METODY TRANSFORMACE RELAČNÍCH DATABÁZOVÝCH SYSTÉMŮ DO OBJEKTOVÝCH

Doktorská disertační práce

Doktorand: Ing. Vít Holub

Školitel: Prof. RNDr. Jiří Vaníček, CSc.

Obor: Informační management

Česká zemědělská universita

2007

SEZNAM SCHÉMAT

SEZNAM TABULEK

OBSAH

1 Motivace

V souvislosti s organizačními změnami, růstem a dalším vývojem je řada firem postavena před rozhodnutí, zda a jak zohlednit nové podmínky a skutečnosti ve svých informačních systémech. Odpověď na první otázku bývá bez výjimky kladná – informační systém, který nereflektuje aktuální situaci a potřeby, nemá pro organizaci přidanou hodnotu. Druhá otázka již typickou odpověď nemá – způsob promítnutí změn do informačního systému závisí na **charakteru změn** a **stavu informačního systému**.

Typická podniková infrastruktura obsahuje stovky datových zdrojů, množství databázových systémů, desítky aplikací. Míra integrace je přitom různá. Pokud má být IS výrazněji inovován, je obvykle prováděn rozsáhlý reengineering (mj. s dopady propagovanými samozřejmě až na databázovou vrstvu). Vzhledem ke složitosti a náročnosti tohoto procesu je výhodné provést studii o vhodnosti použití dané databázové technologie. V některých případech může být výsledkem doporučení přejít ze stávajícího relačního DBMS na objektový.

Je třeba poznamenat, že takových případů v praxi přibývá. Charakter podnikových dat se v průběhu posledních let vývoje společnosti mění – převážně textové informace a data s relativně plochou strukturou jsou vytlačovány komplexními heterogenními objekty.

Pokud byl původní (velmi pravděpodobně objektový) systém vyvíjen podle zásad OOAD a byla řádně vedena dokumentace, jsou jednotlivé fáze redesignu značně usnadněny a není nezbytně nutné všemi etapami vývoje procházet znovu (pokud např. nedochází ke změnám na business úrovni). Často je ale situace opačná. Protože systém vznikal za nepříznivých podmínek a pro úspěch jeho vývoje byla určující zejména rychlost nasazení, z informací

popisující uložení a význam dat jsou k dispozici fyzický datový model a data samotná.

Zde spočívá hlavní praktická využitelnost autorovy disertační práce, tj. navržení interaktivního procesu zpětné tvorby konceptuálního modelu a automatizovaného sestavení objektového implementačního modelu. Metoda transformace dále usnadňuje i migraci dat. Předností této metody je získání relativně velkého množství vstupních informací ze současného systému a minimální množství potřebných uživatelských interakcí.

2 Struktura disertační práce

Tato kapitola popisuje a vysvětluje způsob členění práce. Snahou je také poskytnout čtenáři rychlý přehled v popisovaných oblastech a nastínit souvislosti mezi nimi.

2.1 Literární rešerše

Rešerše je obsahem kapitol 3 – 6.

Daná, úzce vymezená problematika zasahuje svými aspekty do více oblastí IT. Termín „metody transformace relačních databázových systémů do objektových“ svým významem implikuje zejména využití teoretických databázových znalostí, znalostí modelování a práce s různými úrovněmi modelů a v neposlední řadě principů databázového návrhu. Tyto základní a dobře popsané problematiky je nutné doplnit o reversní (datové) inženýrství, obor, který prochází neustálým vývojem a který v (pro tuto práci podstatné) specializaci reversního inženýrství relačních databází nedisponuje všeobecně uznávaným standardizovaným procesem. Teoretická část se tedy skládá ze 4 uvedených hlavních oddílů.

2.1.1 Databázová paradigmata

Počátečním stavem uvažovaného procesu transformace je relační databáze naplněná daty (tj. je dané schéma a stav relační databáze), cílovým pak stejná situace v objektové formě (tj. je odvozeno schéma objektové databáze a ta je uvedena do ekvivalentního stavu). Obě databázová paradigmata vycházení z vlastního **datového modelu**, mají formálně definované operace pro manipulaci s těmito daty (tyto operace tvoří **algebru**) a tyto operace jsou implementovány v příslušném **dotazovacím jazyce**.

Kromě výše uvedených jsou dále v textu stručně popsána i další databázová paradigmata.

2.1.2 Principy transformačního modelování

Termín transformační modelování není v literatuře používán a pro účely této práce se jím rozumí souhrn modelovacích předpisů a postupů, které zajišťují rozvrstvení komplexního modelu podle zvolených úrovní abstrakce a snadný, automatizovaný přechod mezi těmito vrstvami (viz dále).

Takové přístupy bývají definovány na obecné úrovni (tj. ne přímo pro konkrétní použití) nebo jsou snadno zobecnitelné. Přinášejí široce použitelné zásady a jejich společným znakem je snaha o zachycení maximálního množství sémantiky a umožnění jednoduchých a transparentních přechodů mezi modely.

2.1.3 Postupy databázového návrhu

Databázový návrh, vyjádřen v pojmech předchozích odstavců, je postupná transformace modelu nejvyšší úrovně abstrakce (konceptuální úroveň) do modelu nejnižší úrovně abstrakce (fyzická, datová úroveň). Cílový model je vyjádřen konstrukty platnými pro dané databázové paradigma.

Tyto postupy jsou využity při návrhu objektového datového modelu a studiu možností zpětné transformace relačního datového modelu.

2.1.4 Datové reversní inženýrství

Reversní inženýrství obecně slouží ke zpětnému odvození zadání z implementace zadání. Zadáním je zde chápána konceptuální úroveň modelu

a implementací fyzická úroveň relačního datového modelu. Jde o opačný proces k databázovému návrhu.

Tyto postupy jsou využity při vytváření konceptuálního modelu z relační implementace.

2.2 Vlastní práce

Vlastní práce je uvedena v kapitolách 7 – 9.

Je možné dotyčné modely transformovat přímo, tj. (1) bez interakcí a (2) zprostředkujících modelů? Po bližším seznámení s problematikou je patrné, že zcela automatizovaně ne, a přímo jen za cenu neuspokojivého výsledku a nečitelnosti celého procesu.

Konkrétní relační databáze obsahuje velmi málo metainformací. Objektová naopak zachycuje veškerou sémantiku statického business modelu vyjádřeného prostředky EER. Dodatečné popisné informace je možné získat několika způsoby, každý z nich má však svá úskalí a nevýhody.

Metoda popisovaná v práci předpokládá absenci doménového (konceptuálního) modelu a dalších artefaktů tvořících zadání k návrhu databázového modelu. Jak již bylo naznačeno, tento model je významný pro všechny etapy životního cyklu IS a nejen pro účely migrace systému je důležité ho sestavit. Existuje-li konceptuální model, existuje i možnost neformálního ověření správnosti datového modelu a pochopení problémové domény.

Proto byla zvolena 2fázová metoda transformace:

- vytvoření rozšířeného ER modelu (EER model) z relačního databázového modelu postupem opačným k databázovému návrhu[1]
- automatizované odvození objektového datového modelu z EER modelu

Transformace databázového systému z relačního do objektového je složitý proces, který se skládá z několika na sebe navazujících kroků. Ty je možné rozdělit do dvou skupin:

1. Aplikace zvolených transformačních metod na zdrojový relační datový model vedoucí k získání očištěného a normalizovaného relačního modelu.

    a. Import schématu relační databáze do struktur UML (meta)modelu. Tímto modelem je zde rozuměno rozšíření UML takovým způsobem, aby bylo možné do něj načíst a v něm interpretovat všechny relevantní struktury vyjádřené v SQL DDL.

    Tento krok je popsán v kapitole *Získání logického relačního datového modelu (PSM → PIM).*

    b. Uživatelská úprava takového modelu, jíž se načtený model konsoliduje (odstranění nedokonalostí plynoucí ze špatně provedeného návrhu relačního datového modelu a z úprav,

[1] Tento postup využívá jiné techniky a předpokládá uživatelskou interakci

které v modelu vznikaly dodatečně a měly negativní vliv na jeho kvalitu).

Tento krok je popsán v kapitole *Konsolidace (PIM → PIM)*.

c. Normalizace modelu ve smyslu automatické transformace struktur modelu do požadované normální formy. Tento krok odstraní nedokonalosti plynoucí z provedených výkonových optimalizací a významně zvýší kvalitu modelu a věrnost jeho popisu reality.

Tento krok je popsán v kapitole *Normalizace (PIM → PIM)*.

2. Vlastní transformace relačního datového modelu do objektového pomocí stejných transformačních metod.

a. Zpětná transformace relačního modelu do konceptuálního modelu, tedy modelu společného libovolné realizaci databázového stroje. Tato transformace je označena jako zpětná, protože v případě klasického postupu (od obchodních požadavků k implementaci) je prováděna opačně – zdrojový model je cílovým a naopak.

Tento krok je popsán v kapitole *Konstrukce konceptuálního modelu (PIM → CIM)*.

b. Transformace konceptuálního modelu do objektového logického datového modelu, tedy modelu společného libovolné realizaci objektového databázového stroje.

Tento krok je popsán v kapitole *Konstrukce objektového modelu (CIM → PIM)*.

c. Vytvoření fyzického objektového modelu vlastního konkrétní implementaci objektové databáze.

   Tento krok není v práci popsán, protože ta si klade za cíl vytvořit obecnou metodiku použitelnou nezávisle na zvoleném technickém řešení. Po výběru konkrétního databázového produktu je možné jednoduše model upravit a rozšířit tak, aby respektoval specifika daného ODBMS a obsahoval údaje nízké úrovně (fyzické uložení, clustering, atd.).

3 Databázové systémy

Databázový systém (DS) je pojem, který může být v různých souvislostech chápán odlišně. V některých zdrojích je takto označován software pro řízení báze dat (DataBase Management System – DBMS), jinde jsou zahrnuta i data samotná (DataBase – DB). Vzhledem k tomu, že k pochopení konkrétního databázového systému, je třeba popsat základní teoretická východiska, je zde tento pojem chápán spíše jako paradigma příslušné dané technologii.

3.1 Databázové systémy první generace

Takto jsou označovány databázové systémy, jejichž model vývojově předcházel relačním a později objektovým modelům. Jejich společným znakem je provoz na (z dnešního pohledu) málo výkonných strojích pracujících s úložišti na magnetických páskách.

3.1.1 Počátky vývoje DBMS

Společnost IBM pro své sálové počítače vyvíjela souborový systém FFS (Formatted Files System). Prvním prototypem byl IR (Information Retrieval) uvedený v roce 1958. V roce 1961 letectvo Spojených Států provozovalo SAC FFS, pravděpodobně první databáze obsahující sémantické informace. Jako první samostatný SŘBD byl v roce 1965 představen GIS (Generalized Information System). Ten se později stal součástí IMS (viz dále).

3.1.2 Hierarchický model

Společnost IBM vyvíjela databázový systém první generace v druhé polovině šedesátých let, kdy byl představen systém IMS (1968-9). Poprvé byl komerčně použit NASA pro projekt Apollo.

Data v tomto modelu jsou uspořádána do stromové struktury, každý prvek tedy může být odkazován (pomocí ukazatelů) nejvýše jedním jiným prvkem. Jde o formu vztahu předek → potomek, tato vazba je typu 1:N[2]. Model je založen na teorii grafů.

Tento model je vhodný zejména k realizaci „katalogových" systémů, kde je hierarchické strukturování dat dostačující. V těchto případech se vyznačuje dosud nepřekonanou rychlostí, nedostatkem je ale obtížná udržovatelnost a absence možnosti ad-hoc dotazování.

3.1.3 Síťový model

Síťový model byl formálně definován na „Conference on Data Systems Languages" (CODASYL) v roce 1971. Zdokonaluje hierarchický systém, který nebyl z výše uvedených důvodů vhodný pro obecné použití. Také síťový model je založen na teorii grafů, přípustné jsou i grafy, které nejsou stromy.

Data jsou seskupena do množin (model je založen na matematické teorii množin a teorii grafů), kde každá množina má jméno, vlastníka a členské prvky. Na začátku navigační cesty je jeden vlastník, který odkazuje na první členský prvek, ten odkazuje na další… Vlastník i členský prvek může být obsažen ve více množinách, čímž je možné implementovat vazbu M:N.

Podobně jako hierarchický model, i zde hlavní nedostatky vyplývají z výhradně navigačního přístupu a obtížné udržovatelnosti.

---

[2] Tento koncept je velmi podobný struktuře jazyka XML – shodný je navigační přístup. I současné XML

3.2 Relační databázové systémy

Formální základ položil v roce 1970 E. F. Codd. v [CODD70]. Využívá zde matematické teorie množin (set theory). V první části mj. definuje pojmy relace (relation), doména (domain), záznam (tuple) a pojmy primární a cizí klíč (primary, foreign key). Poprvé také zmiňuje nutnost normalizace, ta však zatím sestává jen ze zajištění první normální formy, tedy nezbytného předpokladu relačního modelu. V druhé části se zabývá matematickými operacemi nad relacemi, jmenovitě permutací (permutation), projekcí (projection), spojením (join) a restrikcí (restriction).

3.2.1 Datový model

Relační paradigma a potažmo i relační datový model používá velmi úzkou sadu konstruktů. Velké množství sémantických informací zůstává skryto mimo datový model, tj. bez znalostí modelované domény – pouze z fyzického datového modelu, není uživatel schopen odvodit její skutečný sémantický model.

Model relační databáze M. Gogolla rozděluje na syntaktickou (definiční, strukturní) a sémantickou (datovou) část ([GOGO02], [GOGO05])[3]. Definiční část tvoří objekty definující strukturu relačního modelu. Jde o:

- schéma relace (definice struktury relace),

---

databáze trpí stejnými nedostatky, jaké jsou uvedeny u hierarchického modelu.

[3] Syntax a sémantika se těchto pracích vztahuje k popisu (meta)modelů, sémantika dat, jak je chápána v kontextu této práce, je obsažena v obou složkách.

- atribut daného schématu a
- doménu (datový typ) daného atributu.

Datovou část pak tvoří:

- relace (tabulka),
- záznam dané relace a
- atomické hodnoty, data.

V objektové terminologii představuje definiční část třídy a datová instance. Pro potřeby transformace schématu je podstatná část definiční.

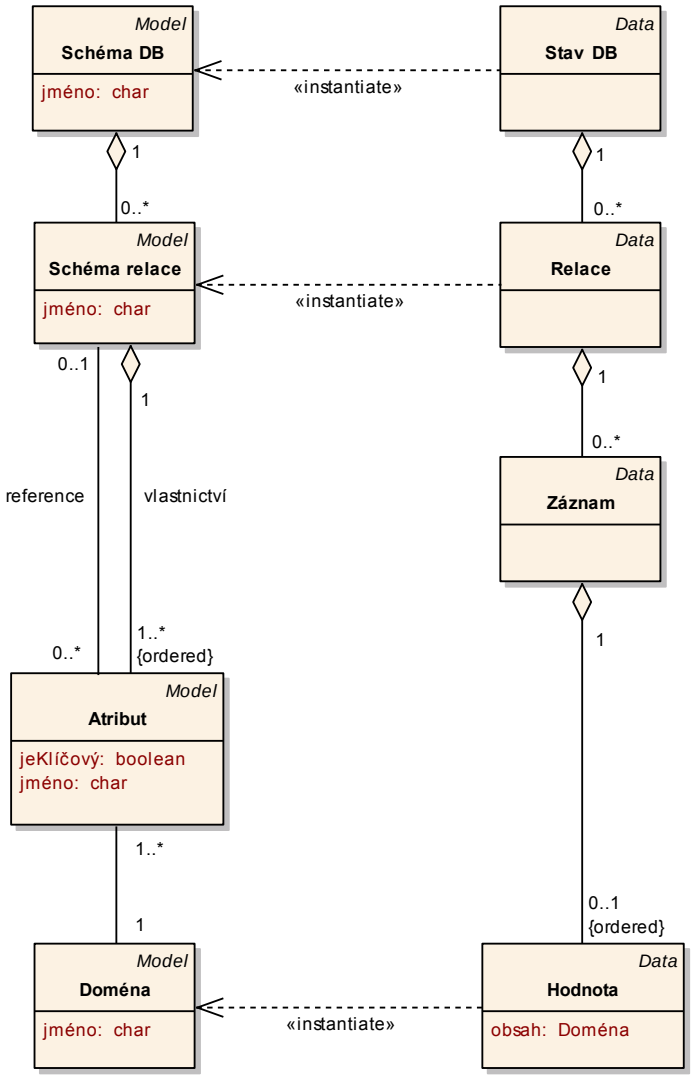


schéma 1: relační model podle [GOGO02], upraveno a doplněno

Kromě struktur uvedených ve schématu existují v databázi další, pro tuto abstrakci nepodstatné objekty. Ty souvisejí např. s fyzickým uložením dat (tabulkové prostory, datové soubory…) nebo optimalizací výkonu (indexy).

