.run, testy databázové evoluce obsažené workflow test_rdb_atomic.mw v packagi migdb.testing.rdb.atomic.run, testy validátorů aplikačního a databázového modelu obsažené v packagi migdb.testing.validators.run, test ORM transformace struktury aplikace na DB schema + ORMo transformace obsažené v workflow migdb.testing.orm.run, testy generování SQL schématu obsažené v packagi migdb.testing.generators.run, integrované testy celého frameworku migdb obsažené v packagi migdb.testing.migdb_executer a posledními testy jsou testy algoritmů rozpoznávajících operace obsažené v packagi migdb.testing.app.oracle.run.

Testy komponent testují správnou funkcionalitu komponenty Comparator, která musí správně porovnat očekávaný a reálný výstupní soubor xmi ostatních testů. Tyto testy jsou rozděleny do více workflow, protože některé musí být neúspěšné, jak už napovídá klíčové slovo fail v jejich názvu. Dalšími komponentami vzniklými v rámci projektu Migdb byly QVTO-Executor, TestWorkflow, DirectoryCleaner a TestComponent. Komponenta TestComponent vznikla, aby bylo přehlednější a efektivnější psát QVT testy Migdb, je složena z několika dalších načítacích, ukládacích a porovnávacích komponent a zkracuje zápis testů ve workflow asi 10 krát. Nebylo jasné, jak vytvořit testy na správnou funkcionalitu ostatních komponent - správné jejich zapojení do jiných testů bylo pro nás dostatečným testem.

Ostatní testy se řadí do xmi kategorie testů, tj testů, které mají jako vstupní model xmi soubor. Po spuštění xmi testů se v projektu migdb.testing.run vygeneruje složka output-tests, která obsahuje výstupní data jednotlivých testů. Výstupní data mohou obsahovat výstupní xmi soubory porovnávané s očekávaným xmi souborem nebo SQL souborem, který je možné aplikovat na databázi PostgreSQL.

Speciálním testem je test_code_generator.mwe2, tento test vygeneruje pro zadaný vstupní aplikační model a sadu aplikačních operací výstupní rdb model, sadu databázových operací, výstupní SQL generující strukturu databáze a SQL reprezentující výstupní databázové operace. Kromě těchto vygenerovaných dat je možné najít ke každému testu v adresáři test_data reálná data, s kterými může být test spuštěn. Reálná data jsou vytvořena jen pro testy komplexnějších operací (operace 007 - 010), které jen nemění strukturu databáze, ale i manipulují s daty. Postup testování tohoto testu je - spuštění workflow vedoucí k vygenerování SQL db

schematu a SQL db operací. Spuštění SQL vytvářející strukturu databáze, spuštění SQL s daty uloženými v souboru data.sql z přidružené složky v adresáři test_data, aplikace vygenerovaného souboru transformačních SQL, zobrazení výstupních dat za pomoci souboru check_selects.sql ze složky s daty. Bohužel nebyl nalezen lepší způsob otestování než manuální spuštění a vizuální porovnáných selectů s očekávaným výstupem po apliaci dané operace na data.sql.

