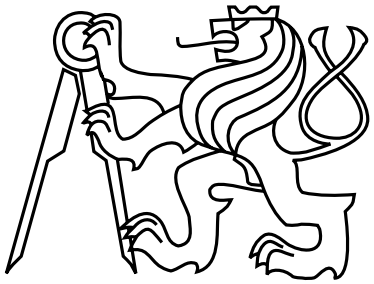
Na tomto místě bude oficiální zadání vaší práce

- 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

3. ledna 2015

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

Seznam obrázků

Seznam tabulek

List of Algorithms

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.

Tato diplomová práce se věnuje dvěma základním tématům. Prvním tématem je dokončení projektu Migdb, aby byl implementovaný framework nasaditelný k reálnému použití, tomuto tématu se věnuje kapitola 3, přičemž kapitola 2 popisuje projekt Migdb, jeho historii a dosažené výsledky před započetím této diplomové práce. Druhým tématem je automatické rozpoznávání operací nad aplikačním modelem. Tomuto tématu se věnuje kapitola 4, im-

plementaci rozpoznávacích algoritmů se potom věnuje kapitola 5. V kapitole 6 jsou shrnuty způsoby otestování jednotlivých částí implementovaného frameworku a jsou zhodnoceny jejich výstupy. Kapitola 7 bilancuje dosažené výsledky této diplomové práce a navrhuje další možné směry k rozvíjení daných dvou témat.

Kapitola 9 obsahuje manuál popisující použití frameworku Migdb, který je uložen na přiloženém CD. Kapitola 10 potom obsah přiloženého CD.

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 (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 databázových elementů odpovídajících příslušným elementům v 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 projektu Migdb bylo v posledních letech vytvořeny a obhájeny 4 bakalářské práce a 2 diplomové práce členů týmu Migdb.

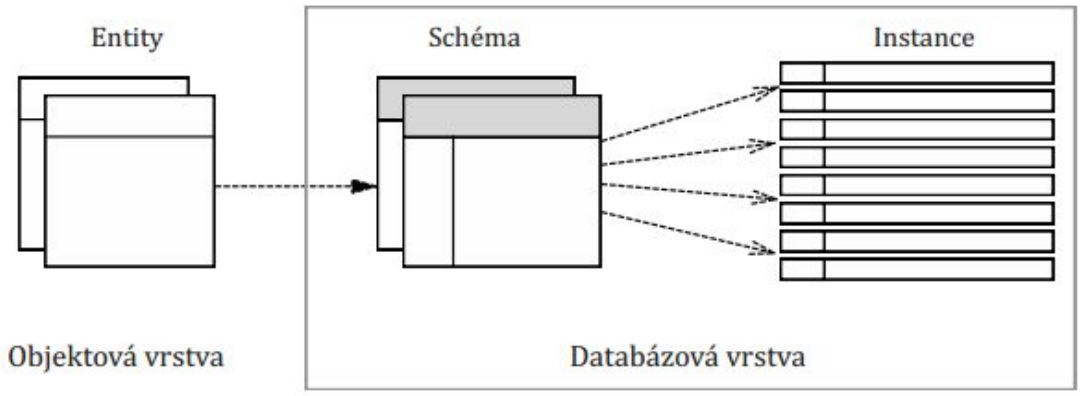
Jednalo se o tyto bakalářské práce - bakalářská práce mé osoby [Luk11], jež pojednávala o problematice mapování aplikačního modelu na model databázový, bakalářské práce Jiřího Ježka [Jez12] popisující aplikační model a jeho transformace, bakalářská práce Petra Taranta [Tar12] popisující databázový model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

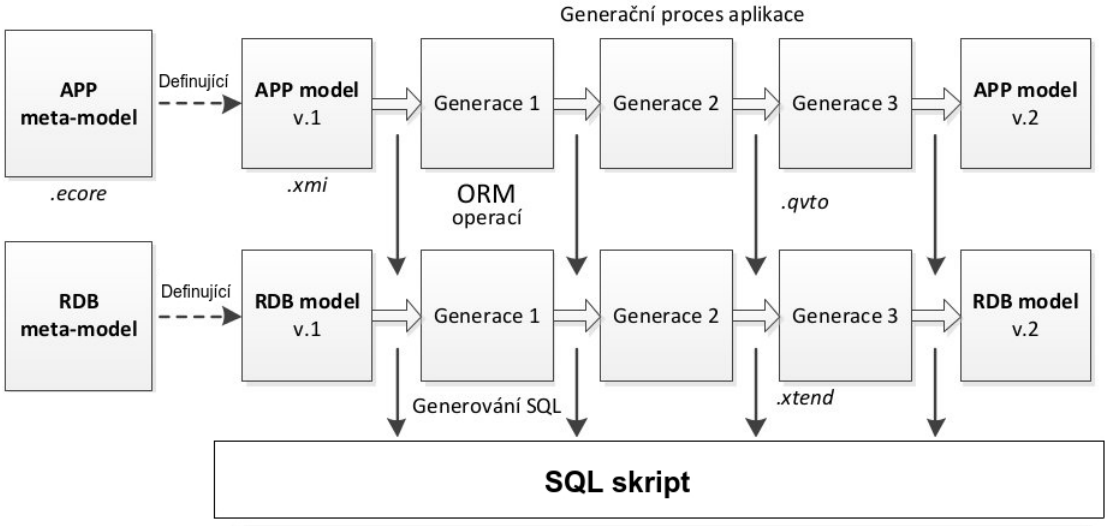
2.1.1 Metamodely

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. Databázový metamodel definuje elementy tvořící databázi a operace aplikovatelné na tyto elementy.

Skutečnost, že model A je definován pomocí metamodelu M_A, vyjadřuje, že metamodel M_A popisuje model A a model A je instancí metamodelu M_A.

2.1.2 Proces transformace modelů

Proces transformace modelů frameworkem je zobrazen na obrázku 2.2.



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

Na obrázku vidíme aplikační a databázový metamodel. Instancemi aplikačního APP metamodelu jsou jednotlivé aplikační modely. Na vstupní aplikační model v.1 je aplikována sada aplikačních operací a tento model je transformován do výstupního aplikačního modelu v.2. Množinu stavů, kterými aplikační model prochází před dosáhnutím své výstupní podoby nazýváme generacemi aplikačního modelu. Procesu změny aplikačního modelu budeme říkat Aplikační Evoluce nebo Evoluce aplikačního modelu.

Instancemi databázového RDB metamodelu jsou potom jednotlivé RDB modely. Stejně jako aplikační model je i model databázový postupně transformován. Procesu vývoje databázového modelu budeme říkat Databázová Evoluce nebo Evoluce databázového modelu. Stavy, kterými databázový model prochází potom nazveme generace databázového modelu.

Změny, které se provádějí v aplikaci jsou transformovány na databázové operace aplikovatelné na RDB model. Databázové operace jsou potom transformovány na SQL skripty. 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 frameworku, ale byl použit při testování frameworku - viz kapitola 6. Framework Migdb pracuje oproti původní sekvenční představě iteračně viz obr. 2.2. 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 vyjádřit 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 aplikační model a jaké podmínky musí daný aplikační model splňovat, aby se tato operace dala úspěšně provést. Více v sekci 2.6.

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 2.7.

Spojnici mezi aplikačním a databázovým modulem tvoří dva moduly - modul ORM a 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 2.9.

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. Detailnějším popisem implementace těchto modulů se nebudu v této práci blíže zabývat. Oba dva generátory byly napsány v jazyce Xtend.

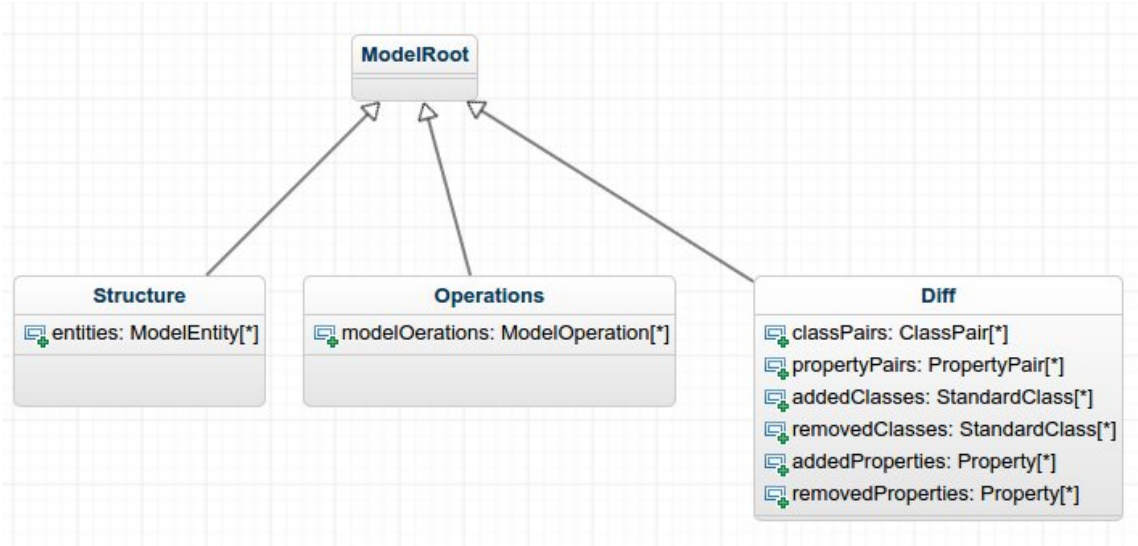
Nejnovějším modulem frameworku je modul OpsRecognition. Tento modul se snaží ze dvou vstupních aplikačních modelů odvodit seznam aplikačních operací. První vstupní model označíme jako model zdrojový. Druhý vstupní model označíme jako cílový model. které byly provedeny nad zdrojovým modelem a transformovaly ho do modelu cílového. Tento modul je samostatným nástrojem a pracuje nezávisle na ostatních modulech. V rámci tohoto modulu byly definovány dva algoritmy pro rozpoznávání operací. Tématu rozpoznávání operací je věnována kapitola 4.

Moduly Aplikační Evoluce, Databázové Evoluce, ORM, ORMo a OpsRecognition 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 Aplikační metamodel

Aplikační metamodel je rozdělen do 3 částí - definice struktury aplikace, seznam aplikačních operací a seznam rozdílových elementů. Pro každou část je definován vlastní container, do kterého se ukládají elementy obsažené v této části. Na obrázku 2.3 jsou znázorněny kořenové elementy nynějšího aplikačního modelu - každý aplikační model musí obsahovat nejméně jeden container - potomka třídy ModelRoot. Aplikační operace jsou popsány v sekci 2.6. Kořenový element Diff je popsán v sekci 5.6.1.



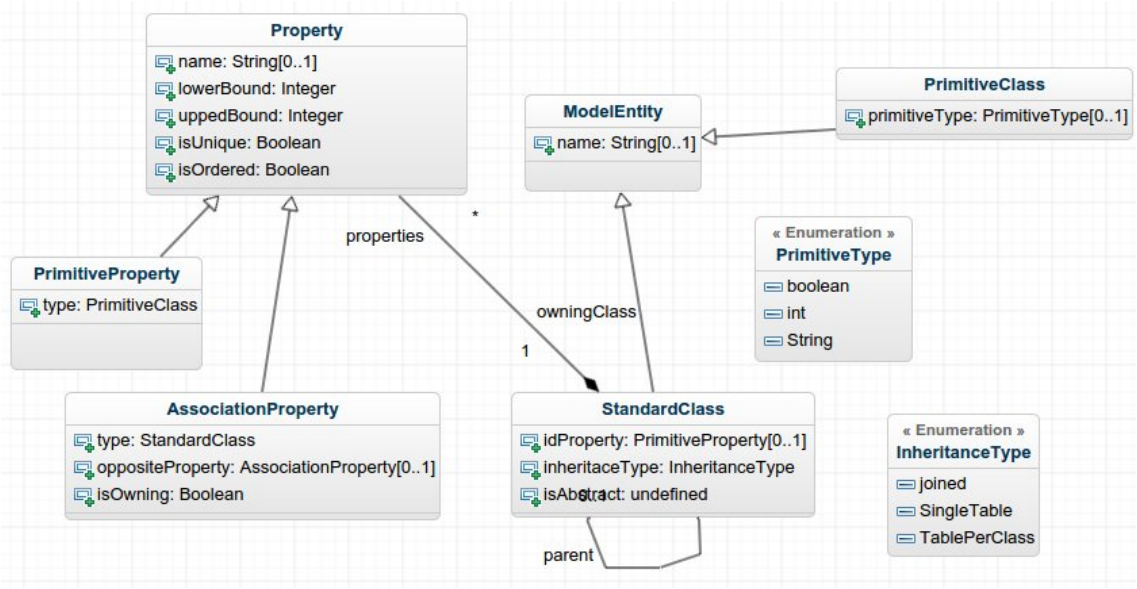
Obrázek 2.3: Rootové elementy aplikačního modelu

2.4 Struktura aplikace

Struktura aplikace zachycuje vztahy mezi jednotlivými objekty tvořícími aplikaci. Jednotlivé elementy struktury aplikace jsou obsažené v kořenovém elementu *Structure*. Na obrázku 2.4 jsou zobrazeny elementy patřící do Struktury aplikačního metamodelu.

Struktura aplikačního modelu obsahuje množinu elementů *ModelEntity*. Každá *ModelEntita* obsahuje svůj identifikátor *name*. Primitivní typy programovacího jazyka jsou reprezentované elementem *PrimitiveClass* potomkem *ModelEntity*. První potomek *ModelEntity*, *PrimitiveClass* obsahuje jen *primitiveType*. Druhým potomkem *ModelEntity* je *StandardClass* a obsahuje specifikaci svého *inheritancetType*, určující způsob uložení dat. *StandardClass* dále obsahuje 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*, které dohromady určují násobnost vazby či vymezují vlastnosti primitivní Property. *Property* dále obsahuje pro kolekce důležité atributy *isUnique* a *isUnique*. *PrimitiveProperty*, potomek Property, rozšiřuje svou rodičovskou třídu jen o svůj *type*, který musí být primitivní. *AssociationProperty*, druhý potomek Property, obsahuje také

type [StandardClass], referenci na *oppositteProperty* pro bidirectional vazby a atribut isOwning, který existuje z implementačních důvodů více o něm bude zmíněno v kapitole 3.