Schéma DB je množinou schémat všech relací v databázi.

Schéma relace definuje strukturu záznamů, které tvoří prvky dané relace. Definicí se rozumí stanovení uspořádané množiny dvojic atribut-doména.

Schémata relace jsou dále v textu odkazována velkými písmeny **R** a **S**.

Atribut (spolu s doménou) je předpis pro datovou položku (hodnotu) záznamu. Definuje, jak na tuto položku bude odkazováno (určuje její jméno). Má vlastnost `jeKlíčový`, která určuje jeho příslušnost do **primárního klíče** relace.

Atribut (nebo skupina atributů) je dále odkazován velkým písmenem **A**, příp. **B**. Množiny atributů relací r a s se označují stejně jako schémata těchto relací, tedy R a S.

Datový typ atributu definuje množinu hodnot, jakých mohou konkrétní hodnoty danému atributu příslušné nabývat. Množina přípustných hodnot atributu se také nazývá **doména**.

Relace je množina jí příslušejících záznamů. Všechny záznamy jedné relace musejí mít stejnou strukturu, tedy stejný počet atributů a jejich datový typ.

Relace jsou značeny malými písmeny **r** a **s**.

Záznam (tuple) je prvkem relace, jeho struktura musí respektovat předpis v podobě schématu relace. Záznam je tedy uspořádanou množinou hodnot.

Záznam je značen malým písmenem **t** nebo **u**.

Hodnota je atomická datová jednotka. Musí být prvkem domény atributu, tj. odpovídat předepsanému datovému typu. Pokud to specifikace atributu umožňuje, může nezávisle na doméně atributu nabývat i hodnoty **`NULL`**. Hodnota pak pro daný záznam není známa, nebo není definována.

Hodnota se zapisuje malým písmenem **a**, příp. **b**.

Pozn.: V souvislosti s normalizací (viz dále) se schémata relace rozkládají na další, dílčí schémata. V tomto smyslu je možné chápat schéma celé databáze jako schéma jedné relace, která je z různých (pro tuto diskusi nerelevantních) důvodů rozdělena do více schémat[4]. Aby nedošlo ke ztrátě informace o vztazích mezi schématy relace, je definován pojem **cizího klíče** (FK, foreign key). Ten je v modelu vyjádřen jako atribut s vazbou reference na odkazované schéma relace.

3.2.2 Relační algebra

Základ relační algebry tvoří operandy a operace s přípustnými operandy a definovaným výstupem. Operandy a výstupy operací jsou vždy relace. Relační algebra umožňuje pracovat s relacemi na abstraktní úrovni bez nutnosti znalosti konkrétního dotazovacího jazyka. Je využívána např. při optimalizaci postupu vyhodnocování dotazů nebo ověřování vyšších relačních jazyků ([CERI85]).

Operace relační algebry se dělí na operace **množinové** (sjednocení, průnik, rozdíl, kartézský součin) a **speciální relační** (selekce, projekce, spojení, dělení). **Základní operace** selekce, projekce, kartézský součin, sjednocení a rozdíl tvoří úplnou množinu potřebnou k manipulaci s daty a většinu z nich lze přímo vyjádřit jazykem SQL (viz dotazovací jazyk dále). **Doplňkové operace** spojení, průnik a dělení jsou definovány pomocí základních operací a

---

[4] Prototyp databáze navržené podle Coddových pravidel - System R - udržoval všechna data v jediné relaci.

slouží ke zjednodušení složitých algebraických výrazů. V praxi nejčastěji používané jsou operace selekce, projekce a spojení.

Přestože výsledkem všech operací musí být další množina, objevují se v praxi ve výsledkových sadách opakující se prvky. Pro jejich omezení je nutné použít SQL klausule DISTINCT. To platí pro operace sjednocení a projekce.

3.2.2.1 Kalkul

K získání libovolné informace uložené v databázi plně postačují výše uvedené operace a jejich kombinace, tj. tyto operace postačují k realizaci libovolného dotazu vyjádřeného **kalkulem** pro relační databáze. Kalkulem se rozumí matematický zápis operací výrokové logiky (operátory $\neg$, $\wedge$ a $\vee$ a kvalifikátory $\exists$ a $\forall$) používaný jako dotazovací nástroj. Podle přístupu k relacím a následné tvorbě dotazů pak rozlišujeme n-ticový a doménový kalkul (tuple, resp. domain calculus). N-ticový kalkul je základem jazyka SQL (viz dotazovací jazyk).

3.2.3 Dotazovací jazyk

De jure standardem pro dotazování nad relačními databázemi je jazyk SQL. Objevil se jako součást systému System R společnosti IBM koncem sedmdesátých let a od té doby byl implementován téměř ve všech relačních systémech. Poprvé byl standardizován společnostmi ANSI a ISO v roce 1986 a 1987, později byl standard rozšířen o integritní omezení (1989). Poslední přijatý a dosud platný standard je z roku 1992 (X3.135-1992 a ISO/IEC 9075:1992), označován jako SQL92. V praxi se objevují početné mutace, odvislé od konkrétní implementace DBMS, SQL92 je jejich společnou podmnožinou.

Příkazy jazyka SQL lze podle jejich charakteru rozdělit do tří skupin. Jazyk DDL (Data Definition Language) umožňuje definovat strukturu databáze, DML (Data Modification Language) pracuje s daty v této struktuře a jazyk DCL (Data Control Language) slouží k řízení procesů DBMS.

Část DDL slouží k vytvoření schématu databáze, takový kód pak při transformaci schématu slouží jako hlavní zdroj sémantických informací. DML umožňuje dotazovat data v databázi, pomocí jeho kódu jsou migrována (a analyzována) data. Kód DCL není pro tuto práci významný.

- DDL

Základním příkazem pro definici struktury DB je příkaz `CREATE TABLE` (případně `ALTER TABLE`). Tímto příkazem je vytvořeno schéma relace, zároveň jsou definovány atributy a jejich datové typy. U atributů je nastavena vlastnost klíčovosti.

- DML

Nejdůležitější operace relační algebry, tedy selekce, projekce a spojení jsou realizované příkazem

$$\texttt{SELECT } A_1\texttt{, } A_2\texttt{... FROM r, s... WHERE p.}$$

Ekvivalentní zápis pomocí operací relační algebry zní

$$\pi_{A1,\ A2...}\ (\delta_{(p)}(r \times s \times ...)),$$

v n-ticovém kalkulu pak

$$[A1,\ A2...]\{r \oplus s|p\}.$$

3.3 Objektové databázové systémy

Relační paradigma se týká pouze uložení dat v databázi, objektové naopak souvisí s programovacími jazyky (Smalltalk, Java, C++…). Protože není implicitně jasné, jak zasahuje objektové paradigma do databázové teorie, je nejprve třeba objektový databázový systém definovat.

V roce 1989 byla v [ATKI89] poprvé uveřejněna kritéria, které musejí nebo mohou objektové DS splňovat. Povinné požadavky lze rozdělit na databázové a objektové:

Požadavky na DBMS:

- **Trvanlivost** (persistence) – data musejí být dlouhodobě uložená (k dispozici více procesům), a to implicitně (bez zvláštně uživatelsky vyvolávaných mechanismů).
- **Správa druhotných úložišť** (secondary storage management) – data musejí být transparentně spravována tak, aby DBMS umožňoval efektivní správu velkého množství dat (tj. musejí být k dispozici mechanismy jako např. indexy, clustering, buffering…).
- **Souběžnost** (concurrency) – musí být umožněn současný přístup více procesů (uživatelů), je zde předpokládána, nikoli však vyžadována, správa transakcí.
- **Obnovitelnost** (recovery) – po výpadku systému musejí být data obnovitelná do konzistentního stavu.
- **Ad-hoc dotazování** (ad-hoc query facilities) – data musejí být efektivně přístupná ad-hoc (tj. nikoli sekvenčně).

Požadavky vyplývající z objektově orientovaného přístupu:

- **Komplexní objekty** (complex objects) – komplexní objekty jsou složeny z jednodušších; požadavek explicitně jmenuje primitivní datové typy (literály), další skládání komplexních objektů zmíněno není.

- **Identita objektů** (object identity) – jsou podporovány 2 koncepty ekvivalence objektů: 2 objekty jsou identické (jde o tentýž objekt), nebo ekvivalentní (různé objekty mají shodný stav).

- **Zapouzdření** (encapsulation) – data a implementace metod objektů jsou uživateli skryty, což přináší výhody v podobě modularity (jsou možné lokální nezávislé změny v implementaci při zachování rozhraní); zapouzdření dat není striktně vyžadováno.

- **Podpora typů nebo tříd** (types or classes) – oba koncepty adresují sdružení společných charakteristik objektů do abstraktních datových typů, třída navíc umožňuje objekty vytvářet (object factory) a sdružovat (object collection).

- **Dědění** (inheritance) – typy nebo třídy je možné uspořádávat do hierarchií tak, že podřazený prvek přebírá strukturu a chování nadřazeného.

- **Přetížení, potlačení a pozdní vazba** (overloading, overriding and late binding) – přetížení umožňuje různě definovat stejnojmenné metody (souvisí s polymorfismem), potlačení děděné metody dovoluje její redefinici v dědícím objektu a pozdní vazba, vynucená těmito koncepty, znamená kompilaci metod v reálném čase.

- **Rozšiřitelnost** (extensibility) – typy, které jsou v systému předdefinované, je možné uživatelsky rozšířit; tyto nové typy se používají stejným způsobem, jako předdefinované

- **Výpočetní úplnost** (computational completeness) – dotazovací jazyk musí umožňovat implementaci libovolného algoritmu, tj. výpočet libovolné funkce realizovatelné v jiném programovacím jazyce

Kromě povinných vlastností jsou v [ATKI89] jmenovány i dobrovolné (a doporučované) znaky objektového DBMS. Jsou to **mnohonásobná dědičnost** (multiple inheritance), **kontrola a dedukce typů** (type checking and inferencing), **distribuovanost** (distribution), **definované transakce** (design transactions) a **verzování** (versions). Tyto vlastnosti nejsou vyžadovány, a proto zde nejsou podrobně popisovány.

Výše uvedené principy byly dále rozvíjeny v [STON90]. Dokument však explicitně neříká, že specifikuje požadavky na objektové databáze.

3.3.1 Datový model

Objektový datový model podle [ODMG00] rozeznává 2 druhy základních konstruktů – objekty a primitivní datové typy (literály). Přes drobné neshody v terminologii je možné objektový model podle ODMG považovat za nadmnožinu modelu OMG.

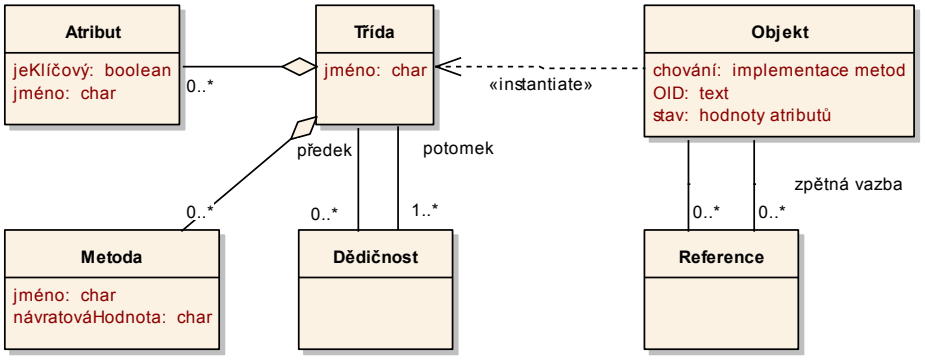


schéma 2: objektový model podle [ODMG00] a [ATKI89]

Třída je objekt, který předepisuje strukturu (**atributy**) a chování (**metody**) svým instancím (**objektům**). Je vždy příslušná nějakému typu (viz dále).

Dědičnost realizuje vztah generalizace a určuje strukturu a chování objektů děděných tříd. Mnohonásobná dědičnost je povolena pouze pro atributy, chování je možné dědit pouze od jednoho předka.

Objekt ve smyslu instance třídy má svůj unikátní identifikátor OID, který je nezávislý na stavu daného objektu. Objekty mají stav (hodnoty **atributů** a **vztahy**) a chování (**metody**).

3.3.2 Objektová algebra

Díky absenci standardu se objektové algebry používané v různých komerčních DBMS liší, podobně jako algebry vznikající mimo produkční společnosti. Různé návrhy objektové algebry lze nalézt např. v [SHAW89], [RUND92], [KORT99] nebo [ALHA99].

Operace společné pro většinu navržených objektových algeber odpovídají základním operacím relační algebry:

- **Selekce** je definovaná analogicky s ekvivalentní operací relační algebry (relace je nahrazena kolekcí instancí)
- **Projekce** je definovaná analogicky s ekvivalentní operací relační algebry (atributy jsou díky zapouzdření nahrazeny metodami k nim přistupujícím)
- **Kartézský součin** je definovaný analogicky s ekvivalentní operací relační algebry (výsledkem je sloučení hodnot atributů a/nebo identit objektů, v závislosti na tom, zda je jeden nebo oba slučované objekty literálem)
- **Sjednocení** je definované analogicky s ekvivalentní operací relační algebry (výsledkem jsou návratové hodnoty metod průniku obou rozhraní)
- **Spojení** je definované jako kombinace kartézského součinu a selekce

3.3.3 Dotazovací jazyk

Standardem pro dotazování nad objekty je jazyk **OQL** společnosti ODMG. Vychází z relačního dotazovacího jazyka SQL 92 a objektového modelu ODMG. Jeho součástí je jazyk ODL (Object Definition Language) sloužící k definici schématu, který je založen na IDL (Interface Definition Language) společnosti OMG a jazyk OML (Object Manipulation Language).

Jazyk OQL je však v praxi velmi málo používán. Vznik objektových databází je mj. reakcí na problémy impedančního nesouladu, jehož opakem a ideálním stavem je „bezešvá“ integrace aplikace s daty. OQL má s objektovými programovacími jazyky velmi málo společného (je procedurální – vychází z SQL a sdílí veškeré jeho nevýhody, mj. část OML není výpočetně kompletní) a tak byl vytlačen nativní podporou nejpoužívanějších objektových

jazyků. Proto mají objektové databáze jako např. Gemstone, O2, Versant, Jasmine, ObjectStore a další zabudováno rozhraní pro jazyky Smalltalk, Java nebo C++.

4 Modelování

Obecný postup databázového návrhu ([RUMB91], [TEOR99]) znamená vytvoření konceptuálního modelu a jeho (postupnou) transformaci do implementačního. Druhý jmenovaný model je pak snadno převoditelný do programového kódu (SQL DDL, Java, SmalľTalk…). Během vývoje tedy postupně dochází k přechodu od abstraktní ke „strojové" úrovni. Aby byl tento proces robustní a přenositelný, je třeba striktně oddělovat jednotlivé vrstvy abstrakce a jasně mezi nimi definovat vztahy a transformační pravidla. Takový přístup má podle [OMG_03] dlouhodobé výhody ve smyslu pružnosti následujících etap vývoje IS:

Implementace: Změny v podnikové infrastruktuře mohou být zohledněny rychle a levně.

Integrace: Je možné automatizovat tvorbu „datově-integračních mostů" (pro připojení k novým systémům) a přechod na nové platformy.

Údržba: Strojově zpracovatelné modely umožňují vývojářům snadnou vertikální navigaci mezi modely a přístup ke specifikaci systému.

Testování a simulace provozu: Modely mohou být verifikovány proti specifikaci systému, modely na nejnižších úrovních abstrakce mohou být verifikovány proti specifikacím platformy, a stejně tak funkčnost systému může být validována ještě před jeho nasazením.