Kapitola 7

Ukázka zdrojového kódu práce

Ukázka helperu mergeProperty

```
helper AssociationProperty::ormMergeProperty(appStructure : APP::Structure,
     targetClassName : String, operations : RDB::Operations) : RDB::Operations {
   var defaultSchemaName : String := getDefaultSchemaName();
   var sourceClassName : String := self.owningGeneralClass.name;
   var typeClassName : String := self.type.name;
   var sourceTableName : String := sourceClassName.translate();
   var targetTableName : String := targetClassName.translate();
   var typeTableName : String := typeClassName.translate();
   var mergedColumnName : String := self.name.translate();
   var idSourceTableName : String := getDbIdColumnName(sourceTableName);
   var idTargetTableName : String := getDbIdColumnName(targetTableName);
   var idTypeTableName : String := getDbIdColumnName(typeTableName);

   if(self.upperBound = 1)then{
      var equalWhereCondition : String := getEqualityWhereCondition(
                                                      sourceTableName,
                                                      idSourceTableName,
                                                      targetTableName,
                                                      idTargetTableName);
      var safeUpdateCondition : String := getSafeCondition(
                                                     defaultSchemaName,
                                                     idTargetTableName,
                                                     sourceTableName,
                                                     idSourceTableName);
      var updateRows : RDB::ops::ModelOperation := _updateRows(
                                                          defaultSchemaName,
                                                          sourceTableName,
                                                          mergedColumnName,
                                                          targetTableName,
                                                          mergedColumnName,
                                                          equalWhereCondition,
```

```
                                                        safeUpdateCondition);
addOperation(updateRows, operations);
    var fkName : String := getFkRefencingOppositeName(
                                                self.name,
                                                sourceClassName,
                                                typeClassName);
    var removeFkConstr : RDB::ops::ModelOperation := _removeConstraint(
                                                            defaultSchemaName,
                                                            sourceTableName,
                                                            fkName);
    addOperation(removeFkConstr, operations);
    var removeMergedColumn : RDB::ops::ModelOperation := _removeColumn(
                                                            defaultSchemaName,
                                                            sourceTableName,
                                                            mergedColumnName);
   addOperation(removeMergedColumn, operations);
   //M x N association
   }else{
       var sourceAssociationTableName : String := getAssociationTableName(
                                                             self.name,
                                                             sourceClassName);
       var targetAssociationTableName : String := getAssociationTableName(
                                                               self.name,
                                                               targetClassName);
       var sourceColumnNames : OrderedSet(String) := OrderedSet{
                                                        idSourceTableName,
                                                        idTypeTableName};
       var destinationColumnNames : OrderedSet(String) := OrderedSet{
                                                           idTargetTableName,
                                                           idTypeTableName};
       if(self.isOrdered)then{
          sourceColumnNames += getDbOrderingColumnName();
          destinationColumnNames += getDbOrderingColumnName();
       }endif;
       var transferData : RDB::ops::ModelOperation := _insertRows(
                                                        defaultSchemaName,
                                                        sourceAssociationTableName,
                                                        sourceColumnNames,
                                                        targetAssociationTableName,
                                                        destinationColumnNames);
       addOperation(transferData, operations);
       if(self.isOrdered)then{
          var ordName : String := getUXOrderingName(sourceAssociationTableName);
          var removeOrdConst : RDB::ops::ModelOperation :=
                                                  _removeConstraint(
                                                         defaultSchemaName,
```

```
                                                    sourceAssociationTableName,
                                                    ordName);
   addOperation(removeOrdConst, operations);
   var removeOrdCol : RDB::ops::ModelOperation :=
                                              _removeColumn(
                                                  defaultSchemaName,
                                                  sourceAssociationTableName,
                                                  getDbOrderingColumnName());
   addOperation(removeOrdCol, operations);
}endif;
if(self.isUnique)then{
   var uxName : String := getUXName(sourceClassName, self.name);
   var removeUxConst : RDB::ops::ModelOperation :=
                                               _removeConstraint(
                                                   defaultSchemaName,
                                                   sourceAssociationTableName,
                                                   uxName);
   addOperation(removeUxConst, operations);
}endif;
var fkSourceName : String :=
                    getFKAssociationTableRefName(
                                sourceAssociationTableName,
                                sourceTableName);
var removeFkSourceConst : RDB::ops::ModelOperation :=
                                          _removeConstraint(
                                               defaultSchemaName,
                                               sourceAssociationTableName,
                                               fkSourceName);
addOperation(removeFkSourceConst, operations);
var fkTypeName : String :=
                    getFKAssociationTableRefName(
                                             sourceAssociationTableName,
                                             typeTableName);
var removeFkTypeConst : RDB::ops::ModelOperation :=
                                        _removeConstraint(
                                             defaultSchemaName,
                                             sourceAssociationTableName,
                                             fkTypeName);
addOperation(removeFkTypeConst, operations);
var removeIdSourceRefCol : RDB::ops::ModelOperation :=
                                           _removeColumn(
                                                defaultSchemaName,
                                                sourceAssociationTableName,
                                                idSourceTableName);
addOperation(removeIdSourceRefCol, operations);
var removeIdTypeRefCol : RDB::ops::ModelOperation :=
```

```
                                                    _removeColumn(
                                                       defaultSchemaName,
                                                       sourceAssociationTableName,
                                                       idTypeTableName);
         addOperation(removeIdTypeRefCol, operations);
         var removeSourceAssociationTable : RDB::ops::ModelOperation :=
                                                          _removeTable(
                                                            defaultSchemaName,
                                                            sourceAssociationTableName);
         addOperation(removeSourceAssociationTable, operations);
      }endif;
      return operations;
}
```