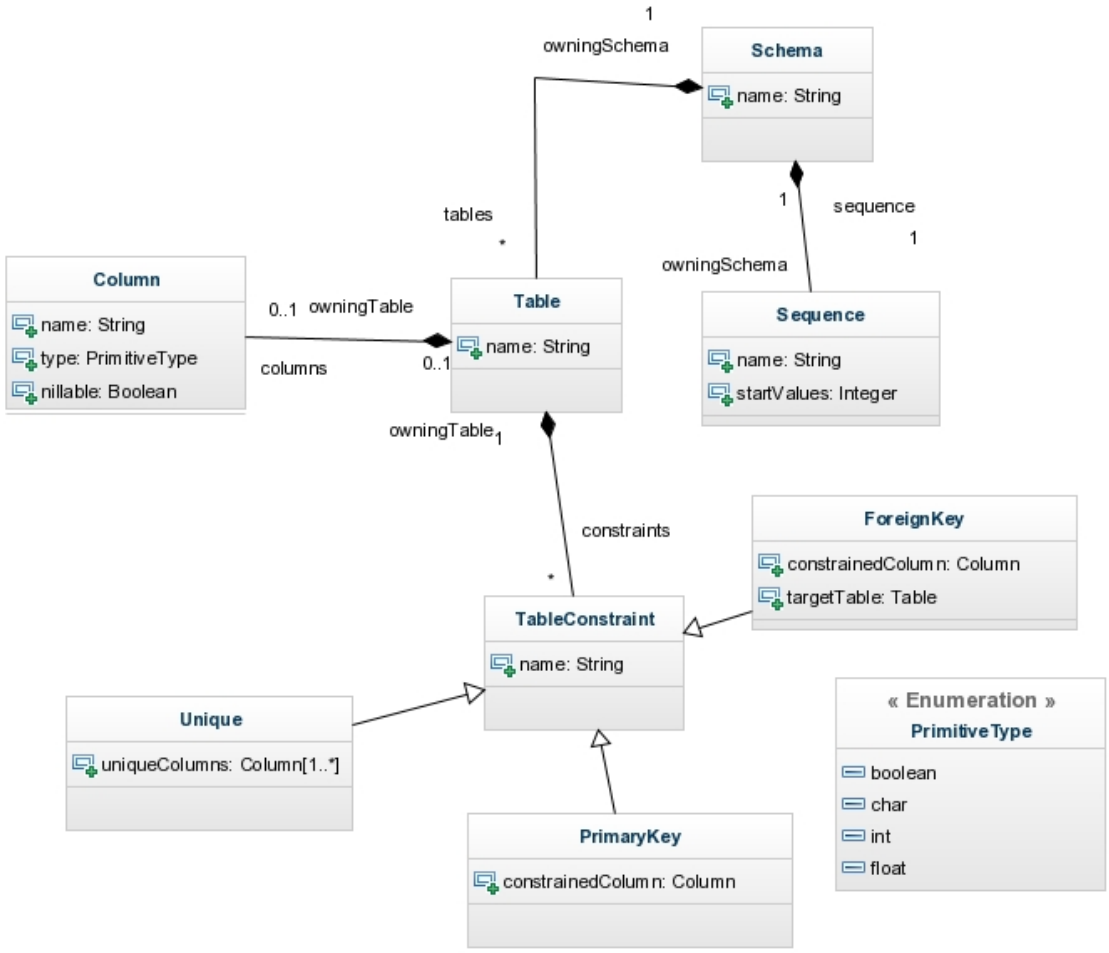


Obrázek 2.4: Struktura aplikačního metamodelu

2.5 Struktura databáze

Databázový metamodel od počátku vývoje definuje elementy nutné k specifikaci struktury databáze a databázové operace. Námi používanou databází je databáze PostgreSql, databázový metamodel je vytvořen na základě této databáze a může se mírně odklánět od jiných relačních databází. Základním databázovým konstruktem je *Schema*, které je jednoznačně identifikované svým *name*, obsahuje sezname tabulek a seznam sekvencí. *Sequence* je databázový element potřebný k postupnému generování čísel, 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 tabulku *owningTable*.

Potomci *TableConstraint* jsou jednotlivé IO a mají společné *name* zpřístupňující daný TableConstraint. TableConstraint *Unique* obsahuje seznam unikátních sloupců *uniqueColumns* a odkaz na vlastnickou tabulku *ownintTable*. Ačkoliv je v databázi možné mít vícesloupcový primární klíč, omezili jsme si *PrimaryKey* tak, aby ho bylo možné definovat jen nad jedním sloupcem *constrainedColumn*, protože již od začátku vývoje bylo zřejmé, že v našem frameworku budeme pracovat jen s umělými jednosloupcovými klíči. *ForeignKey* je poslední TableConstraint a podobně jako PrimaryKey má omezen počet sloupců tvořící cizí klíč na jeden *constrainedColumn* a obsahuje také odkaz na referencovanou tabulku *targetTable*. Na obr. 2.5 je zobrazen aktuální databázový metamodel.



Obrázek 2.5: Struktura databázového metamodelu

2.6 Operace nad aplikačním modelem

Operace nad aplikačním modelu definují možné transformace, které je možné provést aplikačním modelem. Každá aplikační operace má definované dvě metody. První metodou je metoda se signaturou boolean isValid(Structure structure), která pro danou aplikační strukturu zjistí, jestli je daná operace proveditelná nad touto strukturou, a vrátí tuto informaci ve své návratové hodnotě. Druhá metoda se signaturou void apply(Structure structure) aplikuje operaci na danou strukturu, tj. pozmění elementy v ní obsažené.

2.6.1 Seznam aplikačních operací

Operace jsou uvedeny v následujícím seznamu. Pro každou operaci jsou v seznamu uvedeny její validační podmínky a důsledky změny aplikace na model struktury. Validační podmínky určují, kdy by měla metoda isValid pro danou strukturu vracet true.

Kromě regulérních operací, které může vytvořit uživatel jsou v tabulce uvedeny i virtuální operace DistributeProperty, MergeProperty a operace ExportProperty. Virtuální operace nemůže vytvořit uživatel, jsou používány jako pomocné v implementaci složitějších operací a manipulují s Property v rámci dědičné hierarchie tříd. Tyto operace mohou narozdíl od nevirtuálních operací být aplikovány na model, který je z nějakého hlediska nevalidní a nemají definovanou operaci isValid. 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 asociační property odkazující typem 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, existuje ModelEntity s názvem shodným s typeName
- 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 s jménem shodným s newName v dané třídě
- 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 trída a property
- Operace nastaví horní a dolní mez property na nové hodnoty

Operace IX: SetOrdered(owningClassName, name, isOrdered)

- Validační podmínky - musí existovat daná trída a property
- Operace nastaví property atribut isOrdered na odpovídající hodnotu

Operace X: SetUnique(owningClassName, name, isUnique)

- Validační podmínky - musí existovat daná třída a property
- Operace nastaví property atribut isUnique na odpovídající hodnotu

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 (aplikuje virtuální operaci MergeProperty) kolizní atributy do rodičovské třídy

Operace XII: RemoveParent(className, parentClassName)

- Validační podmínky - musí existovat třída s name className a mít nastavenou hodnotu parent != NULL
- Operace odstraní třídě s name rovným className rodičovskou třídu a použije virtuální operaci DistributeProperty na 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 přes asociační property associationPropertyName, 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 s jménem associationPropertyName typu Inlinované třídy, která má upper bound 1

- Operace exportuje (aplikuje virtuální operaci exportProperty) na 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 ExportProperty) 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

2.6.2 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 ve výsledném modelu, která nemá žádné vazby na jiné entity. Příklady aditivní operace je operace AddClass.

Aplikace destruktivní operace zapříčiní, že entita ze vstupního modelu je odstraněna. Destruktivní operace jsou inverzí k operacím konstruktivním. Příkladem destruktivní operace je operace RemoveProperty.

Některé entity v modelu zůstávají či jsou nahrazeny entitami s jiným jménem a „stejným“ obsahem. Těmto entitám budem říkat řídící. Operace expanzivní přídává do výstupního modelu jednu entitu, čímž se podobá operaci konstruktivní, nicméně zároveň je vázána na jinou řídící entitu stejného typu a mění 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.

Modifikační operace nemění počet entit určitého typu, ale mění jejich atributy. Příkladem modifikační operace je operace SetBounds.

2.7 Databázové operace

Databázové operace popisují transformace, které mohou být provedeny nad databázovým modelem v průběhu Evoluce databázového modelu. Každá databázová operace má definovány dvě metody boolean isValid(Structure structure) a void apply(Structure structure). Metody mají stejný význam jako metody stejnojmenných aplikačních operací, jen pracují s databázovým modelem místo aplikačního. Metoda isValid(Structure structure) určuje, jestli je operace aplikovatelná na daný model databázové struktury. Metoda apply potom definuje důsledky, které má aplikace operace na daný databázový model.

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 a důsledky jejich aplikakace je uveden v následujícím seznamu:

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 dané 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: RenameColumn(owningSchemaName, 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: RemoveTable(owningSchemaName, name)

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

Operace XIII: RemoveColumn(owningSchemaName, owningTableName, name)

- Odstraní v daném schématu a tabulce column s jménem name
- Musí existovat schéma s name owningSchemaName, tabulka s name owningTableName, sloupec s name rovým name. Sloupec nesmí obsahovat constraint PrimaryKey, ForeignKey ani Unique

Operace XIV: RemoveConstraint(owningSchemaName, owningTableName, name)

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

Operace XV: RemoveSequence(owningSchemaName, name)

- Odstraní v daném schématu sequence s jménem name
- Musí existovat dané schéma a daná sequence

Operace XVI: UpdateRows(owningSchemaName, 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: NillRows(owningSchemaName, 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: InsertRows(owningSchemaName, 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 schéma s name owningSchemaName, v něm table identifikovaná sourceTableName, v ní column identifikovaná sourceColumnName, Ve schématu musí existovat table identifikovaná targetTableName, v ní column identifikovaná targetColumnName. SourceColumn musí mít stejný typ jako targetColumn

Operace XIX: DeleteRows(owningSchemaName, tableName, whereCondition)

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

2.8 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.

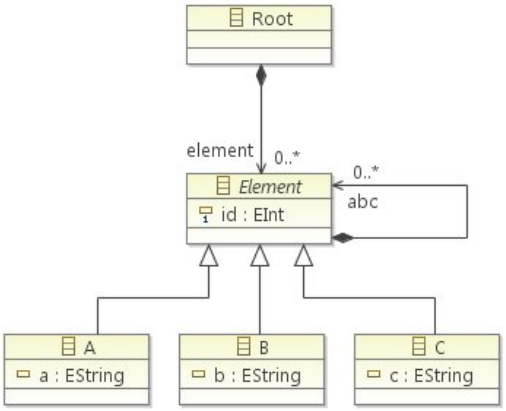
Základními konstrukty jazyka jsou mapping, helper a query. Mapping je konstrukt měníci pomocí nějakých pravidel vstupní element na výstupní. Query je dotazovací konstrukt, který získá potřebnou výstupní informaci z daného elementu. Helper je konstrukt, který může narozdíl od query měnit objekt, nad kterým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

Na následující ukázce vidíme ukázkový HelloWorld příklad QVTo kódu převzatý z [Siq14]

```
modeltype ABC uses ABC('http:///ABC.ecore');

transformation HelloWorld(in source:ABC, out target:ABC);

main() {
   source.rootObjects()[Root]->map Root2Root();
}

mapping Root :: Root2Root() : Root {
   element += self.element[A]->map A2B();
}

mapping A :: A2B() : B
 when {
      self.id > 0
 }
   {
      result.id := self.id;
      result.b := self.a + " World!";
   }
```

Na začátku každého .qvto souboru můžeme importovat potřebné knihovny pomocí klíčového slova import, čehož v ukázce nebylo zapotřebí. Následně deklarujeme typy modelů, které bude naše transformace používat pomocí klíčového slova modeltype. Metamodel ABC použitý v tomto příkladě je zobrazen na obrázku 2.6.

V metamodelu ABC je každý kořenový element složen z 0...n entit Element. Entita Element má své Id. Existují tři potomci entity Element, třídy A, B, C. Každý potomek entity Element obsahuje atribut typu řetězec, který má jméno shodné s názvem třídy. Každá Entita element může být obsahovat 0...n subelementů typu Element.



Obrázek 2.6: ABC metamodel převzatý z [Siq14]

Řádek *transformation HelloWorld(in source:ABC, out target:ABC);* nám definuje hlavičku transformace. Název transformace je HelloWorld a transformace mapuje jeden vstupní model *source* typu ABC na jeden výstupní model *target* typu ABC. Kromě vstupních a výstupních elementů mohou být v transformaci modely vstupně-výstupní označené klíčovým slovem inout.

Vstupním bodem každé transformace je metoda main. Ukázkový příklad v metodě main přistupuje pomocí *source.rootObjects()* k kořenovým elementům source modelu, vybírá z nich pomocí „[Root]“ všechny elementy typu Root. *source.rootObjects()[Root]* je ekvivalentní s ocl selectem *source.rootObjects()->select(e | e.oclIsTypeOf(Structure)).oclAsType(Structure)*. Transformace volá nad vybranou kolekcí elementů mapování Root2Root pomocí *->map Root2Root()*.

Mapování Root2Root s hlavičkou *mapping Root:: Root2Root() : Root* mapuje Element typu Root na jiný Element typu Root. V těle metody je vybrán každý element typu A z kolekce elementů vstupního elementu Root viz *self.elements[A]* a výsledek volání mapování A2B() je přidán do kolekce elements výstupní entity Root.

Mapování A2B() mapuje entity typu A na elementy typu B. Kódem v bloku when je určuna doplňující podmínka, mapovat se budou jen entity typu A s id > 0. Mapování nastaví výsledné entitě id vstupní entity a rozšíří text uložený v entitě o řetězec *World!*.

Ukázkový kód bude tedy transformovat vstupní model

```
ROOT:
  A: id=1, A=''Hello''
  A: id=-1, A=''World''
  C: id=1, C=''Something''
```

Na výstupní model

```
  A: id=1, A=''Hello World!''
```

2.8.1 Ukázka kódu Migdb

Jak již bylo řečeno v předchozí sekci, je možné kromě mapování definovat dotazovací konstrukt query a modifikační konstrukt helper. Oba konstrukty jsou si velmi podobné, jejich jedinou odlišností je, že query nesmí modifikovat objekt, nad kterým je voláno a helper může. V následujícím kódu je ukázáno validační query pro zjištění splnitelnosti validačních podmínek operace AddTable. Query isValid převolává dvě helpery. V helperu checkExistSchema vidíme kontrolu a případné zalogování chyby při nenalezení schématu, Není tedy možné vracet obrácenou hodnotu volání pokud chceme zjistit, že daná validační podmínka neplatí, protože by tímto voláním byla zalogována chyba.

```
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;
}
```

2.9 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 0 až N operací databázových. Většinou je aplikační operace namapována na nejméně 1 databázovou operaci. Výjimkou je operace SetAbstract pro případ v případě, že měníme abstraktní třídu na neabstraktní.

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 vlastnické tabulky sloupec pro primitivní property

Primitivní typ a UpperBound != 1 operace přidá do modelu tabulku, která je obrazem kolekce, do této tabulky přidá datový sloupec, 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

isAbstract = false mapuje na prázdnou množinu operací

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. Do této tabulky vytvoří nové sloupce extrahovaných properties a sloupec 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: virtual ExportProperty(sourceClassName, targetClassName, propertyName)

primitivní typ a UB = 1 Operace vytvoří sloupec v cílové tabulce, updatuje data v tomto sloupci a smaže sloupec v původní tabulce

primitivní typ a UB !=1 Operace přejmenuje referenční sloupec tabulky kolekce, odstraní cizí klíč referencující zdrojovou tabulku, přejmenuje starou tabulku obrazu původní kolekce na nové jméno, smaže z tabulky kolekce data, která nepatří targetClass, vytvoří cizí klíč referencující cílovou tabulku a přejmenuje tabulku kolekce

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(sourceClassName, targetClassName, propertyName)

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 a upravit Databázovou a Aplikační Evoluci. Tato kapitola popisuje změny těch částí projektu, které jsem samostatně nebo z větší části vymyslel a/nebo implementoval v poslední fázi vývoje.