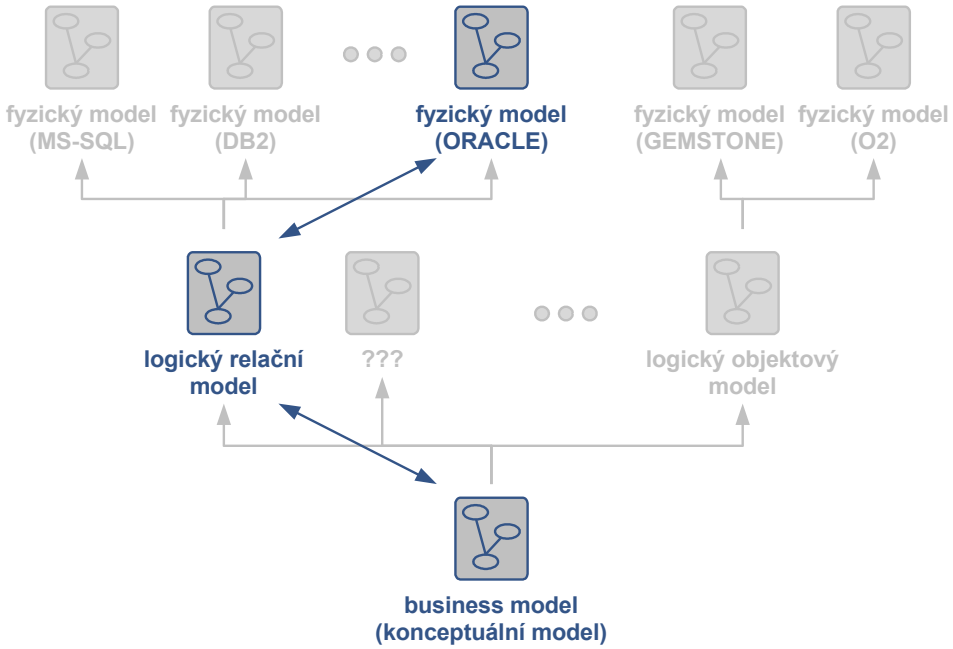


schéma 3: přechod mezi úrovněmi modelů

Pro přechod mezi databázovými modely (konceptuální ↔ logický ↔ fyzický) tento přístup znamená snadné zohlednění nově zjištěných faktů (při nesprávné nebo z jiného důvodu měněné interpretaci datových struktur, viz datové reversní inženýrství) nebo oprav v konceptuálním modelu, které korigují chyby v relační implementaci.

4.1 ANSI/SPARC

Komise ANSI/SPARC doporučila v [TSIC78] již v roce 1978 rozlišovat 3 úrovně abstrakce modelů - (1) **interní** úroveň odrážející použitou technologii, (2) **konceptuální** úroveň na ní nezávislou a (3) **externí** vrstvu reprezentující pohledy uživatele. Jde o první jednoduché rozvrstvení přímo použitelné pro databázové systémy.

Pokud by bylo členění aplikováno na současné relační databázové systémy, byly by výše uvedené relační konstrukty prvky interní vrstvy a z pohledu konceptuální úrovně by tvořily ER model. Externí vrstva by sdružovala sestavy vytvářené aplikací.

4.2 ODP

ODP (Open Distributed Processing) je sada standardů ISO/IEC 10746 (viz [ODP_96]), které se primárně týkají distribuovaného zpracování, obsahují však principy v modelování přijaté a dále rozvíjené. Navržený referenční model umožňuje popsat systém žádoucím způsobem, tj. pojímá artefakty od počátečních fází vývoje (zadání), po jeho ukončení (implementaci). Základními koncepty jsou zde transparentnost, funkce a pohled.

Transparentnost odstiňuje uživatele od technických podrobností vyplývajících z distribuovanosti. Pro vývoj a údržbu je důležité, že systém se jeví tak, jako by distribuovaný nebyl. Transparentnost souvisí s pojmem abstrakce v MDA (viz dále).

Funkce ODP zajišťují výše uvedenou transparentnost.

Pohledy definují vývojový framework ODP a určují, jaké funkce jsou v kterém případě použity, tj. jaká je úroveň transparentnosti. V [ODP_96] je definováno 5 pohledů:

Podnikový pohled (enterprise viewpoint) – pohled na systém a jeho prostředí, který se zaměřuje na účel, rozsah a pravidla použití

Informační pohled (information vp) – pohled na systém a jeho prostředí, který se zaměřuje na tok informací a procesy

Výpočetní pohled (computational vp) – pohled na systém a jeho prostředí, který umožňuje rozdělení (distribuci) pomocí funkční dekompozice systému na objekty, které navzájem reagují přes rozhraní

Projekční pohled (engineering vp) - pohled na systém a jeho prostředí, který se zaměřuje zejména na mechanismy a funkce potřebné k podpoře interakcí mezi objekty systému

Technologický pohled (technology vp) - pohled na systém a jeho prostředí, který se zaměřuje na technologie systému

4.3 MDA (Model Driven Architecture)

„Posláním OMG (v kontextu MDA) je pomoci… řešit integrační problémy poskytnutím otevřené a nezávislé specifikace přenositelnosti.“ [OMG_01].

K dosažení tohoto mj. definuje různé úrovně modelů podle polohy na ose specifikace zadání ↔ implementace (viz architektura). Upřesňuje také, jakým způsobem jsou realizovány přechody mezi těmito modely (viz transformace modelů). MDA tak navazuje na dříve uvedené práce.

MDA využívá v modelování několika základních konceptů, z nichž některé jsou pro tuto práci zejména důležité: pojem modelu v různých úrovních abstrakce a pojem platformy.

Model je reprezentací části funkčnosti, struktury, nebo chování systému. Musí mít jasně definovanou (formalizovanou) formu (syntax) a význam (sémantiku). Předpokládají se zde spíše grafické vyjadřovací prostředky (UML), za model lze však podle této definice považovat i programový kód.

Abstrakce je vypuštění detailů nerelevantních pro daný účel. Typicky je v různé míře abstrahováno od detailů implementace.

Platforma je SW infrastruktura implementovaná konkrétním způsobem na konkrétní HW infrastruktuře

4.3.1 Architektura

MDA rozeznává 3 klíčové úrovně abstrakce modelované reality:

4.3.1.1 CIM

Doménový business model (Computation Independent Model, CIM) zachycuje situaci v reálném, popisovaném světě. Abstrahuje od všech omezení a podmínek vyplývajících z počítačového zpracování. Je sestavován doménovými odborníky nebo business analytiky a je vyjádřen terminologií příslušné problémové doméně.

Tento model a s ním související transformace není ve specifikaci příliš rozebírán, souvisí to pravděpodobně s faktem, že business modelování nemá v UML, což je „nativní" modelovací jazyk MDA, odpovídající podporu. V obecné rovině je ale podle [OMG_03] podporován (*„…přestože nemusí být doménový model součástí konkrétního schématu, je v MDA konzistentně zahrnut do celkové architektury"*)

Podle specifikace [ODP_96] odpovídá podnikovému a informačnímu pohledu.

4.3.1.2 PIM

Model nezávislý na platformě (Platform Independent Model, PIM) zohledňuje zvolenou architekturu, ne však cílovou platformu. Je vytvářen

analytiky, obsahuje např. algoritmizaci složitých procesů nebo logické datové struktury.

Podle specifikace [ODP_96] odpovídá projekčnímu a výpočetnímu pohledu.

4.3.1.3 PSM

Model závislý na platformě (Platform Specific Model, PSM) zohledňuje zvolenou architekturu i platformu. Je vytvářen návrháři a reprezentuje jednoznačný předpis pro kódování.

Podle specifikace [ODP_96] odpovídá projekčnímu a technologickému pohledu.

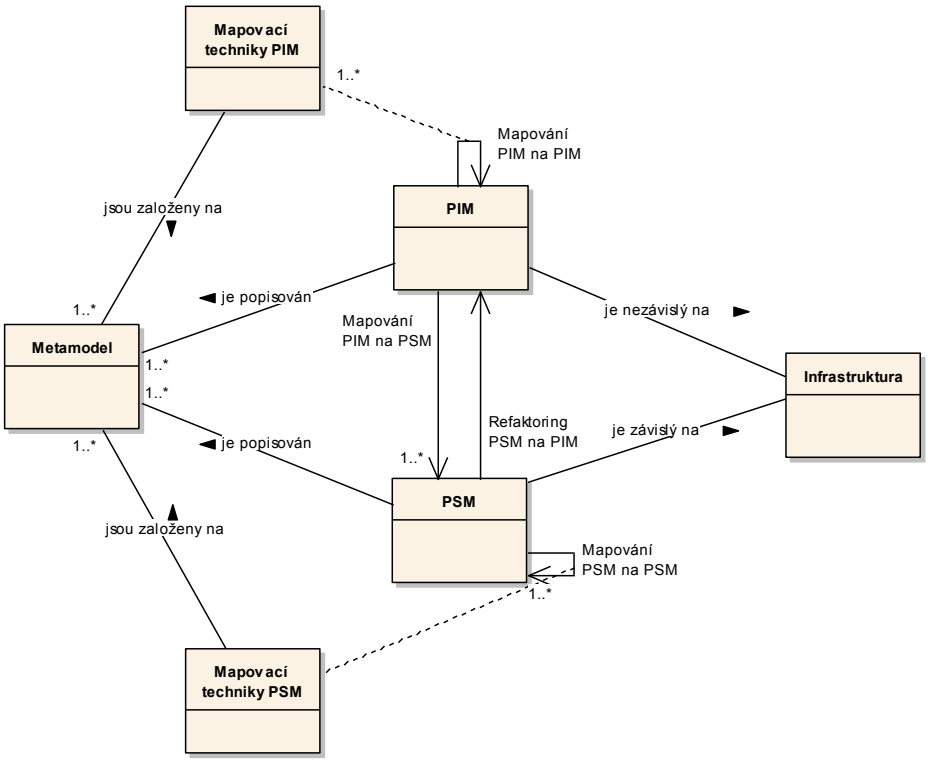


schéma 4: model architektury MDA [OMG_03], upraveno

Schéma představuje metamodel architektury MDA. Ačkoli je CIM (tj. business model) ve specifikaci explicitně uveden, není zde z výše uvedených důvodů zachycen.

Vzhledem k tomu, že je MDA obecně použitelné v systémovém inženýrství a není vázáno na databázové technologie, objevily se v literatuře návrhy úprav (viz [MORA02],[GOGO02],[DOMI03]). Domínguez položil ekvivalenci mezi business model (CIM) a konceptuální model, PIM a logický datový model a PSM a fyzický datový model. Přes některé rozpory ve specifikacích nejdříve uvedených modelů byl tento návrh přijat.

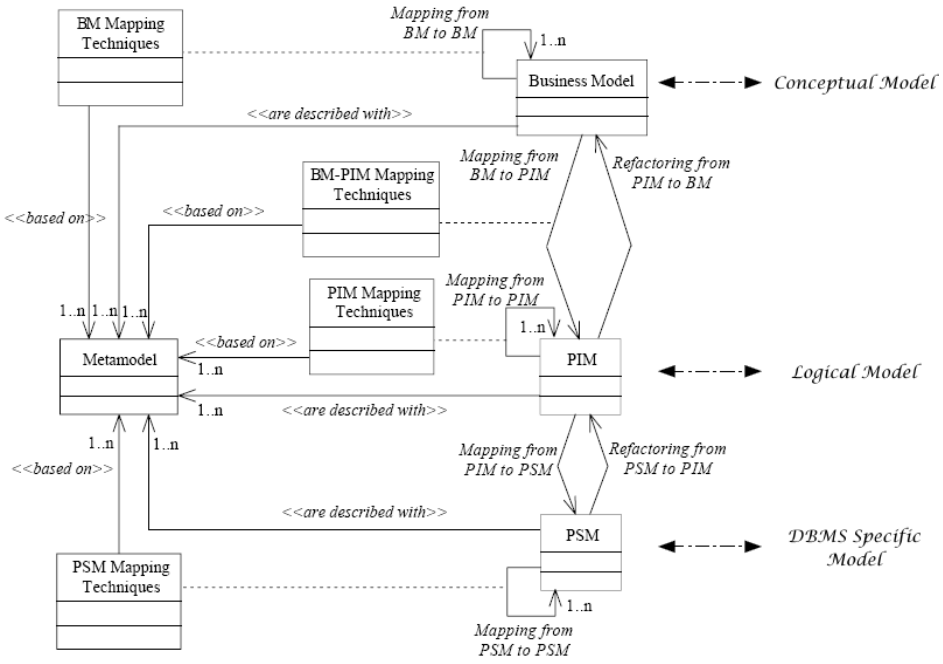


schéma 5: upravený model MDA [DOMI03]

4.3.2 Transformace modelů

Uvedeného cíle dosahuje MDA kromě modelů na vyšší úrovni abstrakce také minimalizací nákladů na změny požadavků, platformy nebo vývojového prostředí ([ATKI02]). To je řešeno (v maximální možné míře) automatizovanou transformací modelů. Každá transformace na konkrétní model na nižší úrovni abstrakce vyžaduje doplnění specifických značek (marks). Ty zajišťují vložení informací, které v modelu na vyšší úrovni abstrakce chybí. Tímto způsobem lze provádět i zpětnou transformaci relačního databázového schématu do konceptuálního: na příslušná místa, která by jinak znamenala mnohoznačnost, jsou vloženy značky reprezentující dodatečně získané sémantické informace.

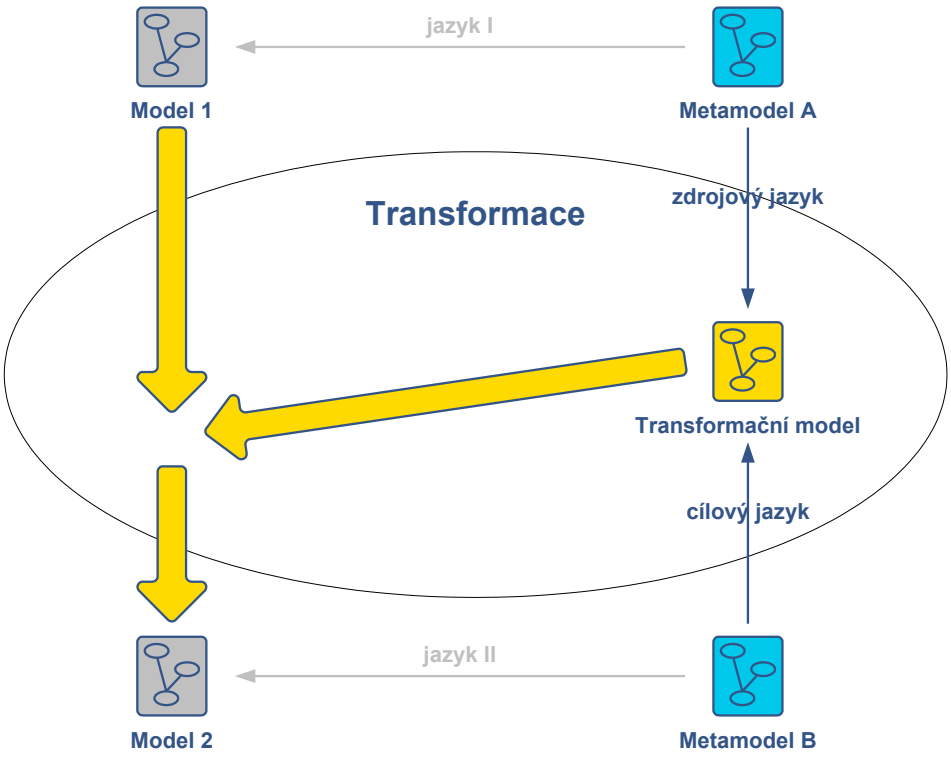


schéma 6: transformace pomocí metamodelů [OMG_03], upraveno

Diagram ukazuje obecný princip transformace modelů pomocí metamodelů. V případě klasického návrhu relačního databázového schématu jsou prvky modelu nahrazeny takto:

- Metamodel A: ER metamodel (prvky Entita, Atribut...)
- Metamodel B: relační metamodel (prvky Tabulka, Atribut...)
- Model 1: ER model (prvky Osoba, Jméno...)
- Model 2: relační model (prvky prirazeniosoby, osoba, jmeno...)
- Jazyk I (zdrojový jazyk) je smyšlený jazyk pro deklaraci ER schématu, jazyk II (cílový jazyk) pak SQL DDL.

4.3.2.1 UML

Pro definici modelů a metamodelů je v MDA používán jazyk UML (Unified Modeling Language) [OMG_01]. Specifikace tohoto jazyka je popsána v [OMG_05]. Specifikace definuje balíčky (packages), které obsahují elementy UML potřebné k vyjádření konstrukcí, pro jejichž definici je balíček vytvořen.

Balíčky obsahují UML elementy a vztahy mezi nimi. V balíčku obsažené elementy jsou s existujícími elementy (tj. elementy, které jsou definovány v jiných, již sloučených balíčcích) slučovány pomocí vazeb `PackageImport` a `PackageMerge`.

`PackageImport` pouze umožňuje používání elementů v původním balíčku bez použití kvalifikovaného jména elementu, tj. rozšiřuje jmenný prostor původního balíčku o balíček importovaný. Tento postup je vhodný pro sady balíčků, jejichž množiny elementů jsou disjunktní, případně shodné elementy

rozšiřujícího balíčku jsou používány jiným způsobem než ekvivalentní elementy původního, importujícího balíčku.

`PackageMerge` je vhodný pro kombinaci balíčků, jejichž některé elementy jsou shodné (ve smyslu jejich shodného jména). Element v rozšiřujícím balíčku je sloučen se shodným elementem v balíčku původním tak, že původní element je rozšířen o vlastnosti elementu rozšiřujícího. Takto jsou „jednoduchým" elementům v základních balíčcích, přidávány vlastnosti využitelné pro „složitější" modelování, tj. tyto elementy jsou rozšiřovány o podporu konstrukcí, k jejichž vyjádření je rozšiřující balíček určen.