Kapitola 8

Závěr

Po několikaletém teoretickém a praktickém studiu změn v aplikačním modelu a jejich projevu v modelu aplikačním se nám podařilo definovat ucelenou množinu operací, pomocí nichž je možné měnit aplikační model a definovat jejich mapování na operace databázové úrovně.

Nejjednodušším a teoreticky nejzajímavějším tématem našeho projektu je aplikační model, kterému byly věnovány celkově 2 bakalářké a 2 diplomové práce BP - [Tar12] a [Luk11], DP [Tar14], a [Maz14].

Implementačně nejnáročnějším a zároveň nejvíce teoreticky vyčerpaným tématem se stala samotná transformace aplikačních operací na operace databázové, čehož jsem využil a k implementaci, otestování a dospecifikaci tohoto mapování jsem přidal téma čerstvé a velmi málo prozkoumané. Tímto atraktivním tématem vzhledem k verzování modelů je automatické rozpoznávání operací provedených nad aplikačním modelem.

Mrzí mně, že jsem se nemohl více věnovat tématu rozpoznávání operací nad aplikačním modelem, takže jsem se nedostal k tématům jako je sémantickém rozpoznávání operací vycházející pravděpodobně z idei sémantického webu a implementací Resource Description Framework (RDF) a Ontology Web Language (OWL). Implementace sémantického rozpoznávání by mohla v budoucnu odhadnout změny nejen na základě syntaxe(struktury) databáze, ale i na základě významu názvů jednotlivých entit v databázi. Předpokládám, že potom by bylo možné detekovat mnohem jednoznačněji přejmenování entit - pokud by pomocí nějaké ontologie či podobného nástroje bylo jasně specifikováno, že zvíře je nadtypem býložravce a tento je nadtypem entity zebra, potom pokud v původním modelu existuje třída Zebra, která má jako rodičovskou třídu nastavenu třídu Býložravec a v výsledném modelu existuje třída Zebra s nadtřídou Zvíře a neexistuje třída Býložravec, potom by algoritmus měl snadněji detekovat přejmenování třídy Býložravec na třídu Zvíře i přes velkou strukturální podobnost s jinou třídou.

Je otázkou, jak by sémantičtější rozpoznávání operací bylo prospěšné v kontextu stále větších informačních systémech, kdy je občas těžké nazvat smysluplně entity aplikačního modelu, natož sémantiku jejich relace vůči jiným entitám.

V průběhu psaní této diplomové práce jsem si uvědomil, proč pro nás bylo občas obtížné definovat správné mapování aplikačních operací na operace databázové. Čím blíže bylo zaměření daného participanta bližší aplikačnímu modelu, tím více si tento participant přibližoval databázový model aplikačnímu a vznikaly tak entity jako jsou HasNoInstance s názvem rodičovské tabulky bez jakékoliv relace rodičovství existující v databázovém modelu. Druhou

chybou, kterou jsme dělali při zaměření se na aplikační metamodel bylo modelování aplikačních operací, které měly přespříliš složité mapování na databázové operace či někdy až nesmyslný vliv na data v modelu - příkladem bylo modelování operace SetOpposite, která v aplikačním modelu spojí dvě existující property tříd a až při implementaci mapování na databázové operace bylo zjišťěno, že tato operace pozbývá smyslu. Tato odlehlost členů Migdb od databázového modelu vytvořila nemalé koncepční problémy, jejichž řešení bylo nutné s nemalým úsilím vymyslet a implementovat ve velmi pozdní fázi projektu. Pokud bych psal Migdb znovu od začátku, zaměřil bych se více na kooperaci jednotlivých částí, aby bylo pro mně psaní transformace ORMo snazší. Tato transformace je esenciální pro chod projektu a proto je její správná implementace alfa omegou na úspěch projektu.