3.1 Změny v aplikačním metamodelu

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. 3.1 viz [Luk11].

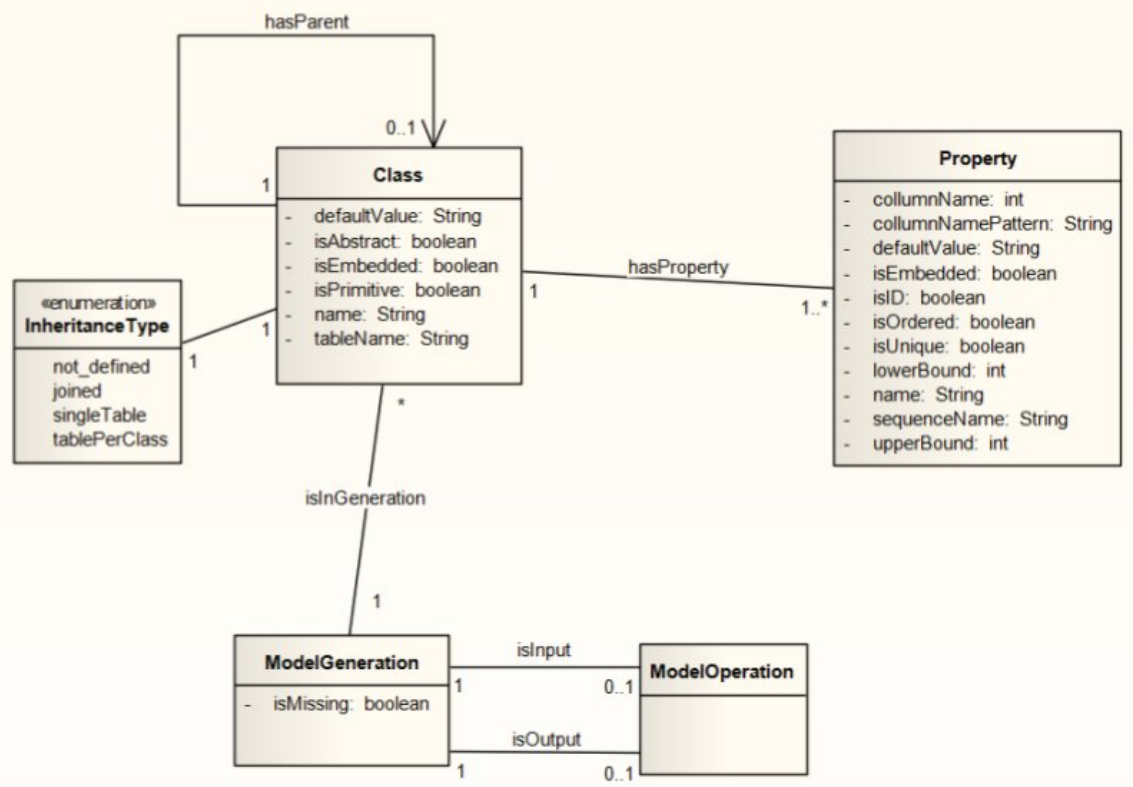
Postupem času byl aplikační model měněn viz. 3.2 [Jez12] a dočasně byly přidány entity podporující EmbeddedClass, které v nynější době v modelu již znovu nejsou.

V nynější chvíli došlo k oddělení struktury aplikace od seznamu aplikačních operací a přibyl kořenový element Diff.

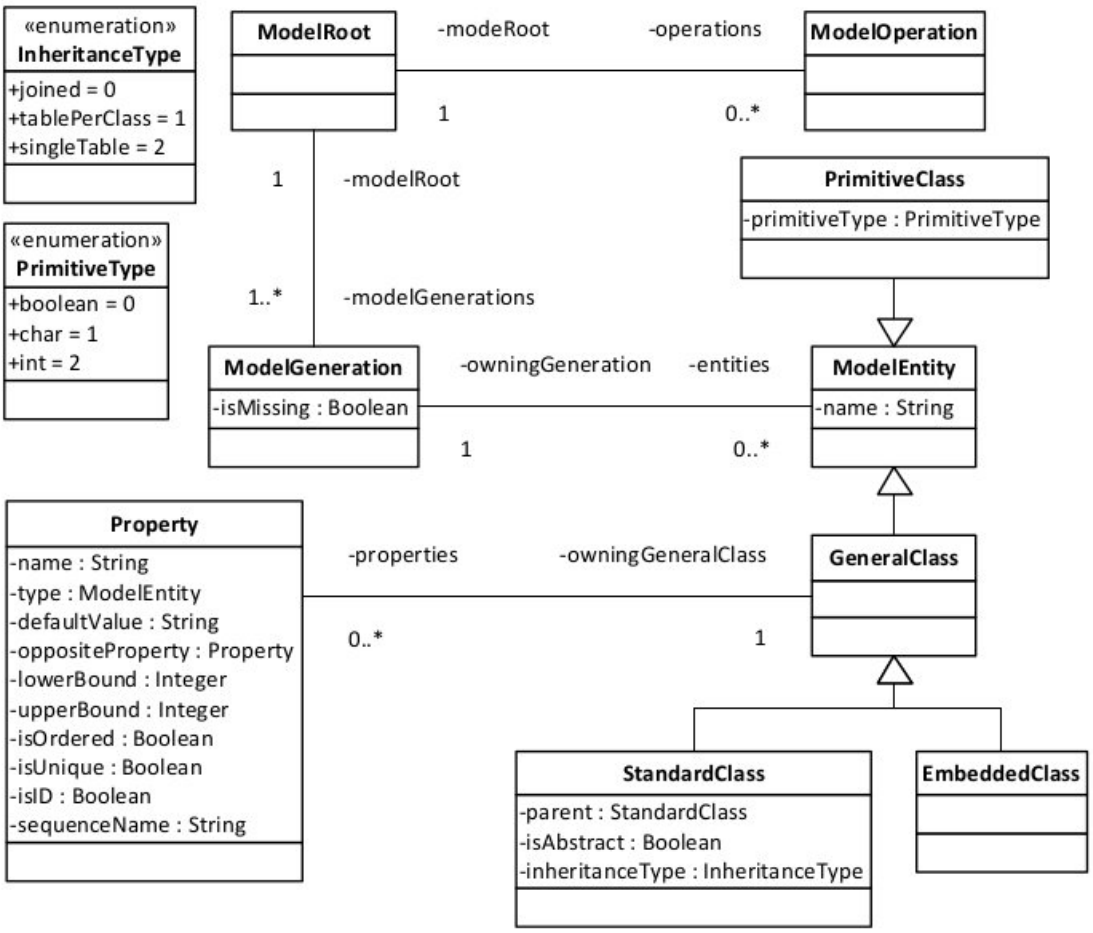
Oproti aplikačnímu metamodelu [Jez12] byly odstraněny entity EmbeddedClass a její předek GeneralClass, dále byla zjednodušena třída Property, u níž ubyly atributy defaultValue, sequenceName a atribut isId. Atribut isId byl nahrazen přímou referencí na idProperty ve třídě StandardClass který byl nahrazen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

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

Kvůli zajištění jednoznačnosti jmen odvozených z jmen aplikačních elementů byl do Property přidán atribut isOwning. Více o významu atributu isOwning bude zmíněno v sekci 3.6.1.



Obrázek 3.1: Aplikační metamodel v počátku vývoje obrázek převzat z [Luk11]



Obrázek 3.2: Aplikační metamodel v průběhu vývoje obrázek převzat z [Jez12]

3.2 Změny aplikačních 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.

3.2.1 Atomic, composed a virtuální operace

V průběhu vývoje existoval entity 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.

Některé nyní aplikované operace se rozkládají na operace virtuální na úrovni kódu, nikoliv modelu, aby bylo zabráněno duplikaci kódu. Na první pohled se zdá , že virtuální operace je obdobou Atomická operace, ale mezi těmito dvěma koncepty existují dva rozdíly. Prvním rozdílem je, že pro atomické operace vznikaly entity v modelu, což vedlo k jejich ukládání do mezivýsledných modelů a bylo nutné je mazat. Druhým rozdílem je, že pro atomické operace se ověřovaly validační podmínky, což se pro virtuální operace nedělá.

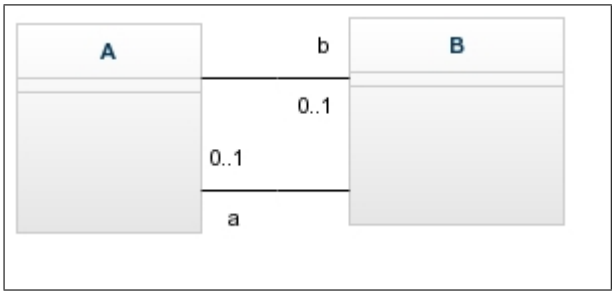
Koncept rozkladu operací na operace atomické se nedá považovat za špatný, ale je nutné definovat širší množinu atomických operací - některé jen pomocné například spojující třídy na základě nějakého kritéria. Tento čistší návrh podlehl nižšímu množství práce na implementaci a měl za následek vyšší složitost testů.

3.2.2 AddPrimitive

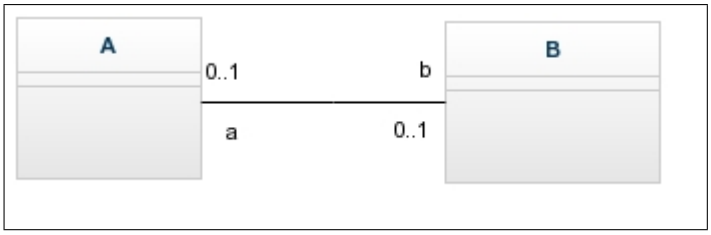
Operace AddPrimitive byla označena za nadbytečnou, protože není cílem modifikace modelu změnit seznam primitivních tříd. Tento seznam bývá definován použitým programovacím jazykem a tudíž by měl být ve vstupní generaci.

3.2.3 SetOppositte

V průběhu analýzy operace SetOpposite bylo zjištěno, že tato operace má smysl na strukturální úrovni, ale stává se problematickou při práci s instancemi dat. Operace bezproblémově funguje, pokud má odstranit nastavenou oppositeProperty, tj. rozpojit oboustraně navigabilní vazbu. Pokud má operace naopak stvořit oboustranně navigabilní vazbu, musí na aplikační úrovni zkontrolovat existenci opozičních properties, zkontrolovat typy nastavovaných properties. Strukturální kontrolu provede operace isValid(). Operace musí zkontrolovat, že existují správné instance dat v databázi a spojit je. A v tom tkví problém této operace. Bez znalosti instancí v databázi není možné najít takové mapování. Díky odstranění kontrol při běhu skriptu nad databází není možné uživatele frameworku upozornit na chybu v průběhu



(a) Stav před operací SetOpposite



(b) Výsledek po aplikaci operace SetOpposite

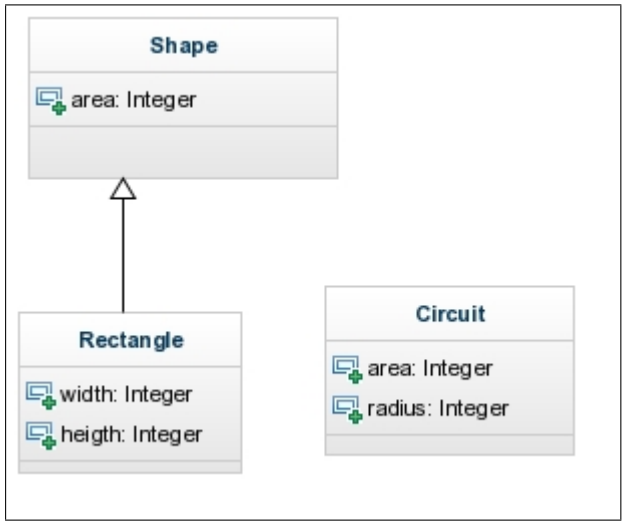
Obrázek 3.3: Ilustrativní příklad k operaci SetOppositte

této operace.

Zdrojový stav máme zobrazený na obrázku 3.3a a cílový stav Validační podmínky jsou v pořádku. Chceme dosáhnout stavu zobrazeného na obrázku 3.3b. Řešením tohoto problému bylo nahrazení operace SetOppositte dvojící operací ChangeBiToUnidir a ChangeUniToBidir. Operace ChangeUniToBidir mění jednostranně navigabilní vazbu přidáním property do třídy typu a nastavuje data opoziční property, operace tak nepotřebuje kontrolovat, jaká data jsou v opoziční property, protože ta před započetím operace neexistovala. Operace ChangeBiToUnidir pak odstraní oppositte atribut z vlastnické property a odstraní samotnou property nesoucí data. Nechtěným, avšak vítaným produktem této změny bylo zjednodušení vytváření oppositeProperty. Původní sekvence nutná k dosažení požadovaného výsledku zahrnovala operaci AddProperty následovanou operací SetOppositte. Nyní je možné dosáhnout tohoto cíle jedinou operací ChangeUniToBidir. Zjednodušuje se i mazání oboustraně navigabilní vazby. V původní variantě bylo nutné spustit operace SetOpposite s parametrem NULL pro opposite sloupec a následně smazat opposite property operací RemoveProperty. Nyní tuto funkcionalitu zajišťuje operace ChangeBiToUnidir.

3.2.4 AddParent, RemoveParent

V původním smyslu měla operace AddParent přidávat předka A třídě B, přičemž třídy A a B neměly kolizní property. Z praktického pohledu je tato aplikace operace AddParent nepoužitelná. Přidáváme-li existující třídu do hierarchie, chceme získat vztah isA, který nám definuje, že třídy mají nejen společnou funkcionalitu, ale téměř vždy i data. Představme si například, že modelujeme grafický editor. V prvním kroku jsme vytvořili třídu Square, která má properties area a side. Naprogramovali jsme kód používající třídu Square, vytvořili jich několik, nakreslili... a uložili. V druhé fázi jsme zjistili, že potřebujeme více tvarů a tak jsme



Obrázek 3.4: Model v průběhu vývoje aplikace

vytvořili třídu Circuit pro kruh. Následně jsme po napsání kódu a uložení některých instancí Circuit do databáze zjistili, že potřebujeme v některých případech pracovat s třídou Circuit stejně jako s třídou Rectangle. Proto jsme extrahovali třídu Shape, předka třídy Square. A v nynější chvíli nám operace ve frameworku Migdb nedostačují, protože potřebujeme nastavit Shape jako rodičovskou třídu třídě Circuit, ale v tom nám zabraňuje kolizní property. Na obrázku 3.4 vidíme nynější stav. Tento stav jsme vyřešili změnou validačních podmínek operace AddParent, kolizní property mohou existovat. V zájmu jednoduchosti operace AddParent a jejího snadného zápisu v jazyku Martina Mazance jsme nepřidávali kolizní property do signatury operace. Operace je schopna si kolizní property dopočítat.