Relevantní ukázky jazyka UML jsou ve vizuální podobě uvedeny v kapitole *Přílohy*.

4.3.2.2 OCL

Součástí modelovacího jazyka UML se dodatečně stal popisovací a deklarativní jazyk OCL (Object Constraint Language) [OMG_05-2]. Tento jazyk slouží primárně k vyjádření složitějších podmínek, pro které je jazyk UML nedostatečný. OCL umožňuje pracovat s elementy modelu UML, tj. dotazovat je, porovnávat jejich hodnoty a v neposlední řadě také definovat metody, kterými objekty reagují na příchozí operace.

K tomuto jsou v OCL definovány prvkové a množinové operace, které pracují s objekty, hodnotami jejich atributů a s kolekcemi objektů. Pomocí kombinací těchto operací je možné vyjádřit libovolnou metodu objektu. Pro její vyjádření však platí omezení uvedené výše, a to že OCL je mj. jazyk deklarativní. Neumožňuje tedy přímou definici „invazivních" metod, tedy metod, které nejsou typu `isQuery`. Cestou k dosažení tohoto cíle je použití OCL výrazů jako vstupních a výstupních podmínek popisované metody.

I díky množinovým operacím je v OCL relativně snadné popisovat algoritmy. Ukázky a použití jazyka OCL je opět uvedeno v kapitole *Přílohy*.

5 Principy databázového návrhu

Terminologie související s databázovým návrhem se v literatuře často i podstatně liší (viz [RUMB91], [TEOR99] a další). V této práci je použito názvosloví převzaté z několika zdrojů a dodatečně upravené. Postupy byly sladěny s principy MDA.

Obecný postup databázového návrhu se skládá z těchto kroků:

- vytvoření konceptuálního modelu
- odvození logického datového modelu z konceptuálního
- odvození fyzického datového modelu z logického

Každý z těchto kroků má pro dané paradigma svá specifická pravidla. Ta jsou pro relační databáze jasně definována, v praxi ověřena a úspěšně používána. Pro mladší objektové bohužel nejsou zatím přesně popsaná.

Kromě uvedených 3 hlavních kroků jsou v rámci daných úrovní modelů prováděny vnitřní transformace, optimalizující některou ze sledovaných veličin. Takové úpravy mají podstatný vliv na zpětnou transformaci modelů a jsou podrobněji popsány dále.

5.1 Konceptuální model (CIM)

Vzhledem k tomu, že neexistuje všeobecně přijatý standard pro konceptuální modelování, je vhodné stanovit si minimální potřebnou množinu komponent modelu. Základní elementy jsou totiž v různých verzích konceptuálních modelů víceméně shodné, liší se spíše počet „doplňkových“ prvků, potažmo pak množství informací v modelu obsažených.

P. Chen předkládá v [CHEN76] „Entity-Relationship Model" (ER model), kde definuje prvky Entita, Role, Atribut a Vztah. Tento výčet překračuje možnosti relačních databází, objektová technologie však využívá dalších konstruktů. Naopak např. ER model podle [DOWE95] je pro transformaci zbytečně „pestrý", přesahuje možnosti relačního i objektového přístupu (v kontextu databází). Proto, pro práci s relačním a objektovým implementačním datovým modelem, je účelné přijmout jednoduchý, ale vyčerpávající aparát. Tím je rozšířený ER model (extended ER, EER model), který oproti [CHEN76] obsahuje navíc hierarchické vazby[5] a vypouští role.

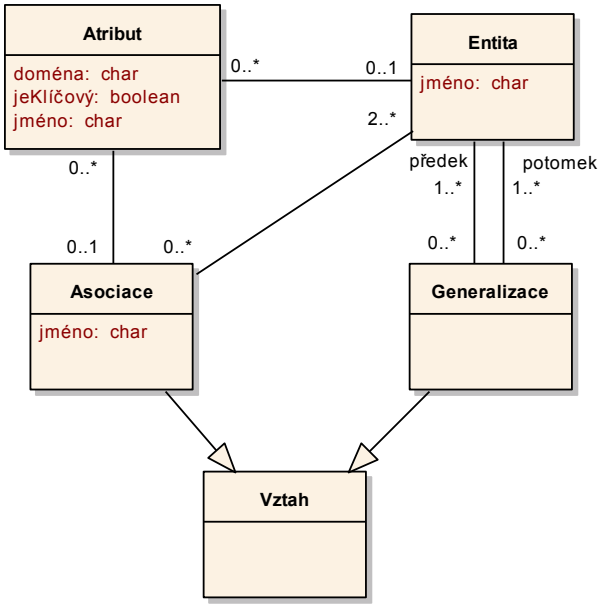


schéma 7: EER model

[5] Chen v [CHEN76] připomíná, že – vyjádřeno soudobou terminologií – entita (ve smyslu instance) může náležet více sadám entit (ve smyslu třídy). Jako příklad uvádí sady entit, kdy jedna je podmnožinou druhé (osoba, muž). Toto je v EER modelu vyjádřeno vztahem „generalizace".

Schéma pro jednoduchost zobrazuje pouze definiční – „třídní“ objekty. V případě zachycení instančních objektů by tyto převzaly názvy třídních objektů a původně třídní objekty by byly nazvány „Sada entit“, „Sada atributů“, atp.

Vzhledem k tomu, že se jedná o model nezávislý na databázové technologii, má některé, pro (relační) databázové modelování netypické, vlastnosti:

- jsou povoleny **asociace typu M:N** – tyto asociace, stejně jako asociace dalších typů, mohou mít své vlastní atributy
- **atributy nemusejí být atomické** – doménu[6] atributu mohou tvořit heterogenní sady hodnot

Postupy vedoucí k sestavení konceptuálního modelu přesahují vymezený tematický rámec a nejsou zde proto diskutovány.

5.2 Mapování konceptuálního modelu na logický (CIM → PIM)

Modely na úrovni PIM odpovídají modelům uvedeným v popisu databázových paradigmat.

5.2.1 Relační DS

Model je definován v SQL92 jakožto společné podmnožině všech mutací SQL.

---

[6] Datové typy atributů jsou zde na logické úrovni (tzv. domény) – mohou být definovány např. jako „cena“, „hmotnost“, atd. To mj. umožňuje důslednější sémantickou kontrolu.

Mapování konceptuálního modelu zde může být plně automatizováno, pro překlad konstruktů konceptuálního modelu existují jednoduchá pravidla, která shrnuje následující tabulka:

Prvek modelu CIM		Prvek modelu PIM
Entita	→	Tabulka
Atribut	→	Sloupec tabulky
Asociace	→	realizovatelné několika způsoby pomocí cizích klíčů
Generalizace	→	

tabulka 1: mapování PIM → PSM v relačních DS

Asociace typu M:N jsou realizovány rozložením na 2 vazby 1:M (vzniká vazební tabulka).

5.2.2 Objektové DS

Model je vyjádřený v jazyce OQL ODL, který je zobecněním implementačních jazyků všech objektových databází.

Prvek modelu CIM		Prvek modelu PIM
Entita	→	Třída[7]
Atribut	→	Atribut, metoda
Asociace	→	Reference
Generalizace	→	Dědění (inheritance)

tabulka 2: mapování PIM → PSM v objektových DS

Asociace typu M:N je v objektových databázích přímo (její implementace DBMS pomocí např. oboustranného vazebního slovníku je transparentní).

5.3 Úpravy logického modelu (PIM → PIM)

5.3.1 Optimalizace zabezpečení dat (normalizace)

Účelem datové normalizace je převod datového modelu do takového tvaru, který v co možná největší míře zajišťuje konzistenci uchovávaných dat. Tu naopak ohrožují anomálie, které mohou vznikat při manipulaci s daty – tedy při jejich vkládání, aktualizaci a mazání. Tyto anomálie jsou pak důsledkem zejména redundance dat, kdy jedna informace je uložena na několika místech.

Ještě před několik lety se o normalizaci hovořilo pouze v souvislosti s relačním datovým modelem, s rozvojem objektových databází je však potřeba podobného objektového aparátu stále naléhavější. Používání návrhových vzorů je mnohdy dostatečné, nicméně teoreticky propracovaný a univerzální postup zde chybí. V literatuře se objevují návrhy odvozující pravidla objektové normalizace z relačních normálních forem.

První 3 relační normální formy (včetně BCNF), sestavil tvůrce relačního modelu E. F. Codd (viz [CODD70, 72, 74]). Pomocí funkčních závislostí (či spíše Armstrongových pravidel) definoval takovou úroveň konzistence dat, která je v dnešní praxi většinou pokládána za dostatečnou.

Většina prací zabývajících se objektovou normalizací (viz [AMBL03], [NUTS02]) odvozuje **3 objektové normální formy** (ONF) od relačních (RNF). ONF zamezují stejným problémům jako RNF, ale jejich definice se v porovnání mnohdy zdají být intuitivní – nemohou se opřít o propracovaný matematický aparát a pracují s příliš vágními pojmy.

R. Fagin odvodil v [FAGI77] a [FAGI79], že ani relační schéma v BCNF, tedy původně nejvíce restriktivní normální formě, nevylučuje v případě existence složených primárních klíčů vznik nekonzistencí. Definoval tedy **další dvě relační normální formy**, 4NF a 5NF (PJ/NF), které vycházejí z dalších, složitějších závislostí (multivalued a projection-join závislosti).

---

[7] Pokud bychom se na entitu dívali jako na množinu objektů se stejnými atributy, odvodíme pro objektový model, že entita spíše než třídě odpovídá množině vzájemně polymorfních objektů.

Tyto normální formy zatím objektově popsány nebyly – týkají se totiž nežádoucích závislostí mezi složkami primárního klíče (PK) a jejich přesná a přesto srozumitelná definice, snadno aplikovatelná v kontextu bez PK, chybí.

Definice RNF, tedy exaktní stanovení přípustných vztahů a způsob jejich dosažení, předpokládají znalost funkčních závislostí atributů. Tento druh informace však součástí konceptuálního CIM modelu není a musí být doplněn na úrovni PIM. Je nutná věcná znalost modelovaného systému k:

- sestavení přehledu funkčních závislostí, použitelných pro relační normalizaci, nebo
- provedení neformalizované normalizace tak jako u objektové normalizace.

Relační normální formy vděčí za svůj podpůrný aparát konceptu primárního klíče. Ten je definován v 1NF, jejíž dodržení je zásadním předpokladem pro pouhé vyhodnocení platnosti vyšších RNF. Pravidla jimi vymezená lze někdy až s jistou mírou zjednodušení interpretovat i „srozumitelně“, tj. bez odkazů na funkční závislosti (viz dále). Tak je sice možné normalizovat již konceptuální model, ale bez podpory matematického aparátu.

Dále v textu jsou popsány nejpoužívanější normální formy – v praxi se normální formy vyšší než 3NF téměř nepoužívají. Důvodem jsou (1) negativní dopady na výkonnost a (2) charakter běžně uchovávaných dat, který při splnění 3NF zajišťuje uspokojení i všech vyšších.

5.3.1.1 Relační normální formy

1NF je základním předpokladem relačních databází. Říká, že hodnoty atributů musejí být atomické, v rámci jednoho atributu se tedy nepřipouštějí žádné další struktury. Toto pravidlo není v některých relačních databázích

respektováno, čímž jsou narušeny základy relační algebry. Není pak např. možná přímá dostupnost libovolné informace pouze pomocí základních operací projekce, selekce a spojení.

Některé zdroje spojují 1NF s definicí primárního klíče. Na něm mají být závislé všechny neklíčové atributy. E. F. Codd však primární klíč definoval nezávisle na normálních formách.

Neformálně lze říci, že v tabulce nesmějí být multizávislosti, tj. primární klíč musí jednoznačně určovat každou hodnotu, nikoli neurčitě vymezovat jejich množinu.[8]

Definice druhé a dalších RNF již využívají funkčních závislostí mezi atributy nebo jejich skupinami. Formální zápis **2NF** vyjmenovává případy, kdy je přípustná závislost atributu A na atributu X ($X \rightarrow A$, X a A mohou být skupiny atributů):

- A je podmnožina X (triviální závislost) nebo
- X je nadmnožina primárního klíče (tj. ovlivňuje všechny atributy) nebo
- A je podmnožina primárního klíče nebo
- X i A jsou neklíčové prvky

[8] Pokud by tabulka obsahovala násobná data, nebylo by možné přistupovat k nim jednotlivě. Pokud by data byla rozepsána do více řádků, musel by dříve násobný atribut být součástí prim. klíče. Potom lze již přistupovat ke všem atributům a jejich dalším atributům (předpokládá se porušení minimálně 3NF). Lze se domnívat, že účel 1RNF je umožnění přímého přístupu ke každé informaci.

Alternativní vyjádření téhož zní: „Všechny neklíčové atributy musejí být závislé na celém primárním klíči“.

Počet přípustných případů závislosti atributů uvedených u 2NF je v **3NF** dále redukován. Pro X → A platí:

- A je podmnožina X (triviální závislost) nebo
- X je (nad)množina klíče (tj. ovlivňuje všechny atributy)
- (nebo A je podmnožina klíče) – neplatí u BCNF[9]

Jinými slovy: „Každý determinant (tedy atribut, na kterém je jiný atribut závislý) musí být klíč a zároveň 2NF musí být splněna“.

Vyšší relační normální formy se v praxi obvykle nepoužívají a přesahují rámec potřeb této práce.

5.3.1.2 Objektová normalizace

Objektová definice **1NF** uvedená v [AMBL03] analogicky k RNF říká: „Pokud existuje chování[10] objektu vlastní skupině jeho atributů, musí být toto chování zapouzdřeno v jejich vlastní třídě“.

Účelem **2NF** je v relačním paradigmatu zamezit existenci relací, které sdružují již „odhalené“ složené objekty, tj. skryté asociace, generalizace nebo „ploché“ agregace, kde jsou známé primární klíče obou rozdělovaných entit. Sledování

[9] BCNF (Boyce – Coddova normální forma) je původním vyjádřením 3NF, která je v současné podobě méně restriktivní

stejného cíle je patrné i v jejím objektovém znění [AMBL03], ačkoli je tato snaha o paralelu možná neopodstatněná: *„Třída nesmí obsahovat chování sdílené s jinou třídou. V opačném případě je nutné toto chování z dotčených tříd vytknout do samostatné třídy“.*

Definice zde poprvé odkazuje na více než 1 třídu. Porovnání více tříd má usnadnit odhalení třídy, která by v relačním paradigmatu byla určena „autonomní“ částí primárního klíče, tedy podklíčem, který je sám o sobě determinantem.

Podobně jako v předchozím případě, i u **3NF** je možno pozorovat motivaci k odhalení stejných zdrojů možných nekonzistencí jako v relačních databázích. Znalost modelované domény je však již nutností: *„Třídy nesmějí obsahovat chování, která mají samostatný význam, oddělený od třídy jako celku“.*

5.3.2 Výkonová optimalizace

Vzhledem k dodatečné strukturalizaci dat plynoucí z normalizace a tudíž častěji prováděným spojením tabulek, resp. složitější navigaci, je degradována výkonnost systému při manipulaci s daty. Proto je v praxi běžné, že není splněna ani 2NF.

Principem výkonové optimalizace je tedy denormalizace. Není však pravidlem, že schéma denormalizované do 1NF (tj. v extrémním případě pouze 1 relace) přináší maximální výkon. Denormalizaci je třeba provádět účelně, s přihlédnutím na potřeby výkonově kritických dotazů (viz např.

---

[10] Chováním Ambler rozumí metody i atributy třídy.

[PROP04]). Detaily denormalizace však nejsou pro tuto práci podstatné a nejsou proto dále rozebírány.

5.4 Mapování logického datového modelu na fyzický (PIM → PSM)

Modely na úrovni PSM odpovídají proprietárním řešením jednotlivých výrobců. Míra prováděných transformací se v obou paradigmatech řádově liší.

5.4.1 Relační DS

Vyžadované transformace jsou na této úrovni minimální. Rozdíly v modelech jednotlivých výrobců (Oracle, DB2, MS-SQL, Informix...) se týkají spíše fyzického uložení dat, které není pro tuto práci relevantní. Proto, a díky existenci standardu SQL92 (jazyk relačního PIM modelu) a jeho poměrně důslednému dodržování, nedochází na této úrovni v podstatě k žádným transformacím.

5.4.2 Objektové DS

Vzhledem k neexistenci uznávaného a používaného standardu pro definici objektových struktur a jeho faktickému nahrazení vyššími objektovými programovacími jazyky (C++, Smalltalk, Java...) je ve většině případů nutný překlad z jazyka OQL (PIM). Strukturální transformace se stejně jako v případě relačních databázových systémů provádí jen v minimální míře.

6 Datové reversní inženýrství

Reversní (systémové) inženýrství (RE) je opačným procesem k tvorbě informačních systémů. Artefakty, které tvoří vstupy „dopředného" inženýrství, tedy různé formy zadání, jsou v případě RE sledovaným cílem. Naopak implementace zadání je v zásadě jediným vstupem, který je řešiteli k dispozici.