Samotná definice operací aplikovatelných na databázový model není pevná, což vzhledem k malému vzorku sbíraných požadavků vede k nejednoznačným ORMo mapováním. Proto v navazujících pracích by bylo dobré udělat operační výzkum a získat větší množství požadavků na tyto operace.

Věcí, kterou bych udělal znovu jinak při psaní Migdb od začátku by bylo využití jiného jazyka než je QVT. Tento mapovací jazyk se našim potřebám hodil málo, proto jsme časem přestali používat jeho zakladní koncept - mapování a nahradili ho Java-like programovacím stylem queries a helperů. Naráželi jsme na stále větší problémy a QVT nám nepřinášelo moc užitku, nýbrž některé nevýhody. Jedna z těchto nevýhod je například automatické ukládání jakýchkoliv pomocných entit do výstupních modelů, které bylo nutné vyřešit (za účelem správného otestování) speciální transformací kopírující jen ty části výstupního modelu, které byly opravdovým výstupem, nikoliv meziproduktem. Tato transformace vzhledem ke svojí povaze samozřejmě zpomaluje exekuci Migdb Workflow.

Poslední otázkou, na kterou jsem neměl moc času hledat odpověď je portabilita projektu Migdb funkčního nad relační databází PostgreSQL na jiné relační databáze. Předpokládám, že by nebylo složité změnit generátor kódu pro jiné relační databáze - algoritmus vygenerování SQL kódu je poměrně přímočarý a pro PostgreSQL nebyl dlouhý. Struktura jiných relačních databází není vždy stejná, ale většinově se shoduje, takže ani modifikace databázového metamodelu by neměla být obtížná. Moje minimální zkoumání tohoto tématu odhalilo, že námi používaná databáze PostgreSQL má maximální délku 64 znaků, databáze Oracle má maximální délku identifikátoru 30 znaků a databáze Microsoft SQL server má maximum stanoveno na 128 znaků. Z tohoto vyplývá, že převod Migdb na databázi Microsoft SQL server by nebyl z tohoto pohledu problematický. 30 znaků pro Oracle nevypadá jako problematické - vývojáři mají málokdy třídy ukládané do databáze s jmény delšími než 30 znaků, problém nastává při zahrnutí service pro získávání názvů databázových entit k entitám z aplikačního modelu. Získání názvu cizího klíče, kolekce a asociační tabulky spojuje název atributu a tabulky s nějakým prefixem a odděluje tyto položky jména podtržítky.

Tudíž skutečné omezení délky názvu třídy může být základních 30 oslabeno o 3 (prefix FK či kolekce), oslabeno o 3 (podtržítka oddělující jednotlivé tři části jména - prefix a dvě tabulky) děleno 2 (dvě části). Aplikací těchto operací dostaneme horní hranici 12 znaků, která vypadá jako dostatečná, ačkoliv ne tolik komfortní. V této hranici jsme nicméně nezapočítaly Camel-podtržítkovou konverzi velkých písmen na malá předražená podtržítky. Předpokládejme, že každý identifikátor nebude mít víc jak 3 slova, tudíž musíme odečíst další 3 znaky. 9 nemusí být vždy dostatečná hranice vzhledem k velikosti nynějších systémů a nezapočítáváme do toho fakt, že třídy v aplikačním modelu mohou(a často jsou) prefixovány nějakým workspacem či balíčkem. Tudíž délka 9 nemusí být maximální horní hranice

jmen našich tříd a property. Je tedy zřejmé, že potom bude muset uživatel projektu Migdb nad databází Oracle používat krátké a naprosto neintuitivní názvy entit typu Vec102 či je nutné vymyslet jiný způsob překládání jmen do databázového modelu, který bude automatizovaný, jednoznačný, nezávislý na ostatních entitách v modelu a nejlépe pro člověka snadno získatelný bez pomoci nějakého překladače.