Operace RemoveParent, inverzní operace AddParent, neví nic o původním stavu, není schopna si původní kolizní property spočítat, takže její implementace se změnila a tato operace kopíruje všechny property rodičovské třídy do potomka. Slabinou v tomto přístupu je vznik nekonzistence inverze s původní operací viz příklad nekonzistence uvedený v subsekci 3.2.6.

3.2.5 SetType

Operace SetType byla zkoumána, 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 tato operace by měla být aplikována na změny typu v hierarchii. Pokud by byla tato operace totiž aplikována na primitivní typy, není možné kontrolovat data či dát uživateli zprávu o nevalidnosti SQL skriptu.

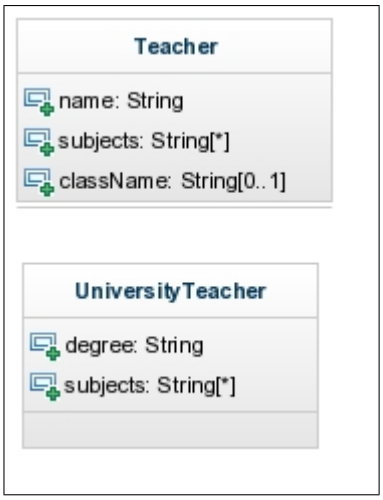
3.2.6 Vlastnosti operací

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

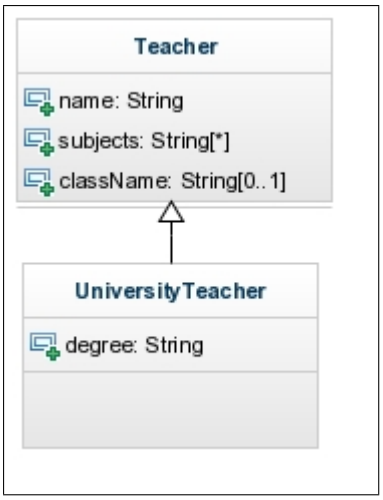
minulosti byly operace aplikovatelné na jeden konkrétní model tzv. intensional a modernější differenční modely jsou tzv. extensional - je možné je aplikovat na jakýkoliv model, například na paralelní vývojové větve.

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.

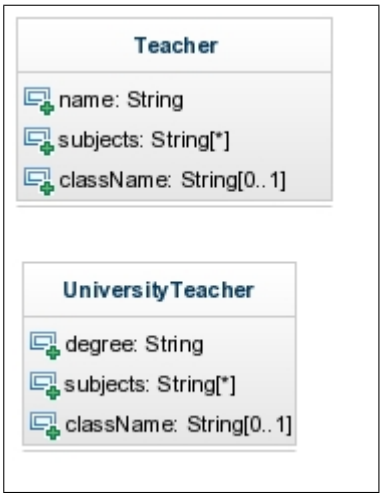
Problémem je, že i po odvození inverze nemusí vést aplikace operace do stejného vstupního modelu. Pokud například na stav 3.5a aplikujeme operaci AddParent(Teacher, UniversityTeacher), získáme cílový stav 3.5b. Pokud se chceme vrátit zpět z 3.5b do výchozího stavu, měli bychom aplikovat operaci RemoveParent(UniversityTeacher), nicméně aplikace této operace nepovede do výchozího stavu 3.5a, ale do stavu 3.5c. 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 3.5: Ukázka operace AddParent

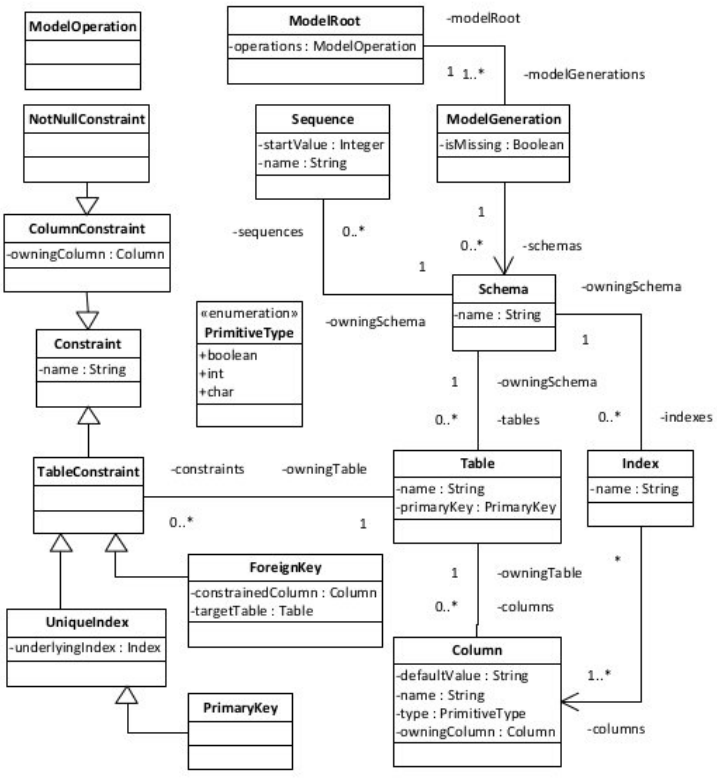
3.3 Změny Databázového Struktury

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

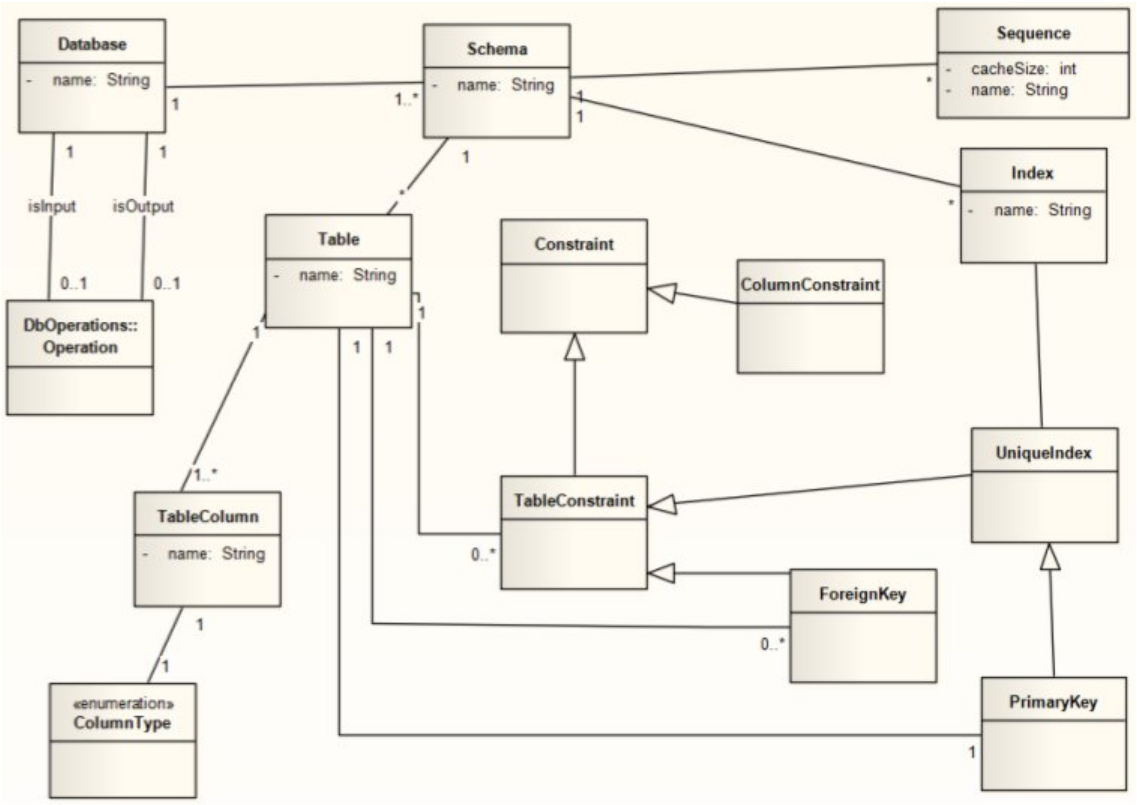
Na obr. 2.5 vidíme aktuální matamodel databázové struktury, na obr. 3.7 vidíme model na počátku vývoje převzatý z [Luk11] a na obrázku 3.6 model v pozdější fázi vývoje převzatý z [Tar12].

Nejvýraznější změnou databázového metamodelu struktury je stejně jako v metamodelu aplikační struktury odstranění generace modelů. Dalšími změnami jsou 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. Bezatributový element NotNullConstraint byl shledán příliš informačně chudým a byl nahrazen atributem isNillable zachovávajícím stejnou informační hodnotu jako element v původních modelech.



Obrázek 3.6: Struktura databázového metamodelu v průběhu vývoje, obrázek převzat z [Tar12]



Obrázek 3.7: Struktura databázového metamodelu v průběhu vývoje, obrázek převzat z [Luk11]

3.4 Změny databázových operací

Databázové operace v původní verzi byly designově nečisté, některé operace obsahovaly ve své definici odkazy na entity z modelu aplikačního, které již v modelu databázovém neexistují a ani nemohou existovat. Mou prací na databázových operacích bylo odstranění entit aplikace, které v databázovém metamodelu již nemohou existovat a dodefinovat potřebné obecně použitelné operace.

3.4.1 AddSchema a RemoveSchema

Operace AddSchema a RemoveSchema byly odstraněny. Pro každou tabulku je definováno, ve kterém schematu se nalézá, ale neexistuje aplikační operace, která by zapříčinila vznik nebo smazání databázového schematu.

3.4.2 HasNoInstances, HasNoOwnInstances

Operace HasNoInstances měla zjistit, jestli v dané tabulce existují data. Operace HasNoOwnInstances zjišťovala, jestli v tabulce odpovídající třídě A existují data odpovídající konkrétní

instanci A, nikoliv data odpovídající instancím jejích potomků. Tyto dvě operace byly napsány jako základ ověřování proveditelnosti migrovaných skriptů, který byl z frameworku vypuštěn. Pokud by se měl do frameworku navrátit, bylo by lepší namodelovat jednu operaci HasNoInstances s where podmínkou vybírající daná data.

3.4.3 GenerateSequenceNumbers

Operace GenerateSequenceNumbers byla shledána nadbytečnou, protože při vkládání dat je možné generovat je ze sekvence přidružené k tabulce či specifikované v parametru. Není tedy nutné vkládání dat do tabulky oddělovat od Generování data ze sekvence.

3.4.4 AddIndex, RemoveIndex

Z databázového metamodelu byl odstaněn element Index. Některé operace sice vytvářejí defaultní index, ale k tomuto vytvoření stačí jejich samotné zavolání. Z těchto důvodů byly operace z metamodelu odstraněny.

3.4.5 SetColumnType

Operace SetColumnType měla konvertovat data z jednoho primitivního typu na druhý. V nynější chvíli není aplikační operace, která by se na tuto databázovou operaci mapovala a je očekávané, že aplikační operace SetType bude měnit neprimitivní typ. Operace se tedy stala nadbytečnou a byla z modelu odstraněna.

3.4.6 UpdateRows

Shledal jsem nadbytečným atribut ToleranceType, jelikož jeho přesný význam nebyl definován a nebyl použit v ORMo mapování. V operaci byl nahrazen atribut idName atributem selectionWhereCondition. Testy odhalily, že je nutné přidat volitelný atribut safeWhereCondition, aby byla zajištěna neměnnost instancí dat, které operace nemá změnit.

3.4.7 DeleteRows

Signatura operace byla změněna z verze:

```
class DeleteRows extends ModelOperation {
  attr String[1] owningSchemaName;
  attr String[1] tableName;
  attr String[*] descendantsNames;
  attr String[1] idName;
}
```

Na verzi:

```
class DeleteRows extends ModelOperation {
  attr String[1] owningSchemaName;
  attr String[1] tableName;
  attr String[1] whereCondition;
}
```

Výstupní verze je čitelnější, univerzálnější a designově čistší.

3.4.8 NillRows

Motivací k vytvoření této operace bylo shledání potřebného nastavování NULL hodnoty v cílové tabulce. Operace nebyla shledána jako speciální typ UpdateRows, jelikož operace neaktualizuje data v cílové tabulce na základě zdrojové tabulky.

3.4.9 InsertRows

Do operace byl přidán atribut whereCondition, aby byla operace lépe použitelná.

3.4.10 RemoveNotNull

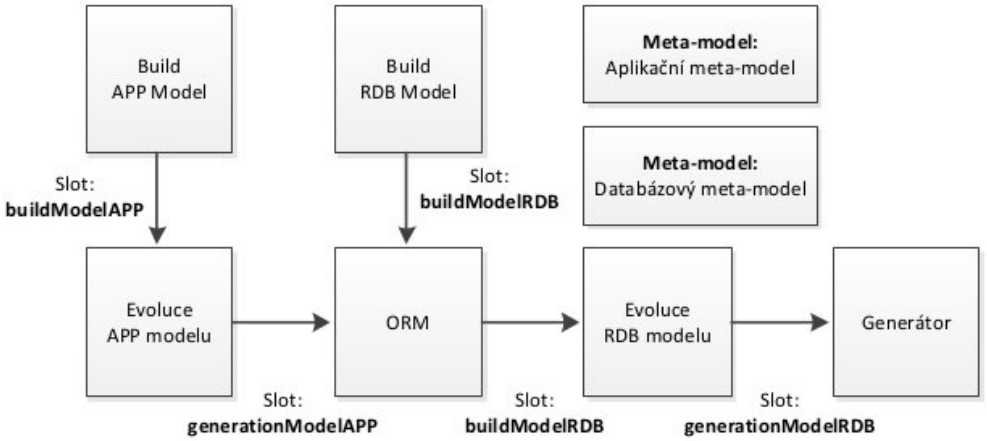
NotNull constraint byl zpočátku vývoje chybně zařazen mezi TableConstraints a bylo předpokládáno, že je odstranitelný operací RemoveConstraint. Po zjištění neplatnosti tohoto předpokladu jsem vytvořil tuto operaci.

3.5 Databázová evoluce, Aplikační Evoluce a Migdb_Executor

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. Nicméně 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. Proto jsem se rozhodl tuto část zachovat ve frameworku.

3.5.1 Sekvenční představa

Původní představa o funkci frameworku Migdb je zobrazena na obrázku 3.8. Problémem tohoto návrhu je úplná sekvenčnost zpracování jednotlivých kroků. V prvním kroku se všechny aplikační operace aplikují na vstupní model. V druhém kroku se tyto operace transformují pomocí ORMo modulu na databázové operace. Díky závislosti ORMo mapování na aktuálním aplikačním modelu zde může nastat problém. Pokud budeme mít sekvenci operací $O_1, \ldots O_n$, která pozmění aplikační model na model M_1 tak, že některá z operací nebude validní, transformace ORMo nemusí dávat korektní výstup. Příklad této anomálie je aplikace operací SetBounds(1, 1), RemoveProperty(„C“, „P“). Pokud aplikujeme na model obsahující třídu „C“ s Property „P“ tyto dvě operace, pak v cílovém modelu nebude existovat Property „P“, kterou transformace ORMo potřebuje k namapování operace SetBounds.