Elliot Chikofsky definuje v [CHIK96] reversní inženýrství jako „...proces vedoucí k porozumění struktury a vnitřních vztahů systému. ... Reversní inženýrství se týká všech tří základních aspektů systému – dat, procesu a řízení." Z uvedeného je zřejmé, že techniky a nástroje datového reversního inženýrství (DRE) tvoří podmnožinu téhož v reversním inženýrství (RE) – zatímco RE se zabývá daty, procesem a řízením, DRE se týká pouze dat. Tomuto odpovídá i jedna z možných definic DRE - „Datové reversní inženýrství je sada nástrojů a metod, které pomáhají určit strukturu, účel a význam (podnikových) dat" [DAVI00].

Datové reversní inženýrství není doménou pouze relačních databází. V 80. a 90. letech minulého století, v době jejich masového rozšíření, bylo k jejich naplnění třeba zpracovat data z existujících struktur. To byly často „ploché" soubory bez hierarchických vztahů (tzv. konvenční systémy) nebo síťové a hierarchické databáze. Prvně zmíněný problém řešily mj. práce [CASA83] a [DAVI85], druhý např. [NAVA87] nebo [FONG93]. Velký praktický význam této problematiky dal vzniknout i CASE nástrojům jako např. DB-MAIN [HAIN95].

6.1 Reversní inženýrství relačních databází

V případě reversního inženýrství relačních databází (RDRE) je základním vstupem databázové schéma, cílovým výstupem pak konceptuální model.

Na rozdíl od relačního návrhu je automatizace zpětného procesu velmi komplikovaná, ne-li nemožná. Existuje mnoho problémů, které přímočaré transformaci zabraňují ([BLAH95], [HAIN98], [PETI94] a [PREM93]). Jejich příčiny je možno kategorizovat takto:

- Nedostatečnost relačního modelu:

Během transformace konceptuálního modelu do relačního dochází ke ztrátě informace. Konstrukty relačního modelu jsou omezené a část sémantiky tak musí být uložena mimo databázové schéma.

- Výkonová nebo zdrojová optimalizace:

Databázové schéma může obsahovat takové struktury, které přímo nesouvisejí s modelovanou skutečností. Tyto byly doplněny jako důsledek optimalizačních úprav a proto nemají v konceptuálním modelu svůj obraz. Charakteristickým rysem je redundance dat spojená s denormalizací.

- Implicitní struktury:

Některé konstrukce vyjádřitelné v jazyce DDL naopak v databázovém schématu mohou chybět. Z jistých důvodů byla jejich hypotetická funkčnost realizována v aplikační vrstvě.

- Nekvalitní návrh:

Většina autorovi dostupných databázových schémat (skutečně nasazených informačních systémů) vykazuje nedostatky, které lze přisoudit nekvalitně

provedenému návrhu. Některé z nalezených nesrovnalostí nemusejí mít přímý vliv na správnost případné zpětné transformace, řada z nich je však pro tento proces kritická. Závažnost těchto chyb přitom může být pro funkčnost systému nízká až nulová.

- Jinak nerelevantní struktury:

Databázová schémata mohou obsahovat struktury, které jsou z různých důvodů pro zpětnou transformaci nežádoucí. Jde například o pozůstatky opuštěných nebo nedokončených funkčností, které v důsledku minimální údržby schématu přetrvaly. Jiným příkladem jsou procesní a řídící struktury bez obrazu v konceptuálním modelu.

6.2 Možné metody sémantického obohacení

6.2.1 Analýza DDL

Většina prací zabývajících se reversním inženýrstvím relačních databází je založena na analýze databázového schématu. Podstatné pro tento přístup jsou deklarace primárních a cizích klíčů. Tento přístup je algoritmizován např. v [ALHA02], podmínkou jsou však zásahy uživatele, který v mnoha situacích zastupuje „rozhodovacího agenta“. Tato práce částečně vychází z [CHIA94].

Schématem se kromě skutečně vytvořených databázových struktur rozumí také kód v jazyce DDL, který je k dispozici jako „zdrojový kód“ sloužící k vygenerování schématu, běžně je také možné jeho dodatečné vygenerování. Definice schématu je také obsažena v systémových tabulkách SŘBD.

Přístupy založené na analýze schématu předpokládají, že toto schéma je v souladu s minimálně třetí normální formou. Vzhledem ke skutečnostem popsaným v předchozí kapitole a tudíž prakticky nereálným nárokům

kladeným na kvalitu databázového schématu se některé práce zabývají využitím jiných zdrojů sémantických informací.

6.2.2 Analýza DML

Jedním z dalších možných způsobů doplnění sémantiky je analýza příkazů jazyka DML. Prostředkem takto založených metod je identifikace funkčních a inkluzních závislostí na úrovni schématu.

Příkazy jazyka DML mohou být součástí systémových tabulek (PLAN_TABLE…), interpretovaných aplikací (HTML…) nebo zdrojových kódů kompilovaných aplikací (Java aplikace, uložené procedury...). Zároveň je možné využít deklarace pohledů, které jsou také sestavovány dotazy DML.

Andersson v [ANDE94] zkoumá podmínky spojení a právě deklarace pohledů (při nejednoznačných interpretacích vyžaduje zásah uživatele). Barbar předpokládá existenci expertů, kteří databázi v průběhu určité doby dotazují, a jejichž dotazy (uchovávané v „Query Base“) podrobuje analýze ([BARB01]). Astrova používá v [ASTR04] jako zdroj dotazů webové formuláře.

6.2.3 Analýza dat

Sémantické informace je také možno „vytěžit“ přímo z dat. Jejich statistická analýza však může být (nejen) časově velmi náročná. Principem je nalézání funkčních a inkluzních závislostí na datové úrovni.

Premerlani a Blaha navrhli v [PREM94] komplikovaný postup, který využívá sadu nesourodých metod a množství uživatelských interakcí. Vstupem jsou mj. unikátní indexy pro nalezení klíčů, jsou prováděny rozsáhlé iterativní analýzy dat (mj. k odhalení generalizací a agregací) a zkoumáno názvosloví atributů (odhalení inkluzních závislostí).

Tari v [TARI97] vyžaduje prvotní uživatelskou klasifikaci relací, závislosti pak získává analýzou schématu a právě dat, kde sleduje korelaci klíčů mezi relacemi a korelaci záznamů uvnitř relací.

6.2.4 Porovnání metod

Závislosti zjištěné z testů dat i schématu (DML) jsou rovnocenné, v ideálním (prakticky ale těžko dosažitelném) případě vedou všechny tři popsané metody ke stejným výsledkům, nejuspokojivější výsledky dává jejich kombinované provedení.

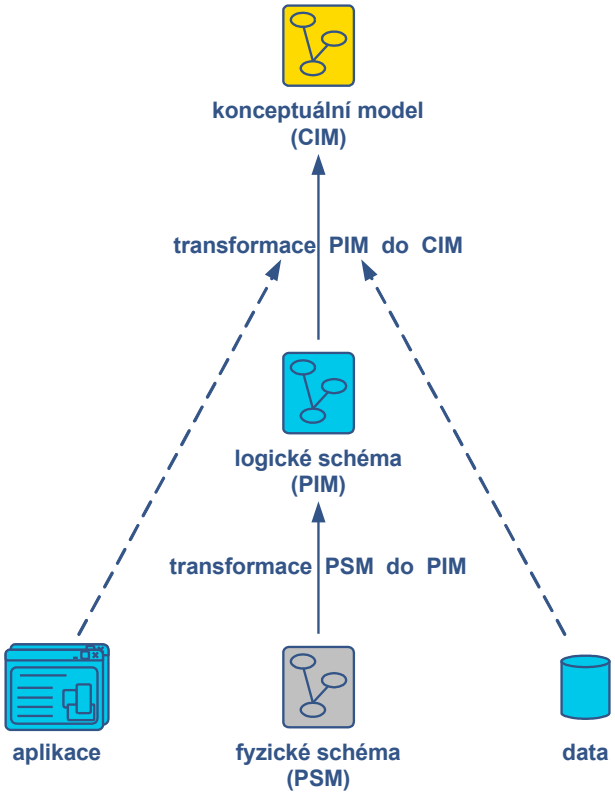


schéma 8: zdroje sémantických informací

Pro nejednoznačné interpretace a řešení konfliktů je ale stále potřeba lidského zásahu, což je jedním z hlavních faktorů vylučující plně automatizované zpracování.

7 Princip metody transformace

Znázornění obecného procesu transformace je uvedeno na následujícím schématu:

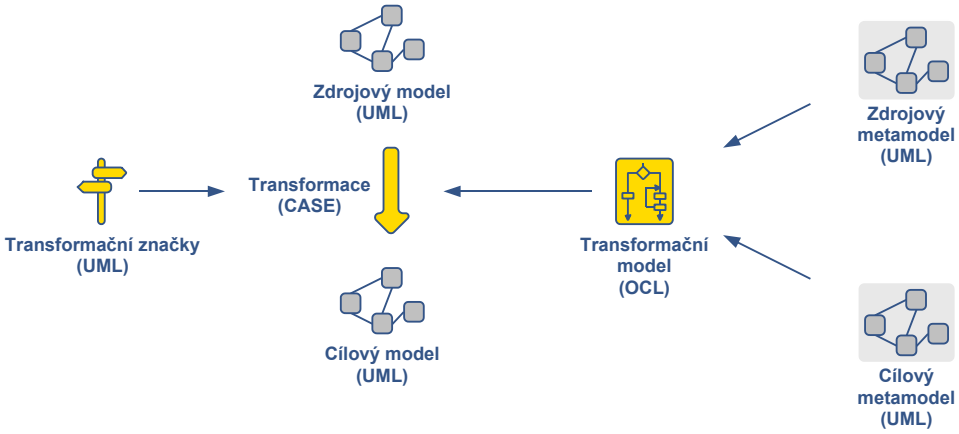


schéma 9: obecný princip transformací

Při transformacích je vycházeno z modelu MDA, který chápe transformaci jako vytvoření cílového modelu ze zdrojového pomocí:

- zdrojového modelu,
- zdrojového metamodelu,
- cílového metamodelu,
- transformačního modelu,
- transformačních značek.

Transformační model obsahuje „předpis“ pro mapování elementů modelu zdrojového do elementů modelů cílového. Tento obecný předpis obsahuje odkazy na elementy zdrojového i cílového metamodelu. Při aplikaci na

konkrétní zdrojový model jsou v něm obsažené elementy (definované v jeho metamodelu) transformovány podle pravidel transformačního modelu do cílového modelu. Protože však zdrojový a cílový model nemusejí být jednoznačně transformovatelné (např. ve zdrojovém metamodelu jsou elementy, které lze podle situace transformovat do 2 různých elementů cílového metamodelu), používají se pro rozlišení jednotlivých (jednoznačných) případů transformační značky. Ty pak nejednoznačné situace upřesňují tak, že může být aplikováno konkrétní pravidlo transformačního modelu.

Oba uvedené modely (vč. jejich metamodelů) jsou ve shodě s [OMG_01] vyjádřené jazykem UML. Proto jsou pro transformace uvedené v dalších kapitolách vždy v UML definovány oba metamodely.

Požadavky na formát transformačního modelu v [OMG_01] nejsou uvedeny. Implementace MDA používají většinou vlastní jazyky. Aby byla uvedená metodika platformně nezávislá, je v transformacích použit jazyk OCL[11].

Transformace popsané transformačním modelem mohou obsahovat odkazy na zmíněné značky. Těmito značkami, opět vyjádřenými jazykem UML, je doplněn zdrojový model, který je vstupem do transformace.

Každá klíčová transformace (transformace uvedená v kapitole *Transformace relačního databázového modelu do objektového*) je uvedena nejdříve matematickým

[11] V praxi je ale použit jazyk zvoleného CASE nástroje (podporujícího MDA). Vytvoření nástroje pro překlad OCL do jazyka používaného v produktu Enterprise Architect (EA) společnosti Sparx Systems, který autor této práce pro transformace používá, je jedním z cílů navazujících prací (viz kapitola *Závěr, zhodnocení, další práce*). Ukázka ekvivalentních zápisů v OCL a běžného programovacího jazyka je uvedena v kapitole *Přílohy*.

zápisem (lambda kalkul). Následuje důkaz úplnosti transformace, na závěr je doplněn zápis v OCL.

Každá transformace uvedená v kapitole *Očištění a normalizace relačního datového modelu* je pouze slovně popsána, její OCL předpis je uveden v kapitole *Přílohy*.

Ukázka možností transformace v EA je v kapitole *Přílohy*.

8 Očištění a normalizace relačního datového modelu

Cílem této kapitoly je popsat způsob získání „správného“ relačního modelu na úrovni PIM. (Správným modelem se rozumí takový model, který je dále uvedenými postupy zpracovatelný a ze kterého po aplikaci těchto postupů získáme uspokojivé výsledky). To zahrnuje jeho vytvoření z implementačního modelu PSM a jeho očištění a normalizaci.

8.1 Získání logického relačního datového modelu (PSM → PIM)

8.1.1 Fyzický datový model (PSM)

Jak je patrné mj. ze schématu v kapitole *MDA (Model Driven Architecture)*, z pohledu MDA je **fyzický datový model ekvivalentní modelu platformě závislému** (PSM).

PSM, ať už v jakékoli formě, kompletně a přesně definuje strukturu databáze tak, jak je implementována. Zahrnuje totiž nejen „logickou“ úroveň, ale i všechny ostatní údaje potřebné k úplnému založení relační databáze. Právě z tohoto důvodu je část informace uložené v PSM modelu pro datovou transformaci nerelevantní. Jsou to např. specifikace tabulkových prostor a alokace tabulek do nich, různé další informace o fyzickém uložení, atd.

Pokud je PSM uložen v nějakém specifickém CASE nástroji, je z něj možné SQL DDL kód jednoduše vygenerovat. Je dokonce možné (a běžné), aby

takový CASE nástroj komunikoval přímo se SŘBD a bez nutnosti generování těchto definičních skriptů realizoval změny provedené ve vizuálním prostředí.

Z důvodů rozšířenosti a kompatibility je zde ovšem na PSM model nahlíženo jako na množinu příkazů jazyka SQL, konkrétně standardu SQL-92[12]. Spíše než o množinu jde o sice o sekvenci, neboť pořadí provádění jednotlivých příkazů musí být ve většině případů dodrženo, nicméně toto pořadí je dáno a nemá smysl se souvisejícími pravidly dále zaobírat (v dalších kapitolách je předpokládáno, že je dodrženo).

Je-li jazyk PSM omezen na definiční část (DDL), je třeba při mapování na PIM jeho konstrukty dále omezit. Z množiny příkazů jazyka SQL-92 jsou pro mapování relevantní pouze níže uvedené, a to s uvedenou syntaxí.

```
Create schema schema_name

Create table schema_name.table_name (column_name
column_type, ...)

Alter table schema_name.table_name add {unique
(column_name, ...) | primary key (column_name, ...)
| foreign key (column_name, ...) references
table_name (column_name, ...)}
```

8.1.1.1 Import modelu PSM do CASE nástroje

Pokud to konkrétní případ umožňuje, je samozřejmě možné jako model PSM využít přímo ten model, který je uchováván ve WISIWIG nástroji pro datové

[12] Tohoto omezení se není třeba obávat, protože většina DBMS produktů obsahuje nástroje umožňující zpětně vygenerovat SQL DDL i v tomto formátu. Totéž je možné provést u většiny CASE nástrojů, které nejsou s konkrétním produktem svázány.

modelování. Tyto nástroje jsou buď součástí databázového produktu, nebo jsou generické a pro komunikaci s DBMS používají rozhraní ODBC. Běžnou funkcí těchto nástrojů je export modelu do souboru. Abychom byly schopni tento soubor nahrát do CASE nástroje, musí být jeho formát podporován, tj. mělo by jít o standard XMI. Dále musí být pro import definováno, jaké struktury jsou pro naše potřeby relevantní (jsou to elementy uvedené v následující kapitole, tedy elementy modelu PIM) a jak tyto, pro CASE neznámé struktury, modelovat (tj. každý element nativní zdrojovému modelu musí mít definovaný protějšek PIM).

V tomto případě by se nejednalo o transformaci, ale o pouhý přepis vybraných typů elementů PSM do PIM.

8.1.2 Logický datový model (PIM)

Platformě nezávislý model již (pro účely této práce) obsahuje pouze relevantní struktury vyjádřitelné v jazyce srozumitelném libovolné implementaci RDBMS.