Není možné generovat zkrátit názvy entit - vedlo by to u kolizních názvů pomocí indexace, protože bez dalších pravidel jako například očív případě potřeby odstranit Constraint není možné zjistit, ke které entitě

8.1 Další poznámky

8.1.1 České uvozovky

V souboru `k336_thesis_macros.tex` je příkaz `\uv{}` pro sázení českých uvozovek. „Text uzavřený do českých uvozovek.“

Kapitola 9

Seznam použitých zkratek

IDE Integrated Development Environment

ORM Object-relational mapping

EMF Eclipse modeling framework

SŘBD Systém řízení báze dat

IO Integritní omezení

LCS Longest common Subsequence

OMG Object Management Group

ORMo Object-relation mapping of operations

QVT Query view transformational

QVTo QVT operational

DDL Data definition Language

SŘBD Systém řízení báze dat

OCL Object Constraint Language

⋮

Kapitola 10

UML diagramy

Tato příloha není povinná a zřejmě se neobjeví v každé práci. Máte-li ale větší množství podobných diagramů popisujících systém, není nutné všechny umísťovat do hlavního textu, zvláště pokud by to snižovalo jeho čitelnost.

Kapitola 11

Instalační a uživatelská příručka

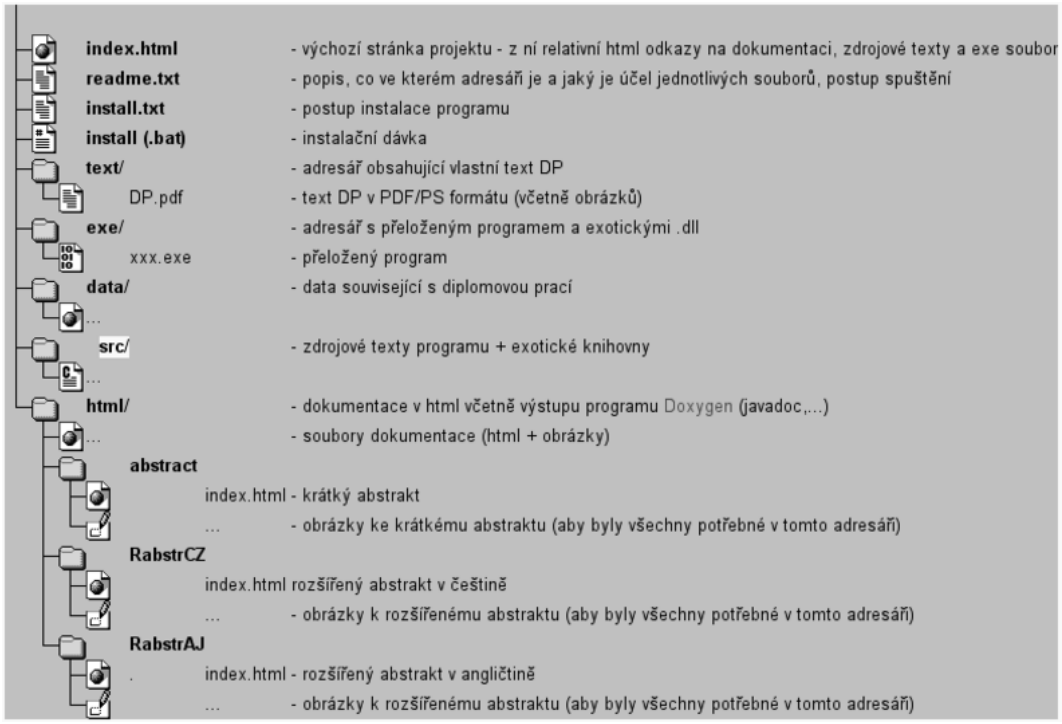
Tato příloha velmi žádoucí zejména u softwarových implementačních prací.