Obrázek 3.8: Původní sekvenční představa. Převzato z [Tar12]

3.5.2 Změny SQL generátorů

Vzhledem k změnám databázových operací bylo nutné upravit i generátor SQL kódu z těchto operací. Druhou podstatnou prací bylo přepsání základního algoritmu pro vytvoření databázového schematu z aplikačního modelu použitého v mé bakalářské práci viz [Luk11], aby bylo možné napsat testy celého frameworku.

3.6 Implementace a testování ORMo mapování

Nejnáročnější implementační částí bylo napsání ORMo mapování a jeho testů. Původní implementace prezentovaná v Cambridge se ukázala jako velmi neflexibilní , špatně rozšířitelná a testovatelná. Proto jsem téměř celou ORMo implementaci přepsal. Implementace ORMo zabírá přes 2000 řádků kódu. Vzniklo 42 testů ORMo mapování.

3.6.1 Mapování jmen

ORMo mapování udržuje konzistenci mezi aplikačním modelem a modelem databázovým. Databáze umožňuje definovat nepojmenované constrainy, ale aby bylo možné smazat TableConstrain je nutné ho referencovat pomocí jeho jména. Databáze neumožňují měnit název TableConstrainů a tak je každou změnu názvu jména TableConstrainu nutné reprezentovat jako odstranění a jeho opětovné vytvoření. Aby ORMo mapování zajistilo konzistenci mezi aplikačním a databázovým modelem, bylo nutné zajistit jednoznačné mapování mezi elementy aplikační vrstvy a elementy databázovými. Kritickým problémem se stalo mapování jmen. Vznikla knihovna name_service, která tento problém řeší. Jména tříd je jednoduché transformovat na jména tabulek pomocí následujících pravidel:

- Počáteční písmeno je zmenšeno
- Každé další velké písmeno je nahrazeno malým písmenem stejného typu předraženého znakem "_"

Stejná pravidla jsou použita při transformaci primitivních properties s upperBound 1 na sloupce. Tedy třída LegalPerson(businessName, registrationNumber) se transformuje na tabulku legal_person(business_name, registration_number). Použití daných dvou pravidel označme jako volání funkce translate na daný název.

Pro asociační property s isOwning = false není odvozeno jméno databázového elementu. Při transformaci asociačních property s atributem isOwning = true a upperBound != 1 na asociační tabulku je nutné zajistit unikátnost názvu této tabulky. Neexistuje předpoklad, že by byl název property unikátní ve všech třídách. Ale dvojice (table, property) musí být unikátní. Proto je název asociační tabulky získán jako spojení názvu vlastnické tabulky s názvem asociační property pomocí znaku "_".

$$owningTableName = translate(owningClassName)$$

$assocTableName = owningTableName$ + "_"+ $translate(associationPropertyName)$

Transformace primitivní property s upperBound != 1 (kolekce) na tabulku má podobný důsledek. Aby bylo pro vývojáře snadnější rozpoznat obraz třídy kolekce je název tabulky předražen prefixem "col_"

$$owningClstranslation = translate(owningClassName)$$

$$propertyTranslation = translate(primitivePropertyName)$$

$collectionTableName$ ="col_" + $owningClsTranslation$ + "_" + $propertyTranslation$

Názvy constrainů musí být unikátní v rámci celé databáze. Toto je zajištěno. Názvy unique constrainů jsou odvozeny od tabulky, nad kterou je constrain vytvořen. Existují dva atributy, které jsou mapovány na UniqueConstraint. Aplikační atribut kolekce isUnique je mapován na UniqueConstraint s jménem předraženým "ux_". Název constraintu pro atribut isOrdered je předražen řetězcem "ux_"a nakonec je doplněn řetězec "_ord". Vzhledem k unikátnosti dvojice (class, property) bylo využito mapování prefix spojený s přelozeným názvem třídy a přeloženým názvem property.

$$owningClstranslation = translate(owningClassName)$$

$$propertyTranslation = translate(primitivePropertyName)$$

$uxName$ ="ux_" + $owningClsTranslation$ + "_" + $propertyTranslation$
$ordName = uxName$ + "_ord"

Relace parent je do databáze transformována jako cizí klíč. Vzhledem k zákazu vícenásobné dědičnosti je každá relace parency nad tabulkou jednoznačně určena vzorem této tabulky. Tento klíč má proto jméno odvozeno od přeloženého názvu třídy předražené řetězcem "par_".

$parentFkName$ ="par_" + $translate(owningClassName)$

Tabulka kolekce referencuje vlastnickou tabulku právě jednou a název cizího klíče je tedy odvozen z názvu tabulky kolekce.

$fkCollectionName$ ="fk_" + $collectionTableName$

Poslední pravidlo vytváření cizích klíčů platí pro asociační tabulky. Každá asociační tabulka referencuje právě dvě tabulky. Název cizího klíče je vytvořen z názvu asociační tabulky a tabulky, kterou referencuje předražený řetězcem "fk_".

$$fkAssociationReference = "fk\_" + associationTableName + "\_" + \\ + referencedTableName$$

Tato pravidla nezajišťují unikátnost jmen elementů ve všech případech, ale zajišťují unikátnost jmen ve všech skupinách. Celkovou unikátnost jmen zajistí dobrý návrh aplikace a zkontroluje Databázová evoluce.

Kapitola 4

Řešení problému rozpoznávání operací

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 cílový.

4.1 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 surname;

        public Man(String name){
```

```
                this.name = name;
        }

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

Listing 4.3: Patch Man1 Man2

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

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.

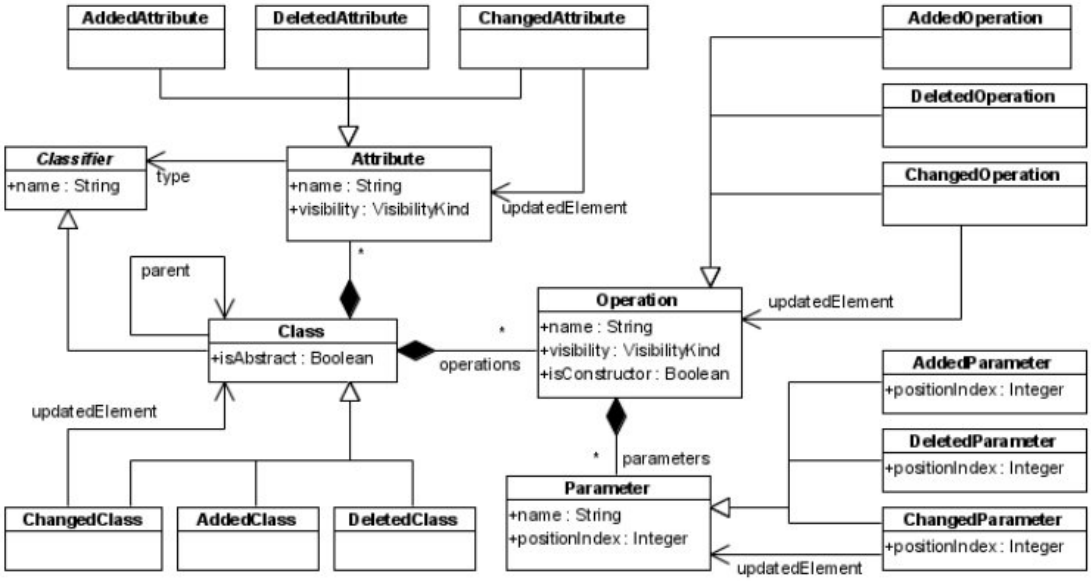
Delta notace je dle autora výhodná díky snadné rozložitelnosti velkých patchů na více menších patchů. Druhá výhoda je použitelná při paralelním vývoji. Pokud vznikne více verzí souboru v různých větvích, delta notace umožňuje snadné oddělení konfliktních operací od operací nezávislých.

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 a CollapseHierarchy naopak nepatří mezi modifikační, konstruktivní ani destruktivní operace. proto jsme mohli postupovat jednou z následujících cest

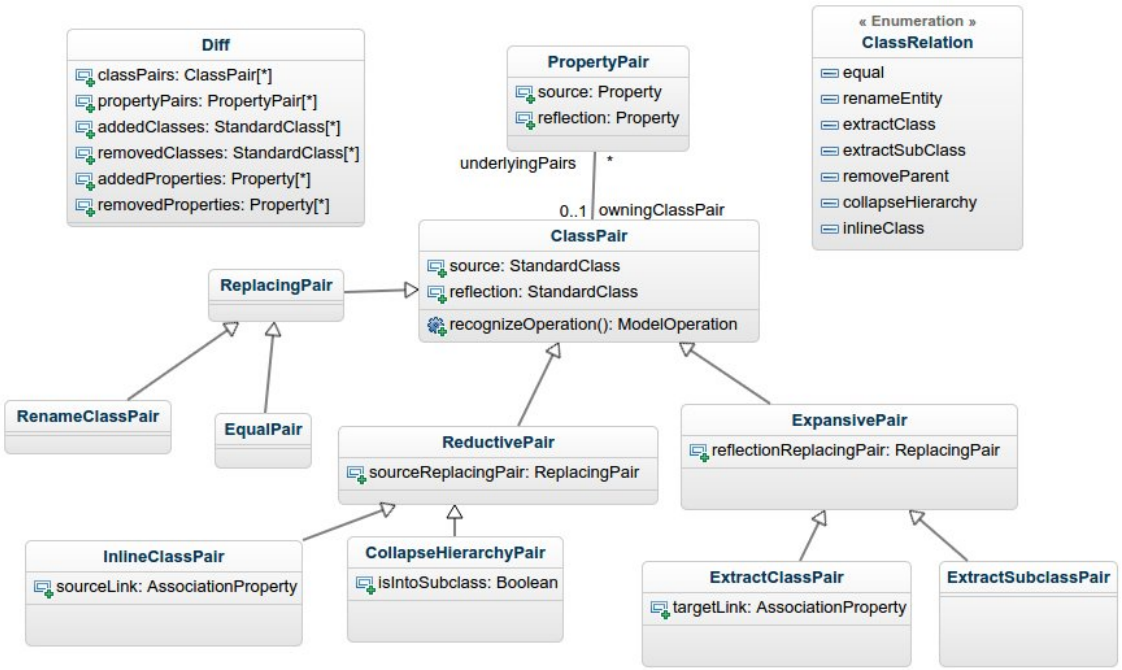
- získat standardní diff dvou modelů a aplikovat na něj algoritmus rozpoznávání jakožto sadu podmínek, které musí platit, aby algoritmus rozpoznal operaci.

- 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 definovaný v [Cic08]



Obrázek 4.2: Seznam diff elementů v projektu Migdb

4.2 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 permutability kolekce operací není znám.

4.3 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.

4.4 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].

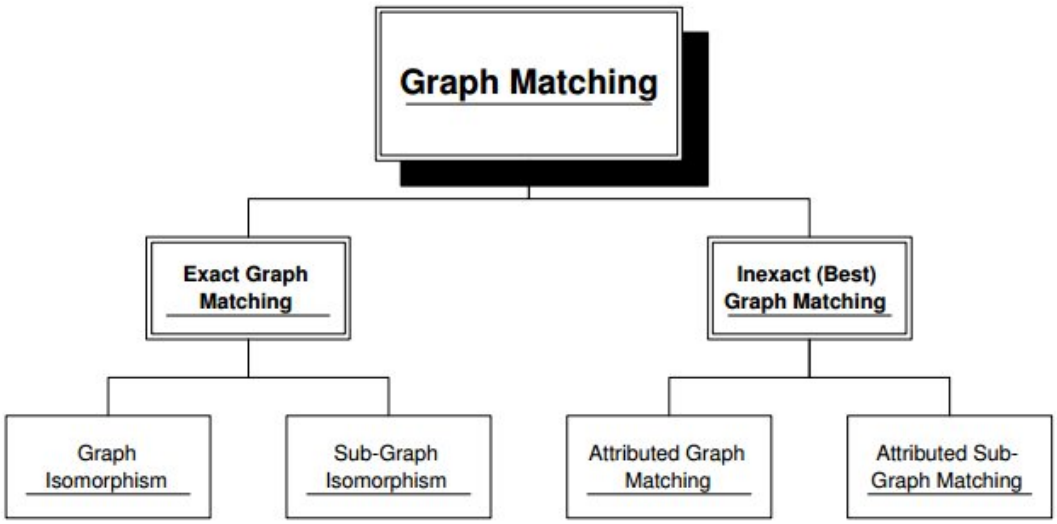
Algoritmus hledání přesné shody je definován následně: Mějme grafy $G_O = (V_O, E_O)$ a $G_V = (V_V, E_V)$, přičemž $\|V_O\| = \|V_V\|$, úkolem je potom najít takové prosté zobrazení $f : V_O \to V_V$, takové, že $(u, v) \in E_O$ iff $(f(u), f(v)) \in E_V$. Pokud takové mapování existuje, nazveme ho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

V rámci projektu Migdb nepracujeme se soubory, ale s modely. Každý model je možné textově reprezentovat. Pro jednoduchost si můžeme reprezentovat textově s následujícími pravidly:

- název třídy budeme reprezentovat velkými písmeny
- 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 $M_1(Abcd, Bef)$ odpovídající této reprezentaci je zobrazen na obrázku 5.1. Na obrázku je slovem undefined znázorněno, že Jednoduchá textová reprezentace neposkytuje žádné informace o typu. Pro reprezentaci odpovídající modelu typ properties tedy neznáme, v textu je vždy uvedena diskuze ohledně typu.

Obrázek 5.1: Model M_1 v UML

5.2 Migdb textový Diff

Pro snadné, rychlé a jednoznačné zobrazení rozdílů mezi modely jsem vytvořil textovou notaci, která je inspirována výstupem programu diff uvedeným v sekci 4.1. Pro $M_1(Abcd, Bef)$ definovaný v minulé sekci a $M_2(Abg, Cef)$ je textový diff

Listing 5.1: Textový diff modelů M_1 a M_2

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

V Migdb diffu vyjadřujeme řádkem +X pro přidaní třídy X do vstupního modelu modelu, +Xyz pro přidané properties "y"a "z"ze třídy X do vstupního modelu, -X pro odstranění třídy ze vstupního modelu a -Xyz pro odstraněné atributy "y"a "z"ze třídy X z vstupního modelu. Tj. do modelu M_1 byla přidána třída C, její property "e"a "f"a do třídy A byla přidána property "g". Naopak ze třídy A byly odstraněny property "c"a "d"a třída B byla odstraněna i se svými property "e"a "f".

5.3 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 1

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 algoritmus se snaží najít operaci, která ho nejblíže přibližuje k cílovému modelu. Algoritmus pro každou třídu operací nalezne hodnotu zlepšení, o kterou daná třída přibližuje současný model k modelu cílovému. Třídou operace rozumíme AddClass, RemoveEntity, InlineClass bez konkrétních parametrů operace. Pro vítěznou třídu operací algoritmus najde vhodné parametry a vloží konkrétní operaci i s jejími parametry do množiny R_{ops}. Algoritmus končí, pokud není rozpoznána žádná operace. Toho může být dosáženo, pokud je aktuální model shodný s cílovým či algoritmus nedokáže rozpoznat viz diskuze v sekci .

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.