Mezi takové struktury patří:

- schéma databáze
- schéma relace
- atributy
- klíče

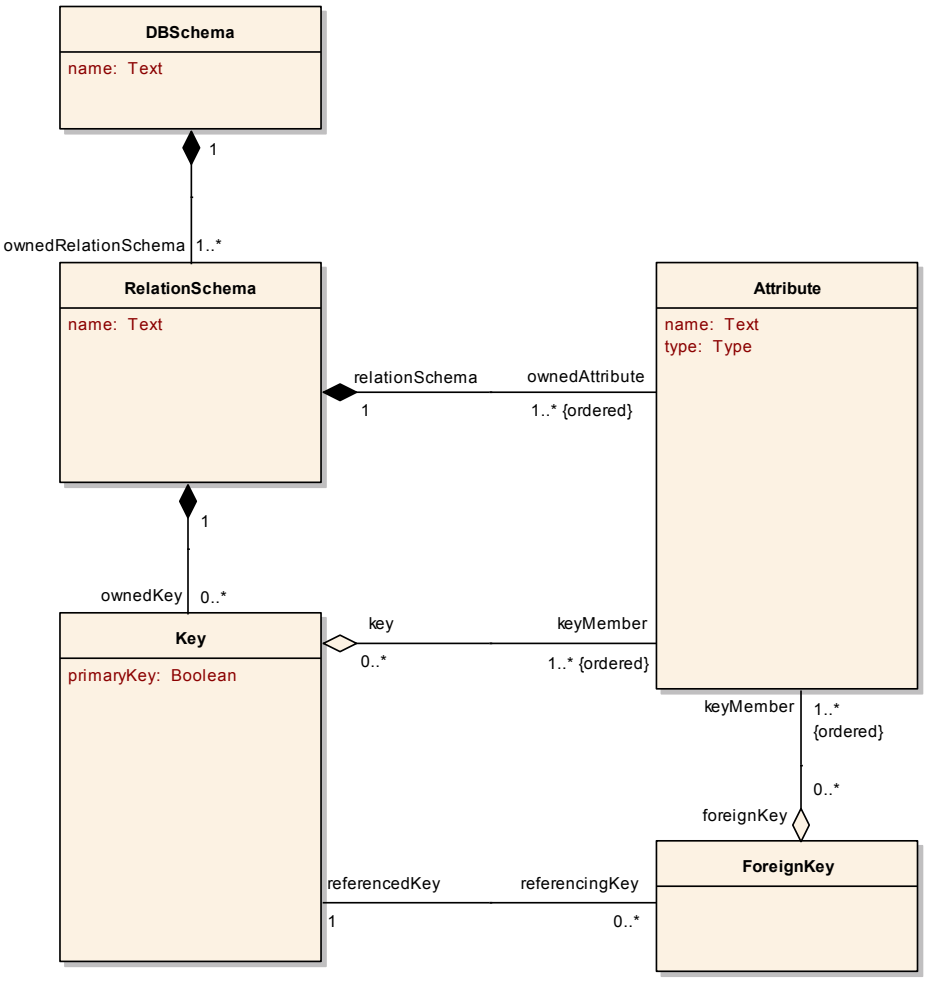


schéma 10: základní relační metamodel

Schéma databáze (DBSchema) je složeno ze schémat relací. Má jméno, které tvoří kontext obsažených chémat relací, další jeho vlastnosti, které jsou v RDBMS definovány, nejsou podstatné. Pokud je schéma databáze prázdné, nemá smysl.

Schéma relace (RelationSchema) definuje relaci, tedy tabulku relační databáze. Množina všech schémat relací tvoří schéma databáze.

Vlastní atributy, které jsou v jeho kontextu řazeny (ačkoli toto řazení má v RDBMS vliv pouze na fyzické uložení dat, kvůli dalším transformacím a migraci dat je vhodné jej zachovat). Atributů může být libovolné množství, minimálně však jeden. Pokud existuje relace bez atributů, nemá pro zamýšlenou transformaci smysl.

Kromě atributů vlastní schéma relace také klíče. Schéma relace by mělo obsahovat alespoň jeden klíč – primární. Mohou však existovat relace odporující první normální formě, kdy primární klíč není definován. Pak je třeba později vhodný primární klíč určit (viz kapitola *Normalizace (PIM → PIM)*).

Atribut (Attribute) reprezentuje jeden z 2 stavebních kamenů vnitřní struktury schématu relace. V kontextu schématu relace je atribut jednoznačně určen jménem. Dále je každému atributu přiřazen **datový typ**.

Klíč (Key) je druhým vlastněným prvkem schématu relace. Seskupuje její atributy. Klíč, který je součástí schématu relace, může obsahovat jeden nebo více atributů dané relace. Pořadí atributů zde většinou nehraje roli, nicméně v případě odkazování cizích klíčů (viz dále) je pořadí podstatné. Klíč bez atributů opět nemá smysl.

Cizí klíč (ForeignKey) relace sdružuje atributy, které dále odpovídají nějakému z klíčů vlastní nebo cizí relace. Jednotlivé atributy cizího klíče pak v daném pořadí odkazují na atributy odkazovaného klíče. V tomto vztahu je pak cizí klíč klíčem odkazujícím.

8.1.2.1 Potřebné elementy UML

Vzhledem ke známým charakteristikám prvků SQL DDL (uvedeným v předchozí kapitole) je třeba najít nebo vytvořit takové UML elementy, které by byly „kompatibilní“, tedy splňovaly uvedené požadavky. Těmi jsou:

PIM element	**Požadavky**
DBSchema	• musí být pojmenovatelný • musí mít schopnost vlastnit další elementy
RelationSchema	• musí být pojmenovatelný • musí mít schopnost vlastnit další elementy a podporovat jejich řazení
Attribute	• musí být pojmenovatelný • musí být typovatelný
Key	• musí mít schopnost odkazovat na další elementy a podporovat řazení vazby • musí mít schopnost vlastnit atributy
ForeignKey	• musí mít schopnost odkazovat na další elementy a podporovat řazení vazby

tabulka 3: elementy relačního metamodelu

Kromě uvedených požadavků existují ještě další, vyplývající z kapitoly *Normalizace (PIM → PIM)*. Těmi jsou zejména schopnost schématu relace vlastnit metody.

Protože v UML jsou elementy požadovaných vlastností definovány, je v následující kapitole popsáno, z jakých předků budou nové elementy požadované vlastnosti dědit.

8.1.2.2 Způsob rozšíření UML

Elementy modelu PIM jsou uvedeny v předchozí kapitole. Zde je popsáno začlenění do frameworku UML. Protože nově definované elementy budou do UML začleněny v novém balíčku „Relations“ pomocí vazby PackageMerge, musí být do balíčku umístitelné. Každý element tedy musí být potomkem elementu PackageableElement. Ten sám dědí z NamedElement a proto je tímto požadavek na pojmenovatelnost splněn. Začlenění do UML struktury, konkrétně do vnitřní struktury základního balíčku Classes, je uvedeno na následujícím schématu:

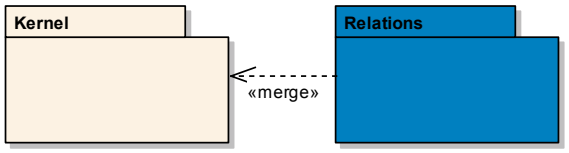


schéma 11: rozšíření UML o balíček Relations

Výběr konkrétních UML elementů a redefinice vlastností těchto elementů (Features) je uvedena na následujícím diagramu:

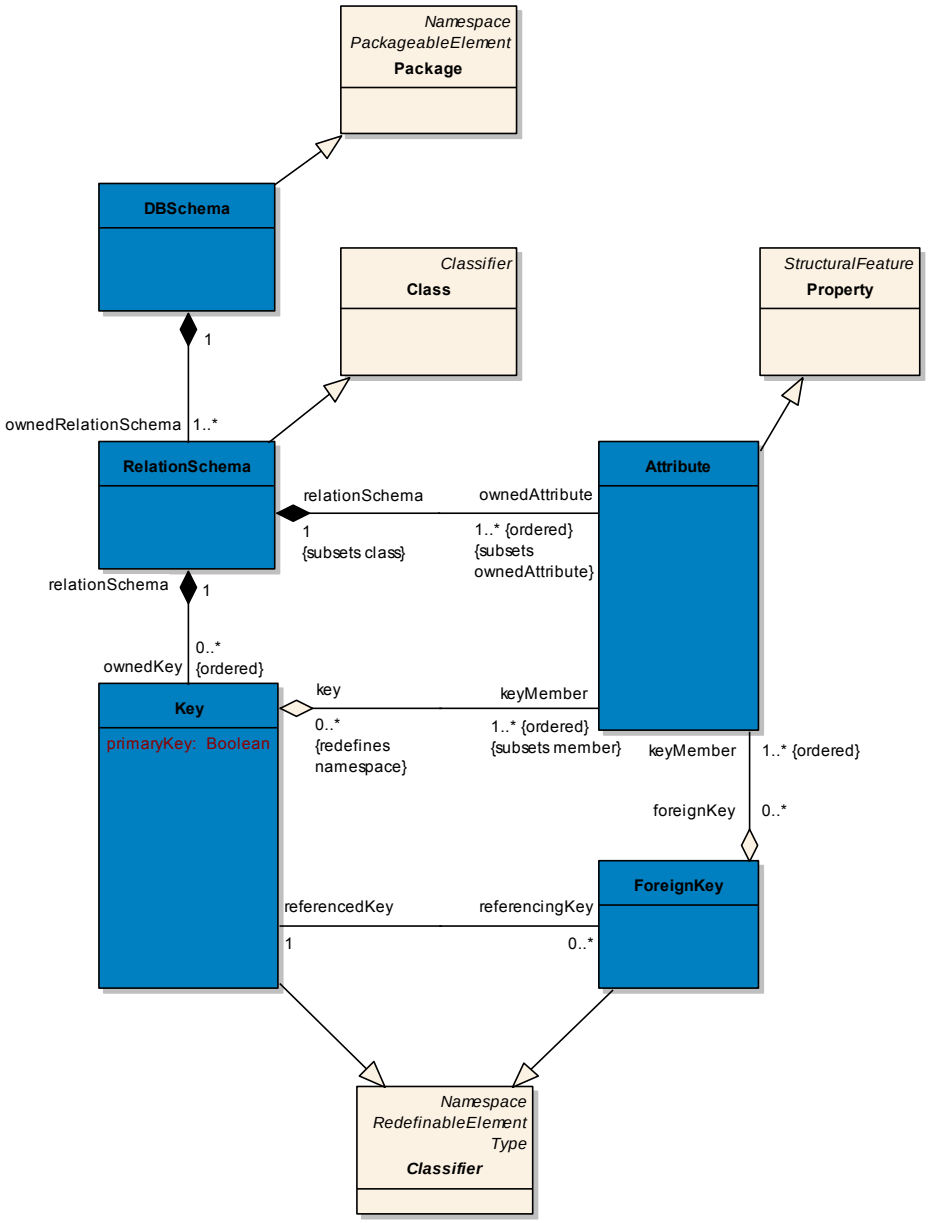


schéma 12: obsah balíčku Relations

8.1.3 Transformace

Jako transformační jazyk je v práci použit jazyk OCL. Jedná se o deklarativní jazyk z dílny OMG. Transformace se tímto jazykem popisují pomocí vstupních a výstupních podmínek transformace – preconditions a postconditions. Vstupní podmínka definuje omezení platná pro provedení transformace, případně popisuje stav pře transformací. Výstupní podmínka naopak popisuje stav modelů po transformaci, tj. např. uvádí, jaké nové elementy vznikly, jaké mají tyto nové elementy vlastnosti, jaké existující elementy byly změněny nebo jaké elementy byly zcela odstraněny. Pokud není uvedena vstupní podmínka, je touto podmínkou pouze existence zdrojového modelu.

Transformace probíhá podle následujícího schématu. Transformační model vyjádřený transformačními pravidly obsahuje veškeré informace potřebné k provedení transformace:

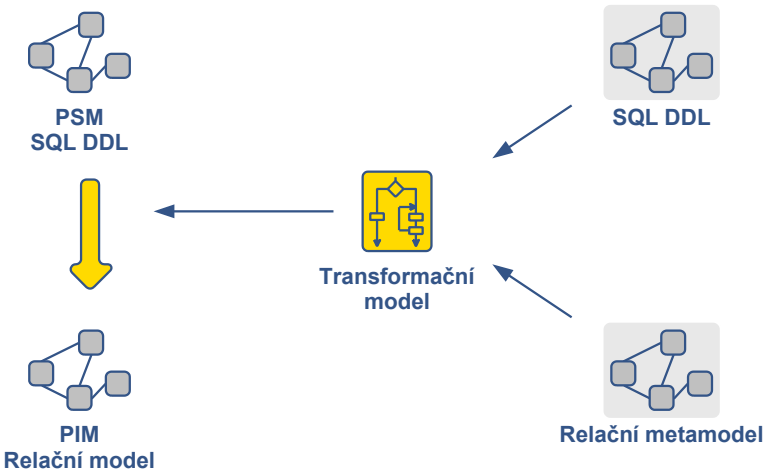


schéma 13: transformace do PIM

Popis postupu transformace z jazyka PSM do modelu PIM, tj. jaké elementy a vazby vznikají při zpracování definovaných příkazů SQL, je uvedeno v kapitole *Přílohy*.

8.2 Konsolidace (PIM → PIM)

8.2.1 Označení nerelevantních schémat relací

V datovém modelu mohou existovat relace, které nenesou informaci o modelovaném universu. Typicky jsou to různé konfigurační tabulky, tabulky logů a jiné „nízkoúrovňové“, případně řídící struktury, u kterých je transformace to objektového modelu zbytečná. Bývají úzce spjaty s technologií, která je nahrazována. Tyto tabulky, případně sloupce je třeba z procesu transformace vyjmout.

Protože jejich automatické rozpoznání je téměř nemožné, je nutné jejich označení uživatelem. Vhodným nástrojem pro to je nastavení značky (tag) a její hodnoty (tagged value):

Značka (tag)	**Hodnota**
`Abandoned:Boolean`	true

tabulka 4: značky nerelevantních schémat relací

Takto označené elementy nebudou do další tranformace vstupovat.

Poznámka: Autor této práce zvolil pro opatřování elementů modelu uživatelskými značkami (marks), které ovlivňují proces transformace, nástroj, jehož český název je také značka (tag). Jde o shodu okolností, tyto značky spolu logicky nemají nic společného.

8.2.2 Doplnění implicitních klíčů

Jak je uvedeno v kapitole *Reversní inženýrství relačních databází*, ne všechny struktury bývají explicitně definovány. Pokud např. aplikační vrstva účinně zamezuje nekonzistenci v datech, mohou např. skupiny atributů, které jsou ze své podstaty unikátní, unikátními zůstat i bez příslušné definice. Chceme-li ale vytvořit správný konceptuální a potažmo objektový model, je třeba tyto nedostatky odstranit. Mezi potenciální nedostatečnosti podobného druhu patří:

- chybějící klíče (candidate keys) a
- chybějící cizí klíče.

Mějme sadu atributů tabulky, které jsou členy implicitního klíče. Každý takový atribut obdrží od uživatele značky `CKSequence` a `CKMemberSequence`:

Značka (tag)	**Hodnota**
`CKSequence:Integer`	pořadí klíče v rámci relace
`CKMemberSequence:Integer`	pořadí atributu v rámci klíče

tabulka 5: značky implicitních klíčů

Takto označovány jsou pouze nově vytvářené klíče, existující klíče mají své pořadové číslo určené pořadím multiplicity. Kvůli zamezení konfliktů při navazování cizích klíčů je však třeba, aby nově přiřazovaná pořadí klíčů uvnitř relace začínala od čísla následujícího po pořadovém čísle posledního existujícího klíče (tj. `self.relationSchema.ownedKey->size() + 1`). Pořadí atributu je důležité v případech, kdy na označovaný klíč bude

odkazovat klíč cizí – pokud toto zamýšleno není, může být pořadové číslo vynecháno a při následné transformaci doplněno automaticky.

Stejně tak se přistupuje k cizím klíčům – přiřazované značky jsou `FKSequence`, `ReferencedCKOwner`, `ReferencedCKSequence` a `ReferencedCKMemberSequence`:

Značka (tag)	**Hodnota**
`FKSequence:Integer`	pořadí cizího klíče, definované pouze pro účely rozlišení mezi nově nalezenými
`ReferencedCKOwner:String`	název[13] relace vlastnící protilehlý klíč
`ReferencedCKSequence:Integer`	pořadí protilehlého klíče v relaci
`ReferencedCKMemberSequence: Integer`	pořadí atributu v protilehlém klíči (odpovídá pořadí odpovídajícího atributu v cizím klíči)

tabulka 6: značky cizích klíčů

[13] Vzhledem k tomu, že se stále pohybujeme v modelu, kde jsou elementy jednoznačně určeny i jménem (v rámci svého jmenného prostoru), můžeme na ně pomocí jména odkazovat.

Uvedené značky mohou mít samozřejmě libovolný název a formát za předpokladu, že značky v případě klíče obsahují jeho identifikaci a pořadové číslo atributu. Pokud je označován atribut cizího klíče, je nutné ve značce specifikovat protilehlou relaci, její klíč, a klíčový prvek. Totéž platí pro všechny ostatní značky (musí být samozřejmě upravena specifikace příslušné transformace).