Kapitola 12

Obsah přiloženého CD

Tato příloha je povinná pro každou práci. Každá práce musí totiž obsahovat přiložené CD. Viz dále.

Může vypadat například takto. Váš seznam samozřejmě bude odpovídat typu vaší práce.



Obrázek 12.1: Seznam přiloženého CD — příklad

Na GNU/Linuxu si strukturu přiloženého CD můžete snadno vyrobit příkazem:

```
$ tree . >tree.txt
```

Ve vzniklém souboru pak stačí pouze doplnit komentáře.

Z **README.TXT** (případne index.html apod.) musí být rovněž zřejmé, jak programy instalovat, spouštět a jaké požadavky mají tyto programy na hardware.

Adresář **text** musí obsahovat soubor s vlastním textem práce v PDF nebo PS formátu, který bude později použit pro prezentaci diplomové práce na WWW.

Literatura

[Ben02] E. Bengoetxea. *Inexact Graph Matching Using Estimation of Distribution Algorithms*. PhD thesis, Ecole Nationale Supérieure des Télécommunications, Paris, France, Dec 2002.

[Cic08] Antonio Cicchetti. *Difference Representation and Conflict Management in Model-Driven Engineering*. PhD thesis, Universit‘ a di L'Aquila, 2008.

[DSK14] D. Di Ruscio R. F. Paige D. S. Kolovos, Alfonso Pierantonio. Different models for model matching: An analysis of approaches to support model differencing. [online], Citováno 16.11.2014.

[Fou14a] The Eclipse Foundation. Eclipse. [online], Citováno 16.11.2014.

[Fou14b] The Eclipse Foundation. Xtend - modernized java. [online], Citováno 16.11.2014.

[FW14] Sabrina Fortsch and Bernhard Westfechtel. Differencing and merging of software diagrams. [online], Citováno 16.11.2014.

[HET14] Ulrike Prange Hartmut Ehrig and Gabriele Taentzer. Fundamental theory for typed attributed graph transformation. [online], Citováno 7.12.2014.

[Jez12] Jiří Jezek. Modelem řízená evoluce objektů, 2012.

[Luk11] Martin Lukeš. Transformace objektových modelů, 2011.

[Luk13] David Luksch. Katalog refaktoringu frameworku migdb, 2013.

[Maz14] Martin Mazanec. Doménově specifický jazyk pro migdb, 2014.

[OCL14] OCL. Object constraint language. [online], Odkazováno 16.11.2014.

[OMG14a] OMG. Meta object facility (mof) 2.0 query/view/transformation final adopted specification. [online], Citováno 16.11.2014.

[OMG14b] OMG. Xmi. [online], Citováno 16.11.2014.

[OMG14c] OMG. Model driven architecture. [online], Odkazováno 16.11.2014.

[OMG14d] OMG. Omg. [online], Odkazováno 16.11.2014.

[PMH12] Jiří Jezek Pavel Moravec, Petr Tarant and David Harmanec. A practical approach to dealing with evolving models and persisted data. [online], [konference], publikováno 15.4.2012.

[Tar12] Petr Tarant. Modelem řízená evoluce databáze, 2012.

[Tar14] Petr Tarant. Migdb - formální specifikace, 2014.

[Val14a] Michal Valenta. Relační algebra. [online], Citováno 16.11.2014.

[Val14b] Michal Valenta. Integritní omezení. [online], Citováno 22.11.2014.

[wc14a] wiki community. Diff definice. [online], Citováno 16.11.2014.

[wc14b] wiki community. Longest common subsequence problem. [online], Citováno 16.11.2014.

[wc14c] wiki community. Orm definice. [online], Citováno 16.11.2014.

[wc14d] wiki community. Patch. [online], Citováno 16.11.2014.