5.3.1 Energie modelu

Do základní formy energie jsou započítány pouze identifikátory properties a tříd. Ostatní atributy tříd a jejich properties jsou zanedbány. Základní forma energie je definovaná jako součet existujících tříd a jejich properties.

Mějme model M_1(Abcd, Bef). 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. Základní výpočet energie je zobrazen na rovnici (5.1).

$$E(M_1) = \sum_{C \in M_1} E(C) = \sum_{C \in M_1} (1 + \sum_{P \in C} 1) \tag{5.1}$$

Algorithm 1 Algoritmus procházení stavů

Input: Zdrojový model S, cílový model T
Output: Seznam operací R_{ops}, po jejichž aplikaci se model S transformuje 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 c_{op} from Ops **do** ▷ Pro každou třídu operací z app metamodelu
        $K_{op} \leftarrow getImprovement(c_{op}, A, T)$ ▷ 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

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.

Model M_1 má podle rovnice (5.1) energii rovnu hodnotě 7 viz výpočet 5.2.

$$E(M_1) = E(A) + E(B) = 1 + \sum_{p \in A} E(p) + 1 + \sum_{p \in B}(E(p)) = 1 + 3 + 1 + 2 = 7 \tag{5.2}$$

5.3.2 Výpočet rozdílu

Pro tento minimalistický přístup můžeme použít výpočet symetrické delty vzorce (5.3). Vzdálenost dvou modelů M_1, M_2 je rovna energii, množinového rozdílu M_1, M_2, sečtené s energií množinového rozdílu M_2, M_1 definovaného v (5.4).

$$distance(M_1, M_2) = E(\Delta(M_1, M_2)) + E(\Delta(M_2, M_1)) \tag{5.3}$$

$$\Delta(M_X, M_Y) = (\Delta C_{XY}, \Delta P_{XY}) \tag{5.4}$$

Rozdíl modelů X a Y je uspořádaná dvojice rozdílu tříd a rozdílu properties viz (5.4).

$$\Delta C_{XY} = \{c : c \in M_X \wedge c \notin M_Y\} \tag{5.5}$$

Rozdíl tříd modelů X a Y je množina tříd, které jsou v modelu X, ale nejsou v modelu Y viz (5.5).

$$\Delta P_{XY} = \{p : p \in X \wedge p \notin Y \wedge p.owner \notin \Delta C_{XY}\} \tag{5.6}$$

Rozdíl 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 viz (5.6).

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

Rovnice (5.3) se nám tedy přepíše na (5.7).

$$\begin{aligned} distance(M_X, M_Y) =& E(\Delta(M_X, M_Y)) + E(\Delta(M_Y, M_X)) = E(C_{XY}, P_{XY}) + E(C_{YX}, P_{YX}) = \\ =& E(C_{XY}) + E(P_{XY}) + E(C_{YX}) + E(P_{YX}) \end{aligned} \tag{5.7}$$

Na výpisu 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:

$$distance(M_1, M_2) = E(C_{12}) + E(P_{12}) + E(C_{21}) + E(P_{21}) = 1 + 3 + 1 + 4 = 9$$

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 jen jednou. Proto jsou primitivní atributy vyčleněny z energie property. Původní vzorec pro vzdálenost dvou modelů se tedy změní na (5.8).

$$\begin{aligned} distance(M_X, M_Y) =& E(\Delta(M_X, M_Y)) + E(\Delta(M_Y, M_X)) = E(C_{XY}, P_{XY}) + E(C_{YX}, P_{YX}) + \\ &+ E(PRIM_{XY}) = E(C_{XY}) + E(P_{XY}) + E(C_{YX}) + E(P_{YX}) + \\ &+ E(PRIM_{XY}) \end{aligned} \tag{5.8}$$

Přičemž $E(PRIM_{XY})$ reprezentuje energii změněných primitivních atributů a γ reprezentuje množinu primitivních atributů splňující podmínku pro započítání atributů do energie viz (5.9). Výpočet $E(PRIM_{XY})$ je zobrazen na rovnici (5.10). Hodnota primitivního atributu A_γ je započítána do součtu (5.10) pokud vlastník atributu (property) není obsažen v modelu X či modelu Y nebo se hodnoty atributů v modelech X a Y liší.

$$\gamma = A : ((A.owner \notin M_X \vee A.owner \notin M_Y) \vee A_X! = A_Y) \wedge \ isPrimitive(A) \tag{5.9}$$

$$E(PRIM_{XY}) = \sum_{\gamma} 1 \tag{5.10}$$

Pro rozšířený výpočet energie property $p_{XY} \in P_{XY}$ je zakomponována do vzorce energie referenčních properties viz vzorec 5.12. Podmínka β zajišťuje, že neprimitivní atribut je do energie property započítán, pokud má nastavenu hodnotu null v modelu Y nebo se jeho hodnoty v modelech X a Y nerovnají nebo vlastnická property atributu není v modelu Y.

$$\beta = A : A \in p_X \wedge \ (A_Y = null \ \vee \ A_Y! = A_X \vee A_Y.owner \notin Y) \wedge !isPrimitive(A) \tag{5.11}$$

$$E(p_{XY}) = 1 + \sum_{\beta} 1 \tag{5.12}$$

5.3.3 Význam energie

Výpočet energie definovaný podle vzorců (5.8) - (5.12) udávají přesný počet konstruktivních destruktivních a modifikačních operací nutných k transformaci modelu M_X na model M_Y.

Pokud bude vzdálenost modelů $M_1 a M_2$ rovna X, tak je možné pomocí X operací přejít z modelu $M_1 do model u M_2$

Výpočet definovaný na vzorci (5.1) udává počet konstruktivních, destruktivních a modifikačních konstrukci operací nutných k vytvoření z prázdného modelu. Pokud bude mít energie hodnotu X, je zaručeno, že maximálně X operacemi je model zkonstruovatelný.

Z toho vyplývá, že $E(M_1) = distance(EMPTY, M_1)$, přičemž EMPTY je prázdný model. Dále vzhledem k symetričnosti výpočtu vzdálenosti platí $distance(M_1, M_2) = distance(M_2, M_1)$. Při substituci $M_1 = EMPTY$ nám energie udává tedy počet operací nutných k odstranění modelu.

Pro výpočty energie definované podle vzorců (5.8) - (5.12) je energie přesný počet kostruktivních, destruktivních a modifikačních operací. V subsekci 5.3.6 je zavedena konfigurace vah atributů. Po zavedení konfigurace vah atributů je energie horní hranicí minimálního počtu konstruktivních, destruktivních a modifikačních operací.

5.3.4 Zlepšující krok

Jak bylo řečeno, energie udává počet konstruktivních, destruktivních a modifikačních operací nutných k konstrukci či destruktci modelu a vzdálenost modelů $M_1 a M_2$ udává počet konstruktivních, destruktivních a modifikačních operací nutných k přechodu od modelu M_1 k modelu M_2. Vývojář může přiblížit model aplikací konstruktivních, destruktivních a modifikační operace mohou přiblížit model o 1 jednotku energie. Kromě konstruktivních, destruktivních a modifikačních operací existují i operace vyšší. Abychom určili efektivitu aplikace těchto operací, definujeme si tzv. zlepšující krok (improvement). V průběhu vývoje je možné aplikovat operaci ze stejné třídy s jinými parametry. Proto algoritmus hledá nejlepší zlepšující krok pro danou třídu operací viz řádek 7.

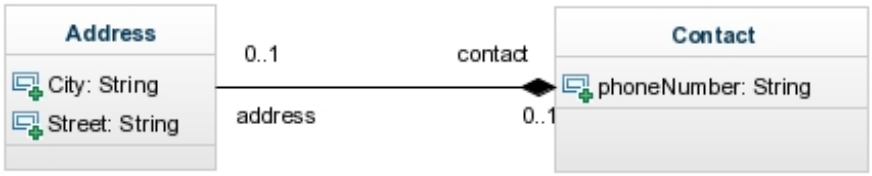
Zlepšující krok operace je definován jako vzdálenost, o kterou se přiblíží aktuální model A po aplikaci operace op k modelu cílovému T s nejlepšími možnými parametry. Nejlepší parametry jsou takové, které přibližují aktuální model o takovou vzdálenost, že neexistují parametry, které by zdrojový model přibližovaly k cílovému o vzdálenost větší. Není zaručena jedinečnost nejlepších parametrů. Při neexistenci jakýchkoliv vhodných parametrů algoritmus vrátí pro operaci hodnotu $-\infty$. Pro zlepšující krok getImprovement(c_{op}, A, T)$plat$(5.13).

$$distance(M_A, M_T) = distance(M_X, M_T) + getImprovement(op, A, \qquad (5.13)$$

5.3.5 Ilustrativní příklad běhu algoritmu

Ilustrujme běh algoritmu na příkladě. Na obrázku 5.2 je zobrazen ukázkový zdrojový a cílový model je zobrazený na obrázku 5.3.

Obrázek 5.2: Počáteční model



Obrázek 5.3: 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 5.2 a cílový model zobrazený na obrázku 5.3. 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.

5.3.6 Konfigurace vah

Stavový 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 5.3.6.

- 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:

- W(classname) = 1
- W(parent) = 0
- W(propertyName) = 1
- W(type) = 0
- W(isOrdered) = 0
- W(isUnique) = 0
- W(LowerBound, UpperBound) = 0
- W(oppositteProperty) = 0

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

5.3.7 Personalizovatelnost

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.

5.3.8 Zhodnocení

Přes počáteční slibné výsledky testované na konstruktivních operacích nebyl tento algoritmus shledán jako příliš efektivní. První nevýhodou je, že algoritmus není časově příliš optimální. 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 počet (či v modifikaci zmiňovaný počet sekvencí 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. Tyto operace je nynější implementace schopna rozpoznat.

5.4 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 4.3 bere hlavně poznámku u algoritmu UMLDiff zmiňovaného pod Custom Language skupinou 4.3 - ze všech praktických důvodů považujeme třídy se stejným jménem jako shodné.

Vznikly dvě implementace párovacích algoritmů.

5.5 Základní párovací algoritmus

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

Základní algoritmus pracuje podle pseudokódu 2

Algorithm 2 Základní párovací algoritmus

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

```
1: Inicializace:
2: SClses ← S.classes                          ▷ Nespárované třídy z vstupního modelu
3: TClses ← T.classes                          ▷ Nespárované třídy z cílového modelu
4: SProps ← S.classes.properties               ▷ Nespárované properties z vstupního modelu
5: TProps ← T.classes.properties               ▷ Nespárované properties z cílového modelu
6: R_ops ← {}

7: MATCH_CLS_BY_NAME()
8: R_ops= RECOGNIZE_OPS()                      ▷ Rozpoznání operací
9: END_ALGORITHM

10: procedure MATCH_CLS_BY_NAME
11:     for all C_S ∈ S.classes do
12:         for all C_T ∈ T.classes do
13:             if C_T.name = C_S.name then
14:                 MATCH_EQUAL_PROPS(S, T)    ▷ Aktualizace nespárovaných properties
15:                 SClses ← SClses \ {C_S}    ▷ Aktualnizace nenamatchovaných tříd ze zdrojového modelu
16:                 TClses ← TClses \ {C_T}▷ Aktualizace nenamatchovaných tříd z cílového modelu
17:             end if
18:         end for
19:     end for
20: end procedure

21: procedure MATCH_EQUAL_PROPS(clsSrcModel, clsTargetModel))
22:     for all P_S ∈ clsSrcModel.properties do
23:         for all P_T ∈ clsTargetModel.properties do
24:             if P_S.name = P_T.name then
25:                 SProps ← SProps \ {P_S}
26:                 TProps ← TProps \ {P_T}
27:             end if
28:         end for
29:     end for
30: end procedure
```

Algoritmus inicializuje množiny nespárovaných tříd z vstupního (a výstupního) modelu všemi třídami z vstupního (resp výstupního modelu) a seznam výstupních operací prázdnou množinou. Následně algoritmus zpáruje třídy podle jména, odstraní spárované třídy z množin SClasses a TClasses a zpáruje jejich property podle jména v proceduře MATCH_EQUAL_PROPERTIES. Procedura MATCH_EQUAL_PROPERTIES odstraní property ze spárovaných tříd se shodným jménem z množin SProps a TProps.

Algoritmus pokračuje rozpoznáváním operací. V fázi rozpoznávání operací jsou již spárovány všechny třídy, které základní párovací algoritmus dokáže rozpoznat.

Základní algoritmus rozpoznává operace v pořadí takovém, aby zajistil nejdříve existenci správné třídy pro přesouvané a odstraňované properties. Následně algoritmus modifikuje třídy modifikačními operacemi. Tato úprava musí být provedena před prací s properties, protože algoritmus v některých modifikačních operacích upravuje množiny SProps a TProps. Následně algoritmus upraví properties modifikačními operacemi pro properties, aby dostal properties do cílového stavu nebo do stavu, kdy je možné je z modelu odstranit. Po úpravě properties odstraní přebytečné properties obsažené v kolekci SProps. V posledním kroku algoritmus smaže odstraněné třídy. Tento základní proces je zobrazen na aktivity diagramu 5.4. Diagram má téměř lineární formu a rozpoznání jednotlivých skupin operací má tedy pro základní algoritmus v zásadě jednoznačné pořadí.

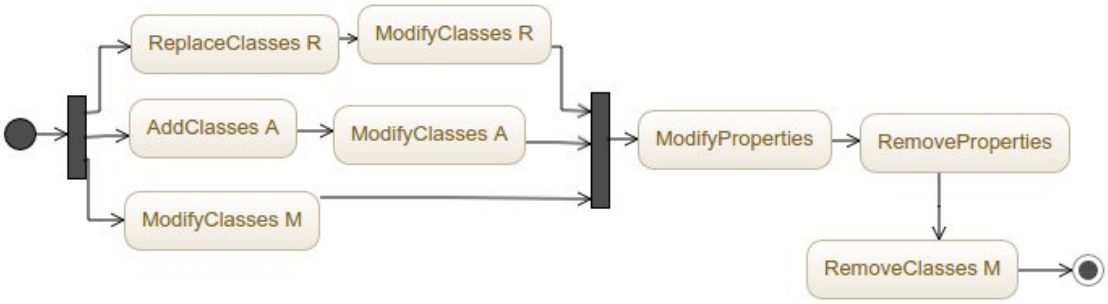
Obrázek 5.4: Rozpoznávání operací pořadí

Základní párovací algoritmus dělí třídy do tří skupin:

1. třídy, které byly v průběhu Evoluce modelu zachovány
2. třídy, které byly v průběhu Evoluce z modelu odstraněny
3. třídy, které byly v průběhu Evoluce do modelu přidány

Základní algoritmus předpokládá, že s třídami z první skupiny bylo manipulováno pouze za pomoci třídních modifikačních operací a s properties těchto tříd mohlo být manipulováno pomocí konstruktivních, destruktivních nebo modifikačních operací. Třídy z druhé skupiny byly z modelu odstraněny pomocí operace RemoveEntity. Aby byly validační podmínky destruktivních operací platné, je nutné před aplikací destruktivních operací RemoveEntity odstranit přebytečné properties operací RemoveProperty a upravit třídy pomocí třídních modifikačních operací RemoveParent. Třídy v třetí skupině je nutné do modelu přidat konstruktivní operací AddClass, následně je možné s třídami manipulovat pomocí třídních modify operací a modifikovat properties pomocí konstruktivních, destruktivních a modifikačních operací. Jediné modifikační třídní operace, které mají vliv na strukturu modelu a byly zařazeny do rozpoznávání jsou operace AddParent a RemoveParent. Operace AddParent může v nynější implementaci odstraňovat z aktuálního modelu kolizní properties, má tedy vliv na

seznam nenamapovaných properties SProps a musí předcházet operaci RemoveProperty.