8.2.2.1 Způsoby identifikace implicitních struktur

Protože manuální označování implicitních klíčů a cizích klíčů způsobem popsaným v předchozí kapitole vyžaduje od uživatele znalost modelované domény a znalost práce s CASE nástrojem (pokud pro tyto účely není k dispozici specializované uživatelské rozhraní), může být tento postup problematický. Uživatel disponujícími takovými znalostmi je většinou schopen zde popsané transformace provést ručně, ať už s podporou principů zde uvedených, nebo jen intuitivně. Takovému uživateli pak může být překážkou snad jen množství úsilí nutného ke konsolidaci rozsáhlého datového modelu.

Existuje však způsob, jak vytyčeného cíle dosáhnout i automatizovaně. Některé obecné koncepty jsou uvedeny v kapitole *Možné metody sémantického obohacení*. Pokud zůstaneme u problému nalezení implicitních klíčů, nabízí se tyto zdroje informací, které lze pro tento účel s úspěchem použít. Je jimi:

- programový kód aplikační vrstvy
- definiční skripty databáze
- data samotná

Aplikováním postupů založených na níže uvedených principech je dosaženo stejného výsledku, tedy přiřazení značek (tags) atributům relací. Kvalitu

automatizace samozřejmě nelze 100% zaručit (není-li dále uvedeno jinak) a uživatel by měl veškeré značkování dodatečně ověřit!

8.2.2.1.1 Programový kód aplikační vrstvy

Spíše než aplikační vrstva jako celek je pro situaci hodnotná zejména ta její část, která zajišťuje přístup k datům v relační databázi. Tato „podvrstva" nutně obsahuje dotazovací příkazy jazyka SQL, tedy jeho DML části. Protože cizí klíče, jejichž hlavním účelem je zajištění kozistence dat, ve spojení s klíči protilehlých relací navíc většinou vystupují v operaci spojení (equi-join), lze se opačnou cestou, tedy analýzou spojování relací, dobrat k potenciálním klíčům a cizím klíčům. V SQL takový dotaz vypadá následovně:

```
select A.column_name, ... from table_nameA A,
table_nameB B, ... where A.column_name ?
B.column_name and B.column_name ?
value_specification
```

, kde ? odpovídá operátorům >(=), <(=), =, <>, in, případně like.

Pouhým vyhledáním atributů předcházejících nebo následujících operátor však můžeme mylně zahrnout i ty atributy, které korektně v žádném klíči obsaženy nejsou, ale slouží k výběru záznamů odpovídajících aplikační proměnné, případně konstanty, nebo vyjádření jiné logiky. Přestože některé z uvedených operátorů mimo tohoto použití mohou indikovat vztah mezi klíči, v dostatečné míře je toto empiricky průkazné pouze u operátoru ekvivalence. Abychom dále zpřesnili výběr atributů, je třeba uvažovat pouze ty, které jsou porovnávány s jinými atributy. Tato skutečnost je z programového kódu velmi dobře patrná (obvykle proměnné předchází otazník nebo jiný speciální znak). Přesto je jisté, že i v tomto výběru budou některé atributy zařazeny nesprávně. Opět je zde tedy nutný uživatelské ověření a případná náprava.

V případech, kdy je v dotazu dáváno na roveň více atributů stejných relací, je třeba všechny tyto atributy zahrnout mezi potenciálně klíčové.

Uvedeným způsobem dochází k identifikaci (sad) atributů, které jsou prvky buď klíče, nebo cizího klíče. Pokud je porovnáním se zdrojovým PIM modelem zjištěno, že jeden atribut (nebo jejich sada) tvoří klíč relace, je pak druhý atribut (nebo jejich sada) typicky cizím klíčem.

Bohužel, v obecném případě, kdy není definován klíč ani pro jednu stranu rovnice (a pro poslední z následujících bodů dokonce i v předchozím případě – viz mapování generalizace v relačním modelu) však mohou nastat následující situace:

- atributy relace A jsou klíčem, atributy relace B jsou cizím klíčem,
- atributy relace B jsou klíčem, atributy relace A jsou cizím klíčem,
- atributy relace A jsou klíčem, atributy relace B jsou klíčem.

Přestože poslední bod je teoreticky možný, je vysoce nepravděpodobný. Může nastat totiž jen ve dvou případech:

1. Relace A a relace B obsahují záznamy o totožných entitách – v tom případě je ovšem zbytečné a neúčelné tyto záznamy udržovat ve dvou relacích (pokud by se nejednalo o vyjádření generalizace).
2. Relace A a relace B obsahují záznamy o různých specializacích téže entity – pak je ale nelogické, aby tyto záznamy byly porovnávány.

Po vyloučení této eventuality zbývá pro oba případy (klíč vs. cizí klíč) tyto klíče rozlišit. K tomu je využito metody popsané v sekci Data.

8.2.2.1.2 Definiční skripty databáze

Těmito skripty se rozumí specifikace pohledů uvedená v kapitole *Fyzický datový model (PSM*. Blok `view_definition` totiž kromě nepovinného `(column_name, ...)` obsahuje klausuli `as select ...` popsanou v předchozí sekci. Postupy k získání potenciálních klíčů a cizích klíčů jsou tedy ekvivalentní.

8.2.2.1.3 Data

Narozdíl od přístupů uvedených v predchozích sekcích je tato metoda za splnění určitých podmínek vysoce efektivní. Stejné podmínky ji však znevýhodňují tak, že je časově a zdrojově velmi náročná. Jedná se o datovou analýzu záznamů jednotlivých relací. Jako jediná pracuje tato metoda s izolovanými relacemi a nezkoumá tedy vztahy mezi atributy různých relací.

Principem této metody je hledání unikátních atributů (nebo sad atributů) každé relace. Postupně je testována hypotéza, že každá kombinace atributů relace je jejím klíčem, neboli je unikátní. Testování začíná od mocnosti klíče = 1 (tj. jednoatributového klíče) a přechází k vyšším mocnostem. Testovány nejsou kombinace, o nichž je ze zdrojového modelu PIM známo, že klíčovými jsou – hypotéza je v těchto případech potvrzena. Za předpokladu neomezených zdrojů je pro každou relaci nalezen vždy alespoň jeden klíč, a to úplná sada atributů relace. Tento fakt vyplývá ze skutečnosti, že každý záznam je určen vždy pouze hodnotami jednotlivých atributů, a že relace je množina bez opakujících se prvků.

V praxi je ovšem uvedený předpoklad těžko dosažitelný, a proto je vhodné definovat mezní mocnost n, po jejíž dosažení testování nepokračuje. Každé další zvýšení mocnosti má na náročnost nelineární dopad. Zkušenosti

napovídají, že vhodnou mezí je mocnost rovna 5, záleží ovšem na velikosti relace.

Každý takto nalezený klíč však nemusí být úplný – může se vždy jednat o podklíč. Pokud je tedy při dosažení mezní mocnosti identifikován klíč, je třeba (pouze) pro tuto sadu atributů dále v testování pokračovat až do vyloučení všech hypotéz o klíčovosti sady o mocnosti n+1. Pokud se některou hypotézu nepodařilo vyloučit, pokračuje se pro danou sadu n+1 atributů v mocnosti n+2.

Uvedená metoda má další nespornou výhodu – lze ji použít k ověření správnosti klíčů označených uživatelem či extrahovaných z aplikační vrstvy.

8.2.3 Transformace

Transformace je opět vyjádřena jazykem OCL a jako zdrojový a cílový model je z hlediska MDA model na úrovni PIM. Kód OCL je v kapitole *Přílohy*.

Transformace probíhá podle následujícího schématu. Transformační značky získané z různých zdrojů (viz níže) zcela definují transformaci:

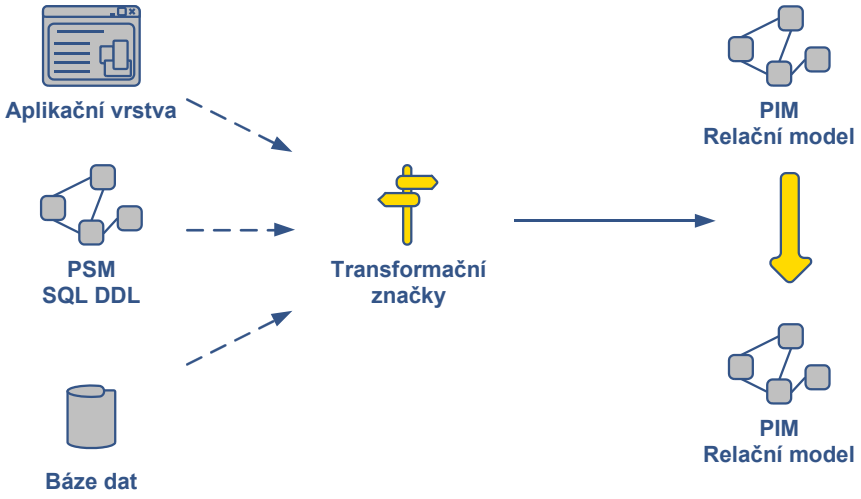


schéma 14: konsolidace

Všechny získané značky (tj. označení klíčů a cizích klíčů) jsou v transformaci zpracovány a podle nich jsou z modelu PIM odstraněny relace a doplněny klíče a cizí klíče. Strukturální změny (myšleno je přesouvání atributů mezi schématy relací a vznik nových schémat relací) nejsou prováděny.

8.3 Normalizace (PIM → PIM)

Protože databázový model často nesplňuje ani základní kritéria správného návrhu, je třeba zajistit, aby struktura odpovídala normálním formám. Aby normalizace mohla proběhnout automaticky, prostředky MDA, je třeba definovat UML elementy zdrojového z cílového modelu a způsob transformace. Z pohledu MDA je zdrojový a cílový model stejný – PIM. Jsou stanoveny elementy rozšiřující strukturu UML o možnost zachycení funkčních závislostí.

Celá normalizace pak probíhá podle následujícího schématu. Nejprve je podle transformačních značek provedena konsolidace modelu a poté, již pouze na základě transformačního modelu, je provedena normalizace:

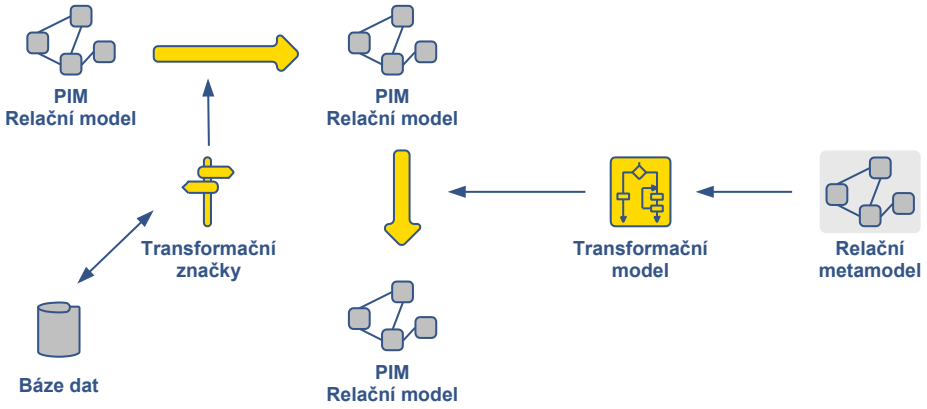


schéma 15: normalizace

Nejprve jsou doplněny funkční závislosti transformací z modelu doplněného příslušnými značkami, poté je podle pravidel dekompozice model transformován na model splňující vyšší a vyšší normální formu, dokud není dosažena BCNF. Pokud se již model v takové normální formě nachází, žádná normalizační transformace neprobíhá.

8.3.1 Způsob rozšíření UML

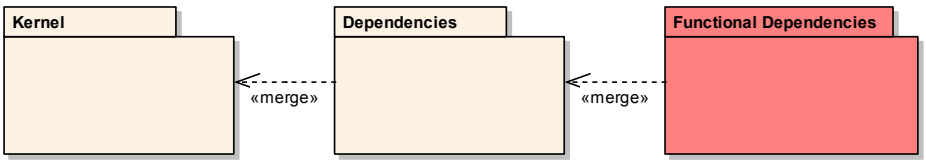


schéma 16: rozšíření UML o balíček Functional Dependencies

Protože jsou normální formy definovány pomocí funkčních závislostí, je třeba i tyto závislosti do UML zavést:

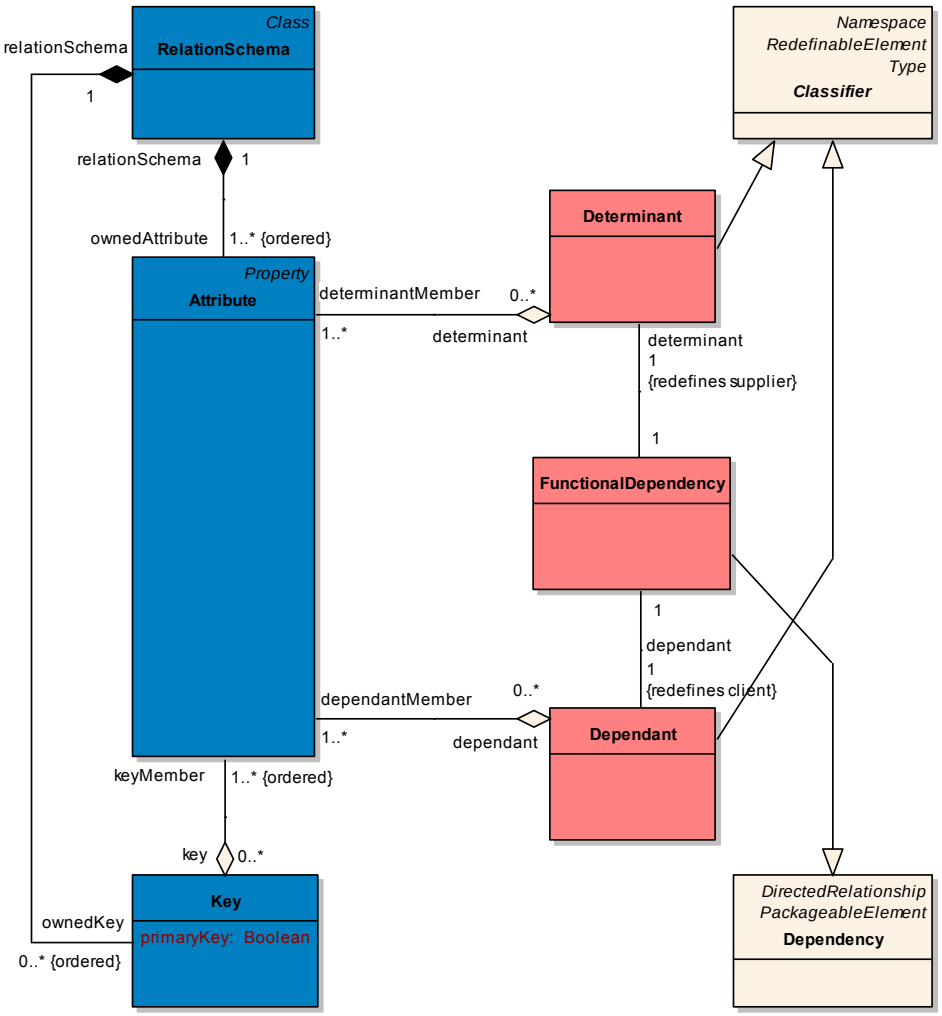


schéma 17: obsah balíčku Functional Dependencies 1

Funkční závislost, potomek obecné závislosti, říká, že atribut (nebo jejich sada) závisí na jiném atributu (nebo jejich sadě) téhož schématu relace. Nezávislá strana funkční závislosti je označena jako determinant, závislá pak jako dependant.

Při hledání determinantů a dependantů je třeba vycházet z analýzy dat, podobně jako při doplňování implicitních klíčů. Funkční závislost je v kontextu relace definována takto:

$$Dt \rightarrow Dp \Leftrightarrow \forall\, t, u;\ \forall\, a, b;\ (t.a = u.a) \Rightarrow (t.b = u.b),$$

kde *Dt* je determinant, *Dp* dependant, *t* a *u* jsou záznamy relace, *a* hodnota libovolného atributu determinantu a *b* hodnota libovolného atributu dependantu. Jinými slovy, dependant je na determinantu závislý (což je tímto označením předjímáno) právě tehdy, když pro záznamy ze všech různých záznamů relace, kde shodují hodnoty všech atributů determinantu, platí, že se shodují i hodnoty každého z atributů dependantu. Pokud by hodnoty determinantu v relaci byly unikátní, je determinant klíčem. Klíč je potom speciálním případem determinantu, kdy dependant tvoří všechny ostatní atributy relace.