Obrázek 5.5: Rozpoznávání operací pořadí, více tříd

Algoritmus by mohl rozpoznat operace v libovolné přípustné sekvenci. Tato přípustná sekvence operací musí splňovat precedenční vztahy zobrazené na obr. REF DOPLNěného obr. Každý model je možné rozdělit na množinu nezávislých hierarchických strukturálních operací. Dvě třídy A, B patří pro základní párovací algoritmus do stejné hierarchické struktury pokud $A.parent = B \vee B.parent = A \vee (\exists X \in M.clses : X \in ancestor(A) \wedge X \in ancestor(B))$, přičemž ancestor(Z) je množina všech předků třídy Z. Sekvence operací $op_1, op_2, \ldots op_n$ je nezávislá, pokud všechny operace z vstupního modelu operují nad třídami třídami z různých hierarchických struktur a žádná z operací op není AddParent.

Pro každou skupinu množinu operací $\{op_1, op_2, \ldots op_n\}$ můžeme najít grupu operací - tj seznam operací operujících nad jedinou hierarchickou strukturou či více hierarchickými strukturami spojenými sjednocenými operací AddParent.

Nejjednodušší implementací zisku validní sekvence je předpoklad existence jediné grupy, tj rozpoznávání operací po skupinách ze seznamu ref algo:match:ops_seq. Tato implementace je zobrazena v pseudokódu 3

Základní rozpoznávací algoritmus má schopnost rozeznat jen konstruktivní, destruktivní a modifikační operace. Zachovává data, jejichž identifikátor se nemění. Ostatní data naopak nekompromisně maže. Tento algoritmus není schopný rozpoznat operace RenameClass, RenameProperty, které jsou dle [Luk13] nejčastěji používanými operacemi. Tento algoritmus není schopný rozpoznat ani žádné expanzivní a reduktivní třídní operace. Výhodou implementace tohoto algoritmu je, že nepotřebuje definovat žádný diff metamodel. Za předpokladu $C_S \gg P_S$ *a* $C_T \gg P_T$ *a* $C_S \approx C_T = c$ má tento algoritmus složitost $O(c^2)$, protože prochází všechny třídy v zdrojovém modelu a hledá pro ně třídu se stejným jménem v modelu cílovém.

Algorithm 3 Rozpoznání operací v základním algoritmu

```
1: procedure RECOGNIZE_OPS
2:     for all C ∈ TClses do
3:         ops ← R_ops ⋃ AddCls(C.name)
4:     end for

5:     for all C ∈ TClses do                ▷ Oprava parency 1.AddParent pro přidané
   Neimplementováno
6:         if C.parent ≠ NULL then
7:             ops ← R_ops ⋃ AddParent
8:             for all P ∈ C.propertiesandP ∈ SProps do
9:                 SProps ← SProps \ P
10:            end for
11:        end if
12:    end for

13:    for all C ∈ SClses do                ▷ Oprava parency 2. RemoveParent pro smazané.
   Neimplementováno
14:        if C.parent ≠ NULL then
15:            ops ← R_ops ⋃ RemoveParent
16:            for all P ∈ C.propertiesandP ∈ C.parent.properties do
17:                TProps ← TProps ⋃ P
18:            end for
19:        end if
20:    end for

21:    for all C_T ∈ T.Classes \ TClses do      ▷ Oprava parency 3 a 4 pro namatchované
22:        for all C_S ∈ S.Classes \ SClses do
23:            if C_S.name = C_T.name ∧ C_S.parent = NULL ∧ C_T.parent! = null then
24:                ops ← R_ops ⋃ AddParent
25:                for all P ∈ C.propertiesandP ∈ SProps do
26:                    SProps ← SProps \ P
27:                end for
28:            end if
29:        end for
30:    end for

31:    for all P ∈ TProps do
32:        ops ← R_ops ⋃ AddProperty(P.owner, P.name)
33:    end for

34:    for all P ∈ SProps do
35:        ops ← R_ops ⋃ RemoveProperty(P.owner, P.name)
36:    end for

37:    for all C ∈ SClses do
38:        ops ← R_ops ⋃ RemoveCls(C.name)
39:    end for

40:    ADD_MODIF_PROP_OPS()
41: end procedure
```

5.6 Rozšířený algoritmus

Složitější implementace algoritmu páruje stejně jako jednodušší v první fázi shodné elementy - modely se mění, ale některé třídy jsou zachovány.

Algoritmus předpokládá, že napárované elementy tvoří tzv. páry. Každá třída ze zdrojového modelu může být nahrazena, expandována nebo redukována.

Třída je nahrazena (tvoří replacing pár), pokud je její jméno změněno či je třída zachována. Třída je zachována, pokud v cílovém modelu existuje třída se stejným jménem. Algoritmus zavrhuje sekvenci operací mazající všechny properties této třídy, smazání třídy a opětovnému vytvoření této třídy s jejich properties. Problém hledání RenamePair bude diskutován později. Každá třída může být součástí maximálně jednoho replacing páru. Je tedy zakázáno, aby v byla třída součástí dvou replacing párů, jednoho replacing a jednoho rename páru či dvou rename párů.

Redukované a expandované páry jsou navázány na právě jeden replacing pár.

Třída tvoří redukovaný pár, pokud existuje v zdrojovém modelu, ale neexistuje v modelu cílovém a je podobná třídě v cílovém modelu, která je již součástí replacing páru.

Třída tvoří expandovaný pár, pokud neexistuje v zdrojovém modelu, ale existuje v modelu cílovém a je podobná třídě ze zdrojového modelu, která je již součástí replacing páru.

Oproti základní verzi algoritmu je nutné vytvořit entitu reprezentující shodu - EqualClassPair. Shodné třídy a třídy přejmenované potom tvoří pilíře pro operace reduktivní a expanzivní, které se vážou na rozpoznané páry. Operace expanzivní a reduktivní není tedy možné bez jejich pilířů rozpoznat. V druhé fázi algoritmus páruje již spárované třídy v zdrojovém modelu s třídami nespárovanými z cílového modelu a obráceně třídy nespárované ze zdrojového modelu s třídami spárovanými. V druhé fázi párování musí algoritmus párovat třídy podle jiného kritéria. Tímto mnou zvoleným kritériem je suma properties se shodným jménem. Čím vyšší je tento koeficient, tím jsou si třídy podobnější.

Algoritmus počítá podobnost tříd pomoci počtu properties se stejným jménem. Další možností by byla definice energetické funkce viz sekce 5.3. Aby bylo snazší rozpoznat operace reduktivní a expanzivní, byly zavedeny ClassPairy a PropertyPairy.

5.6.1 Diff elementy

Kvůli nutnosti rozpoznávat operace vznikly v aplikačním metamodelu nové elementy. Základní párovací algoritmus si nepotřebuje uchovávat páry, třídy jsou spárovány jen na základě shodnosti jména, takže je kdykoliv v průběhu algoritmu tento seznam i jednotlivé páry možné získat. Na druhé straně rozšířený párovací algoritmus může rozpoznat operaci i RenameEntity a její vstupní a výstupní třídu tvořící pár není možné snadno identifikovat. Rozšířený párovací algoritmus rozpoznává operace závislé na dvou podobnostech, proto je nutné vytvořit 2 páry. První pár určí, která třída byla zachována, či měla změněné jméno. Druhý pár potom pomoci podobnosti určí, jestli

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 spárované 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. Je s nimi v rámci rozšířeného algoritmu zacházeno stejně jako v základním párovacím algoritmu. Konstruktivni ani destruktivní operace nejsou reprezentovány 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 a reduktivní a modifikační v Diff modelu zobrazeny jako ExpansiveClassPair a ReductiveClassPair, které mapují element vstupního modelu na element cílového modelu.

Aby bylo možné rozpoznat specifický pár závislý na jiném páru je nutné při nejd, 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 konstruktivních a destruktivních operací jsou elementy DiffClass a DiffProperty, které zaobalují třídy a property tak, aby bylo možné referencovat na jiný objekt než element Structure.

5.6.2 Popis rozšířeného párovacího algoritmu

Pseudokód rozšířeného algoritmu je zobrazen na 4

Algoritmus potřebuje kromě vstupních dat přenesených z původního algoritmu i počet matchovacích fází, které budou v rámci podobnosti provedeny.

V inicializační fázi stejně jako základní párovací algoritmus nainicializuje množiny nespárovaných tříd SClses a TClses, množiny nespárovaných properties SProps a TProps a seznam rozpoznaných operací R_{ops}. Oproti základnímu algoritmu musí nainicializovat i množinu spárovaných tříd CPairs a množinu spárovaných properties PrPairs. Párem p properties je uspořádaná dvojice (P_S, P_T) , kde P_S je property ze zdrojového modelu a P_T je property z modelu cílového, třídě P_S v páru p. Navzdory názvu pár naznačujícího dvojici elementů, je CPair uspořádaná čtveřice $(C_S, C_T, Pairs, owningPair)$, kde C_S je třída ze vstupního modelu, C_T je třída z cílového modelu, třídě C_S v páru říkejme zdrojová třída, třídě C_T potom obrazová třída. Pairs je množina párů, které náleží danému spárování, dá se říci, že Pairs je množina párů podobnosti signalizující spárování tříd, která je využita v dalších fázích algoritmu. OwningPair je nepovinná složka - představuje referenci na ReplacingPair u Reduktivních a Expanzivních párů a je použita v dalších fázích algoritmu. Základní párovací algoritmus nepotřeboval zaznamenávat páry, protože sprárované třídy byly jednoznačně identifikovány svým názvem.

V základní kostře algoritmu se rozšířená verze párovacího algoritmu příliš neliší od základní verze. Algoritmus po inicializaci provede párování tříd podle jména a párování tříd

podle podobnosti a zakončí svůj běh rozpoznáním operací.

V fázi matchování tříd podle jména jsou třídy ze zdrojového modelu párovány s třídami z cílového modelu na základě společného jména. Oproti fázi matchování v základním algoritmu zobrazeném na 2 je v fázi párování podle jména nutné vytvořit pár tříd - viz řádek 17. Následuje rozpoznání shodných properties v proceduře MATCH_EQUAL_PROPS viz. řádek 27. V této proceduře jsou oproti základnímu algoritmu vytvořeny shodné páry properties. Shodné páry jsou přidány do čtveřice Cpair, s nastaveným owningPairem NULL.

Pseudokód 5 ukazuje průběh párovací fáze za pomoci podobnosti. V prvním jejím kroku je nainicializován seznam kandidátů podobnosti pro každou třídu. Třída C_T je podobná třídě zdrojové třídě C_S, pokud existuje nejméně jedna property ve třídě C_S, která má shodné jméno jako property v třídě C_T. Množinu podobné tříd nazývejme množinu kandidátů k párování. Mohou nastat 4 situace pro zdrojovou třídu C_S a jejího kandidáta k párování C_T.

1. Pro zdrojovou třídu již existuje replacingPair,pro kandidáta podobnosti nikoliv.V tomto případě se algoritmus pokusí vytvořit expanzivní pár vytvářející novou třídu viz řádek 20 v pseudokódu 5

2. Pro zdrojovou třídu C_S i pro jejího kandidáta C_T existuje replacingPair viz řádek 14 v pseudokódu 5. Tato situace zahrnuje jednak spárování tříd v rámci replacing páru a také oddělené spárování v rámci modifying operací.

3. Pro zdrojovou třídu neexistuje replacingPair, ale pro kandidáta podobnosti existuje replacing pár. viz řádek 16 v pseudokódu 5. V této situaci se algoritmus pokusí spárovat třídy do reductivního páru.

4. Pro zdrojovou třídu ani pro kandidáta podobnosti neexistuje replacingPair. viz řádek 22 v pseudokódu 5. V této situaci se algoritmus pokusí spárovat třídy do RenamePáru páru. Tato situace může zakrývat situaci 1 či 3

Aby byla třída spárována do reduktivního nebo expanzívního páru, musí být splněny podmínky vycházející z aplikace dané reduktivní či expanzivní operace.

Pro ilustraci si představme, že máme zdrojový model $M_1 = (Abcde)$ a cílový model $M_2 = (Abcf, Bdeg)$. Diff těchto dvou modelů je zobrazen na výpisu 5.3

Listing 5.3: Textový diff pro expanzivní operaci

```
+B          −Ade
+Bdeg
+Af
```

Vidíme, že třída A byla zachována, třída B, třída B je podobná A s koeficientem podobnosti 2, existují 2 property("e"a "d") ve třídě A ve zdrojovém modelu se shodným jménem jako mají property v modelu cílovém ve třídě B. Třída B byla přidána do modelu a je podobná třídě A, algoritmus očekává rozpoznání expanzivní operace. Pokud je přidaná property "f"typu B a property "g"ve třídě B je typu A, algoritmus rozpozná operaci ExtractClass. Pokud v cílovém modelu platí B.parent = A, algoritmus rozpozná operaci ExtractSubClass. V případě nerozpoznaného typu páru algoritmus skončí s matchovací fází, protože neexistuje jiný kandidát pro třídu B na vytvoření páru a podobnost bude zanedbána.

V ilustračním příkladě 2 zaměníme M_1 za model M_2 v původním příkladu, tj. $M_2 = (Abcde)$ a $M_1 = (Abcf, Bdeg)$. Textový diff je zobrazen na výpisu 5.4

Listing 5.4: Textový diff pro reduktivní operaci

+Ade	–B
	−Bdeg
	−Af

Algoritmus pro tento příklad zjistí, že třída A byla zachována, třída B byla odstraněna, z třídy B byly odstraněny properties "d", "e"a "g", z třídy A byla odstraněna property f, naopak do třídy A byly přidány property "de". Třída B je podobná s koeficientem 2 (shodnost properties "d"a "e") třídě A. Třída B byla ze zdrojového modelu odstraněna, algoritmus prověřuje reduktivní operace. Pokud platí B.parent = A algoritmus rozpozná operaci CollapseHierarchy, v případě, že property f a g tvoří oboustraně navigabilní vazbu, algoritmus rozpozná operaci InlineClass. V případě, že algoritmus nerozpozná ani jeden z těchto párů bude podobnost zanedbána.

Algorithm 4 Rozšířený párovací algoritmus

Input: Zdrojový model S, cílový model T, C počet matchovacích fází
Output: Seznam operací R_{ops}, po jejichž aplikaci se model S transformuje 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 ze vstupního modelu
5: $TProps \leftarrow T.classes.properties$ ▷ Nenamatchované properties z cílového modelu
6: $R_{ops} \leftarrow \{\}$
7: $CPairs \leftarrow \{\}$
8: $PrPairs \leftarrow \{\}$

9: MATCH_CLSES_BY_NAME()
10: MATCH_CLSES_BY_SIMILARITY(C)
11: R_{ops}= RECOGNIZE_OPS() ▷ Rozpoznání operací
12: **END_ALGORITHM**

13: **procedure** MATCH_CLSES_BY_NAME
14: **for all** $C_S \in S.classes$ **do**
15: **for all** $C_T \in T.classes$ **do**
16: **if** $C_T.name = C_S.name$ **then**
17: cPair = markMatched(S, T, 'Equal') ▷ Vytvoření equal páru
18: pairs = MATCH_EQUAL_PROPS(S, T)
19: addUnderlyingPairs(cPair, pairs)
20: $CPairs = CPairs \bigcup cPair$ ▷ Aktualizace nespárovaných properties
21: $SClses \leftarrow SClses \setminus \{C_S\}$ ▷ Aktualnizace nenamatchovaných tříd ze zdrojového modelu
22: $TClses \leftarrow TClses \setminus \{C_T\}$▷ Aktualizace nenamatchovaných tříd z cílového modelu
23: **end if**
24: **end for**
25: **end for**
26: **end procedure**

27: **procedure** MATCH_EQUAL_PROPS(clsSrcModel, clsTargetModel))
28: Pairs = {}
29: **for all** $P_S \in clsSrcModel.properties$ **do**
30: **for all** $P_T \in clsTargetModel.properties$ **do**
31: **if** $P_S.name = P_T.name$ **then**
32: pair = markMatched(P_S, P_T, 'Equal')
33: $Pairs = Pairs \bigcup pair$
34: $SProps \leftarrow SProps \setminus \{P_S\}$
35: $TProps \leftarrow TProps \setminus \{P_T\}$
36: **end if**
37: **end for**
38: **end for**
39: **return** Pairs
40: **end procedure**

Algorithm 5 Matchování podle podobnosti rozšířeného algoritmu

```
1: procedure MATCH_CLSES_BY_SIMILARITY(iterCount)
2:     CSets = initSimilarityCandidates()                    ▷ mapa kandidátů CSets
3:     for i = 0 ; i < iterCount ; i++ do                    ▷ Proveď iterCount párovacích iterací
4:         for all cls ∈ CSets.keys() do
5:             candidates = CSets.get(cls)
6:             MATCH_CLS_BY_SIMILARITY(cls, candidates)
7:         end for
8:     end for
9: end procedure

10: procedure MATCH_CLS_BY_SIMILARITY(sourceCls, reflectionCandidates)
11:     for all candidate ∈ reflectionCandidates do
12:         if isReflectionReplacingPairRecognised(candidate) then
13:             if isSourceClassReplacingPairRecognised(sourceCls) then
14:                                 ▷ Nevzniká nový pár, je rozpoznána modifying operace ve fázi
    rozpoznávání operací
15:             else
16:                 matchReductivePair(sourceCls, candidate)
17:             end if
18:         else
19:             if isSourceClassReplacingPairRecognised(sourceCls) then
20:                 matchExpansivePair(sourceCls, candidate)
21:             else
22:                 matchRenamePair(sourceCls, candidate)
23:             end if
24:         end if
25:     end for
26: end procedure
```

Kapitola 6

Testování projektu Migdb

V průběhu vývoje byl za účelem ověření správné funkcionality vytvořen projekt Migdb.testing.run, v kterém jsou obsaženy soubory workflow jednotlivých modulů. Workflow (pracovní tok) je vstupní bod každé samostatně definované množiny testů. Workflow odpovídá JUnit TestSuite.

6.1 Jednotkové testy

V rámci projektu byly vytvořeny některé jednotkové testy, aby byla zaručena izolovaná správnost běhu jednotlivých komponent a jejich nahraditelnost.

6.1.1 Testy komponent

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 existují i negativní testy - běh některých testů musí skončit chybovou hláškou, jak už napovídá klíčové slovo fail v jejich názvu.

Druhou otestovanou komponentou je TestComponent. Tato komponenta vznikla, aby bylo přehlednější a efektivnější psát QVT testy Migdb, je složena z několika dalších načítacích, ukládacích a porovnávacích komponent. TestComponent zkracuje délku zápisu testů ve workflow asi desetkrát. Testem správné funkce je zalogování očekávaného výsledku testu (pozitivního či negativního) do konzole.

6.1.2 Testy Aplikační Evoluce

Aplikační Evoluce byla otestována pomocí workflow
test_app_atomic.mwe2 v balíčku migdb.testing.app.atomic.run. Balíček obsahuje 37 testů přípustných případů užití i případů, které mají zalogovat chybu (tedy případů, pro které metoda isValid dané operace vrací hodnotu false). Každý test má v oddělených souborech definován vstupní aplikační model a operaci, která se má nad tímto modelem provést. Výstupní modely po aplikaci operace a seznam zalogovaných chyb jsou porovnány s očekávaným výstupem definovaným v souborech předaných TestComponentě jako parametry qvtComparison. Vyhodnocení těchto testů je automatizováno.

6.1.3 Testy Databázové Evoluce

Testy Databázové Evoluce jsou obsažené ve workflow
test_rdb_atomic.mwe2 v balíčku migdb.testing.rdb.atomic.run. Balíček obsahuje 22 testů přípustných případů i případů, které mají zalogovat chybu (tedy případů, pro které metoda isValid dané operace vrací hodnotu false). Každý test má v oddělených souborech definován vstupní model databázové struktury a databázovou operaci, která se má na vstupní strukturu aplikovat. Výstupní modely po aplikaci databázové operace a seznam zalogovaných chyb jsou porovnány s očekávaným výstupem definovaným v souborech předaných TestComponentě jako parametry qvtComparison. Vyhodnocení těchto testů je automatizováno.

6.1.4 Testy ORM

Testy modulu ORM jsou obsažené ve workflow migb.testing.orm.run.tests_orm_structure v balíčku
migdb.testing.orm.run. Balíček obsahuje 5 testů mapování jednotlivých inheritanceType. TestComponent obsažené ve workflow dostávají vstupní aplikační model, na který aplikují ORM transformaci. Výstupní model databázové struktury a seznam zalogovaných chyb jsou porovnány s očekávanými výstupy definovanými v souborech předaných TestComponentě jako parametry qvtComparison. Vyhodnocení těchto testů je automatizováno.

6.1.5 Testy modulu ORMo

Testy modulu ORMo jsou obsažené ve workflow
migb.testing.orm.run.tests_orm v balíčku migdb.testing.orm.run. Balíček obsahuje 42 testů mapování aplikačních operací na operace databázové. TestComponent obsažené ve workflow dostávají vstupní aplikační model a aplikační operaci, která se má aplikovat na tento model. Výstupní seznam databázových operací a seznam zalogovaných chyb jsou porovnány s očekávanými výstupy definovanými v souborech předaných TestComponentě jako parametry qvtComparison. V průběhu testování neprobíhá Aplikační Evoluce, tyto testy předpokládají validnost vstupní aplikační operace nad daným vstupním aplikačním modelem. Vyhodnocení těchto testů je automatizováno.

6.1.6 Testy modulu OpsRecognition

6.2 Integrační Testy

6.2.1 Testy generátoru databázového schema

V databázi bylo vytvořeno workflow test_schema_generator.mwe2 v balíčku migdb.testing.generators.run sloužící k otestování generátoru databázového schema. Workflow obsahuje 41 testů. Každý případ užití transformuje vstupní aplikační model na model databázový, z kterého potom komponenta SchemaGenerator vygeneruje SQL skripty do souboru s příponou .sql. Vyhodnocení testu musí udělat buď vývojář zkontrolováním testu nebo tento test spustit nad PostgreSql databází a zkontrolovat, že se vytvořily správné databázové elementy.

6.2.2 Testy běhu celého frameworku

Testy funkčnosti celého frameworku jsou obsaženy ve workflow test_code_generator.mwe2. Vyhodnocení testů je poloautomatické. Po proběhnutí celého workflow jsou pro každý případ užití vygenerovány soubory tvaru XXXb_nazevTestu.sql, kde „XXX“ je prefix (číslo testu) a „b“ je varianta. Proběhnutí testovacího případu v rámci workflow zaručuje pouze, že nenastala žádná chyba v rámci tohoto testovacího případu. V podsložce schema každého případu užití je potom soubor schema.sql. Dále jsou v projektu migdb.testing.run uloženy ve složce test_data testovací data a ověřovací selecty k vygenerovaným složitějším případům užití manipulujícími s daty (prefixy 007-010). K otestování správné funkcionality je nutné mít nainstalovánu databázi PostgreSql. Postup otestování funkce testu je popsán v algoritmu 6

Algorithm 6 Postup otestování celého frameworku

```
1: Spuštění workflow test_code_generator.mwe2
2: for all testCase ∈ output − tests/gen_SQL do
3:     if testCase ma prefix > 007 then
4:         Aplikace soubor schema.sql na databází
5:         Aplikace soubor data.sql z odpovídající složky obsažené v složce test_data
6:     end if
7:     Aplikace sql soubor z složky testCasu na databázi
8:     if testCase ma prefix > 007 then
9:         Aplikace soubor Aplikuj soubor check_selects.sql z odpovídající složky obsažené
   v složce test_data
10:        Porovnání navrácené hodnoty z selectu s očekávaným výstupem
11:    end if
12:    Kontrola struktury databáze
13: end for
```

Kapitola 7

Závěr

7.1 Zhodnocení výsledků diplomové práce

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. Aplikačnímu modelu se věnovaly bakalářské práce [Tar12] a [Luk11], a diplomové práce [Tar14] a [Maz14]. Aplikační model je tedy celkem dobře popsaný a Aplikační Evoluce je plně otestovaná.

Modelem databázovým se věnovaly práce bakalářské práce [Luk11] a [Jez12]. Databázový metamodel je nyní dobře definovaný a Databázová Evoluce je dobře definovaná a plně otestovaná.

Modul ORMo se stal implementačně nejnáročnějším a teoreticky nejméně popsatelným. Počet transformací, testů a nutná specifikace všech případů užití je tak obsáhlá, že by se vešla do samostatné diplomové práce. Tento modul není plně implementován, vybral jsem si podmnožinu aplikačních operací, pro které jsem definoval transformaci na operace databázové. Tyto transformace jsou otestované viz. kapitola 6.

Projekt Migdb je nyní v funkční podobě, jeho nasazení brání jen nízký počet operací, které mají definováno ORMo mapování. Proto si myslím, že jsem úspěšně dokončil projekt Migdb.

Modul OpsRecognition se stal velice zajímavou částí této diplomové práce. Nepodařilo se mi definovat algoritmus pro rozpoznání všech operací, ale podařilo se mi prozkoumat základní principy použitelné při rozpoznávání operací. Tyto principy jsem shrnul v kapitole 4, naimplementoval zkušební implementace algoritmů viz. kapitola 5. Přes nízký počet rozpoznaných operací si troufám tvrdit, že navrhnuté algoritmy jdou správným směrem. Proto si myslím, že i cíl prozkoumání automatizace rozpoznávání operací jsem dosáhl.

7.2 Budoucí rozšiřitelnost

Prvním nutným směrem k úspěšnému nasazení projektu Migdb je dodefinování a otestování ORMo transformací zbylých aplikačních operací. Samotná definice operací aplikovatelných na databázový model není pevná, což vzhledem k malému vzorku sbíraných požadavků vede k nejednoznačným ORMo mapováním. Proto by v dalších pracích bylo dobré udělat operační výzkum a získat větší množství požadavků na tyto operace.

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) [W3C14a] a Ontology Web Language (OWL) [W3C14b]. Implementace sémantického rozpoznávání by mohla v budoucnu odhadnout změny nejen na základě syntaxe(struktury) aplikace, ale i na základě významu názvů jednotlivých entit v aplikaci. 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.

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.

7.2.1 Portabilita

Poslední otázkou, na kterou jsem neměl moc času hledat odpověď je portabilita projektu Migdb na jiné relační databáze. Projekt Migdb funguje nad databází PostgreSQL. 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, je možné zachovat modul mapování jmen definovaný v sekci 3.6.1. 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 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 nějakými dalšími znaky (například balíčkem). Tudíž délka 9 nemusí být maximální horní hranice jmen našich tříd a property, ale reálná maximální délka může být nižší.

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 a nahradit implementaci modulu mapování jmen popsanou v sekci 3.6.1.

Kapitola 8

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 9

Instalační a uživatelská příručka

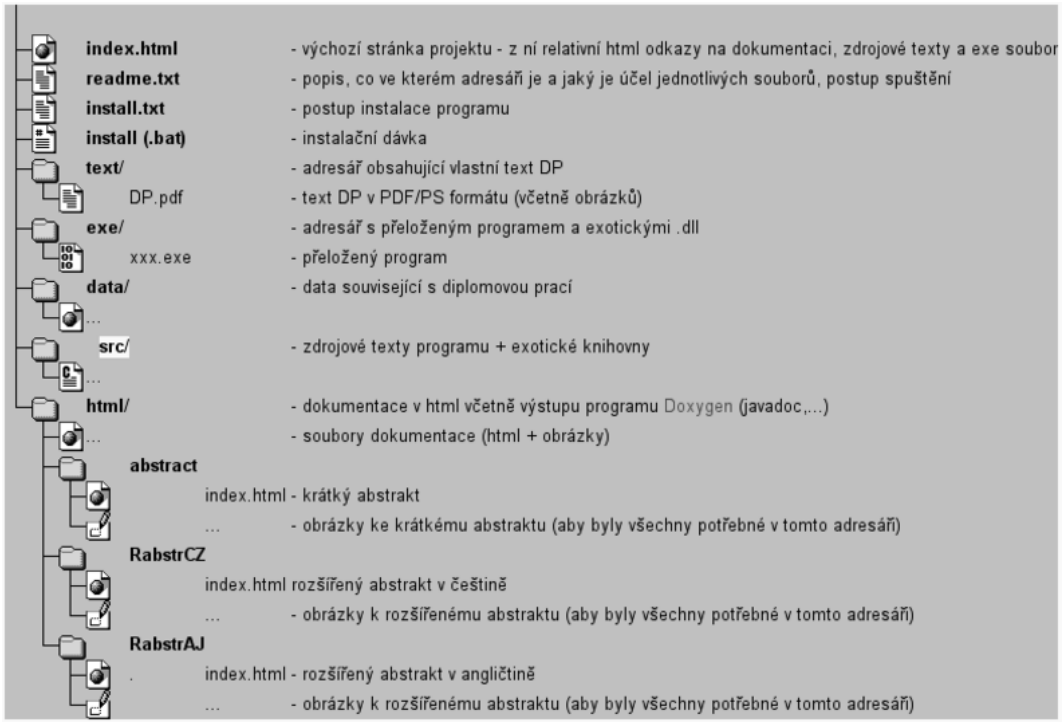
Tato příloha velmi žádoucí zejména u softwarových implementačních prací.

Kapitola 10

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 10.1: 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.

[Siq14] Fábio Levy Siqueira. Qvto tutorial. [online], Citováno 18.12.2014.

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

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

[W3C14a] W3C. Resource description framework. [online], Citováno 16.11.2014.

[W3C14b] W3C. Web ontology language. [online], Citováno 16.11.2014.

[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.