Z uvedené implikace dále vyplývá, že za předpokladu shody hodnot všech atributů determinantu jsou i hodnoty libovolného *jednotlivého* atributu dependantu shodné. Funkční závislost tedy nemusíme posuzovat jako vztah mezi sadami atributů, ale jako sadu „parciálních" funkčních závislostí mezi determinantem jako sadou atributů a dependantem jako jednotlivým atributem. Suma všech takových parciálních funkčních závislostí se shodným determinantem pak vytváří funkční závislost (sloučením „parciálních" dependantů).

Zatímco pro jeden determinant a neparciální dependant neexistuje nikdy více než jedna neparciální funkční závislost (viz předchozí schéma), může pro něho existovat více parciálních funkčních závislostí s parciálními dependanty (atributy). Parciální dependanty mohou být zároveň členy více parciálních funkčních závislostí. Tento princip, který znamená zjednodušení dalších tranformací, je zachycen na následujícím, výsekovém schématu:

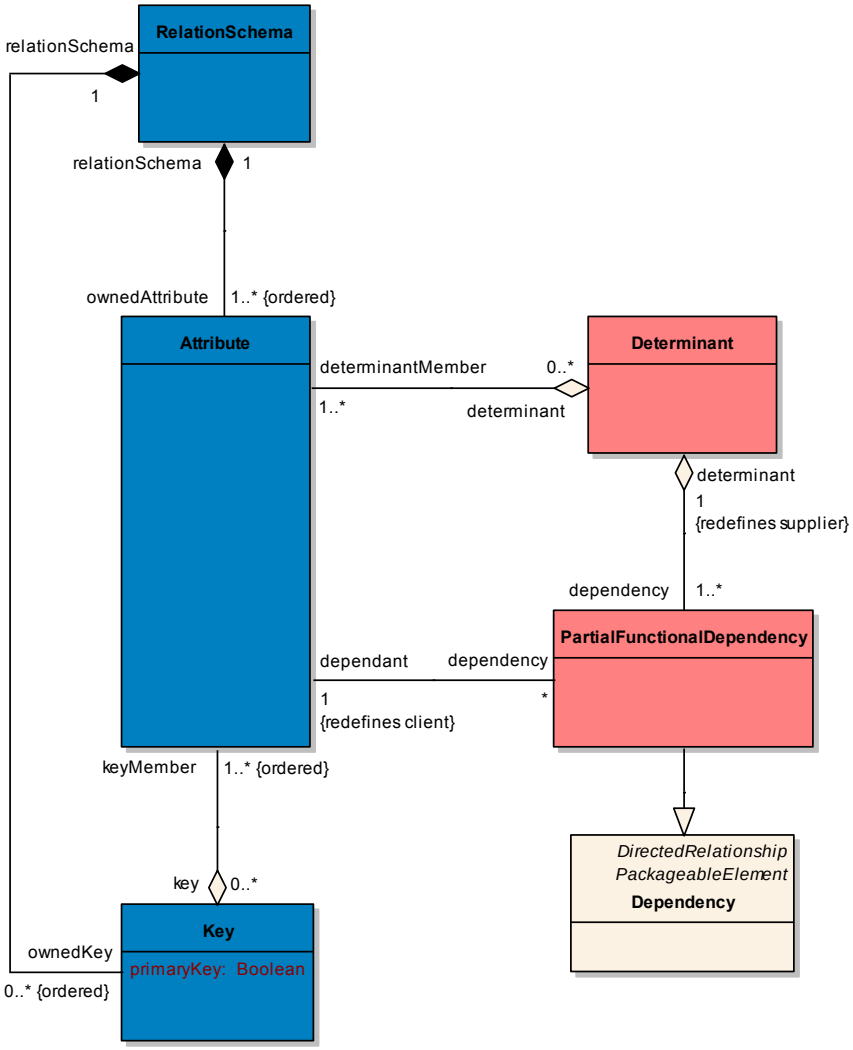


schéma 18: obsah balíčku Functional Dependencies 2

8.3.2 Doplnění funkčních závislostí

Funkční závislosti jsou zjednodušeně vztahy mezi daty sledované na atributech relace. Existuje základní množina funkčních závislostí F, z níž může být pomocí Armstrongových pravidel (axiomy reflexivity, přírůstku a tranzitivity) vytvořena úplná sada funkčních závislostí F^+. Tato sada F^+ je však

pro účely této práce nevhodná a nepotřebná. Cílem je totiž pomocí funkčních závislostí model normalizovat – převést relaci na vyšší normální formu. Při odstranění nežádoucí funkční závislosti automaticky mizí i všechny odvozené závislosti, takže je neúčelné tyto doplňkové závislosti vůbec sledovat.

Vhledem k tomu, že funkční závislosti nejsou dány modelem, ale daty, jsou i data jediným spolehlivým zdrojem informací k jejich získání. Veškeré metody použitelné k jejich získání však vedou ke statistické analýze dat, jsou velmi náročné a díky tomu v praxi téměř nepoužitelné. Aby bylo možné funkční závislosti v rozumném čase identifikovat, musí být potenciální funkční závislosti „vytipovány" tak, aby bylo možné jejich existenci ověřovat na řádově zredukovaném množství dat. K tomu velmi dobře poslouží uživatelská znalost modelované domény. Postup pak vypadá následovně:

1. uživatel označí dvě skupiny atributů, o kterých se na základě svých znalostí domnívá, že tvoří determinant a dependant funkční závislosti

2. analýzou skutečných dat relace je jeho hypotéza potvrzena nebo vyvrácena

8.3.2.1 Návrh potenciálních funkčních závislostí

Díky tomu, že vyslovené hypotézy mohou být v další fázi jednoznačně potvrzeny nebo vyvráceny, je důležité, aby uživatel doporučil k ověření i takové potenciální funkční závislosti, kterými si není zcela jist.

Značky uživatelem doporučené funkční závislosti jsou dvojího druhu – jedna označuje dependanta (dependanty) a druhá atributy tvořící determinant:

Značka (tag)	Hodnota
`FDDependant:Integer`	pořadí funkční závislosti, opět pouze pro účely jejich odlišení během značkování
`FDDeterminant:Integer`	

tabulka 7: značky funkčních závislostí

8.3.2.2 Ověření potenciálních funkčních závislostí

K ověření navržených funkčních závislostí je třeba dotazovat původní datovou základnu:

```
select count (distinct a), count b from
(select a, b from table_name group by a, b)
```

kde a je sada atributů determinantu a b je sada dependantů příslušných determinantu a. Tento dotaz zjistí počet různých hodnot determinantu a a počet různých hodnot depentantu. Aby byla potvrzena hypotéza o funkční závislosti, musí být oba počty shodné.

8.3.3 Normalizace (transformace)

Transformace probíhá tak, že pro každé schéma relace je ověřeno, zda splňuje jednotlivé normální formy. Cílem je, aby nejvyšší normální forma – HNF (highest normal form) všech schémat relací byla BCNF. Proto je postupně každé schéma relace testováno na splnění podmínek 1NF, 2NF, atd. Při zjištění skutečnosti, že schéma relace některé normě odporuje, je provedena transformace. Protože při transformacích mohou vznikat nová schémata

relací, je vždy po provedení nějaké transformace původní a nově vzniklé schéma relace znovu prověřeno.

8.3.3.1 Ověření stupně normality schémat relací

Z diagramu modelu funkčních závislostí, resp. z jeho upravené varianty je patrné, že funkční závislost je navázána na determinant, což je sada atributů daného schématu relace. Normalizace se týká každého jednotlivého schématu relace a funkčních závislostí na něm takto definovaných. Abychom byly schopni pro každé schéma relace zjistit, v jaké normální formě se ještě před vstupem do transformace nachází, jsou definovány metody implementované operacemi typu `isQuery`. Ty pro každou námi kontrolovanou normální formu zjistí, zda ji příslušné schéma relace splňuje. Jsou také definovány pomocné derivované atributy, na které je v těle operací odkazováno. Vše je vidět na následujícím schématu:

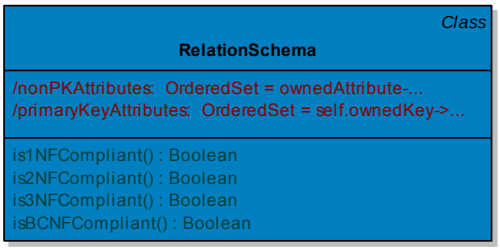


schéma 19: struktura metatřídy RelationSchema

8.3.3.1.1 /primaryKeyAttributes

```
self.ownedKey->select(keys | keys.primaryKey)-
>keyMember.asSet()
```

8.3.3.1.2 /nonPKAttributes

```
ownedAttribute-primaryKeyAttributes
```

8.3.3.1.3 is1NFCompliant

Pokud budeme předpokládat, že z definice RDBMS jsou všechna data atomická, stačí pro kontrolu první normální formy ověřit, existuje-li pro schéma relace klíč, který by byl primární:

```
self.ownedKey->exists(pk | pk.primaryKey)
```

Pro jistotu také ověříme, jestli primárních klíčů není více:

```
self.ownedKey->one(pk | pk.primaryKey)
```

V případě splnění první a nesplnění druhé podmínky je nutné přebytečné primární klíče odstranit, resp. je definovat jako prosté klíče (candidate keys). Vzhledem k tomu, že RDBMS umožňuje existenci pouze jednoho primárního klíče, nebudeme se touto alternativou dále zabývat.

8.3.3.1.4 is2NFCompliant

Aby schéma relace splňovalo druhou normální formu, musí být každý neklíčový atribut závislý na celém primárním klíči schématu relace, nikoli pouze na jeho části. Tato forma se tedy týká pouze relací se složeným primárním klíčem.

Protože Armstrongův axiom reflexivity říká, že podmnožina atributů je vždy závislá na nadmnožině, je tady možné testovat závislost pouze těch atributů, které nejsou součástí primárního klíče.

Jak již bylo uvedeno, klíč je speciálním případem determinantu a tato funkční závislost je implicitní. Omezíme se tedy jen na test absence determinantu, který by obsahoval nekompletní sadu atributů primárního klíče.

Zároveň musí být splněna první normální forma. Pravidlo uspokojení nižší normální formy platí obecně pro všechny další normální formy.

```
is1NFCompliant and

Let incompletePKDeterminants =
self.ownedAttribute.determinant->select(det |

Let missingPKMembers = (primaryKeyAttributes-
det.determinantMember) in

(missingPKMembers <> primaryKeyAttributes) and
missingPKMembers.notEmpty()) in

incompletePKDeterminants.isEmpty()
```

8.3.3.1.5 is3NFCompliant

Tato normální forma rozšiřuje omezení daná druhou normální formou tak, že jsou přípustné závislosti pouze na celém primárním klíči a nikoli neklíčových atributech (nebo jejich sadách). Jakákoli závislost, jejíž determinant by nebyl celý primární klíč, je nepřípustná.

```
is2NFCompliant and

self.ownedAttribute.determinant->forAll(det |
det.determinantMember = self.primaryKeyAttributes)
```

8.3.3.1.6 isBCNFCompliant

Schéma relace je v Boyce-Coddově normální formě, pokud každý determinant je zároveň klíč.

```
is3NFCompliant and

self.ownedAttribute.dependency.determinant->asSet()
->collectNested(det | det.determinantMember) =
self.ownedKey->asSet()->collectNested(key |
key.keyMember)
```

8.3.3.2 Transformace do nejvyšší normální formy

Jak bylo uvedeno, nejvyšší normální formou je pro účely této práce stanovena Boyce-Coddova normální forma. Každé schéma relace proto při nesplnění podmínky kladené na některou z normálních forem projde příslušnou transformací a podle výsledků podmínek vyšších normálních forem pravděpodobně i transformacemi dalšími. Výsledkem všech transformací je splnění podmínky příslušné dané normální formě.

Transformace jsou opět uvedeny v kapitole *Přílohy*.

9 Transformace relačního databázového modelu do objektového

Tato kapitola popisuje získání objektového datového modelu a relačního datového modelu.

Pro každou transformaci je slovně popsán sled dílčích kroků nutných k jejímu provedení.

Veškeré prováděné transformace jsou dále (po krocích) vyjádřeny matematickým zápisem (λ kalkul). Tento zápis popisuje transformaci zdrojových elementů na cílové a přenos jejich vlastností. Doplňkově také popisuje stav cílového modelu po transformaci tam, kde je tento stav z hlediska dalších transformací důležitý.

Pro jednotlivé kroky transformace je také uveden důkaz úplnosti. Úplností se rozumí přenos všech elementů a jejich vlastností ze zdrojového modelu do cílového, tj. neopomenutí transformovat nějaký zdrojový element či jeho vlastnost. Ve výjimečných případech může zdrojem transformace nikoli element, ale speciální konstrukce (sada daných elementů s danými vztahy) ve zdrojovém modelu.

Na závěr je matematický zápis vyjádřen jazykem OCL, díky čemuž je transformace snadno automatizovatelná. Tento zápis zároveň obsahuje vstupní podmínky pro provedení transformace.

9.1 Konstrukce konceptuálního modelu (PIM → CIM)

Metamodel konceptuálního modelu je odvozen z ER diagramu podle Chena a rozšířen o některé prvky:

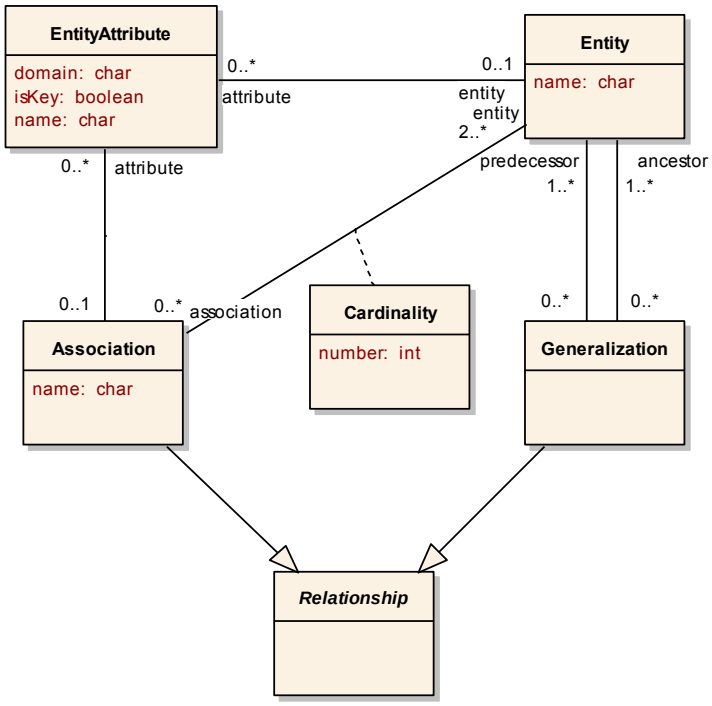


schéma 20: metamodel CIM

Entita (Entity) definuje entitu reálného světa, tedy jakousi skupinu výskytů reálných objektů stejného typu a vlastností. Je jednoznačně určena jménem.

Podobně jako relace vlastní atributy, kterých může být libovolné množství. Toto množství není odspodu omezeno (resp. je omezeno nulou), neboť na konceptuální úrovni mohou být atributy pod požadovanou rozlišovací schopností modelu.

Atribut (Attribute) je jediným elementem vnitřní struktury entity. V kontextu entity je atribut jednoznačně určen jménem. Každému atributu přiřazen datový typ (domain) a vlastnost určující, je-li součástí jednoznačného identifikátoru entity (isKey).

Vazba (Relationship) definuje vztahy mezi entitami a podle typu se dělí na dvě (sama o sobě není v konkrétním modelu použitelná – terminologií UML je abstraktní): asociace, generalizace. **Asociace (Association)** je běžnou vazbou znázorňující vztah různých entit. Pro každou vazbu mezi asociací a entitou je definována **kardinalita (cardinality)**, která určuje, požný počet výskytů entity v asociaci. Pokud není počet entit definován, je tento počet neomezený.

Naproti tomu **generalizace (generalization)** je vztah dvou podobných entit, z nichž jedna je zobecněním druhé, zatímco ta je specializací první. Podobné vazby jsou známy z UML.

Protože konceptuální model je velmi vzdálený počítačovému zpracování a uložení, nepokoušíme se rozšířit metamodel UML o konstrukty v konceptuálním modelu obsažené. Tento model je třeba chápat izolovaně a nehledat souvislosti mezi jeho prvky a prvky UML.

Mezi konceptuálním CIM modelem a relačním PIM modelem[14] jsou na první pohled zřejmé důležité odlišnosti. Konceptuální model se liší v těchto skutečnostech:

- neexistence klíčů – cizí klíč byl nahrazen asociací, informace obsažené v primárním klíči přecházejí na atributy, ostatní klíče zanikají
- neexistuje schéma databáze – zcela chybí konstrukt poskytující jmenný prostor ostatním prvkům modelu

[14] Tímto modelem je rozuměn prostý model bez funkčních závislostí, které byly doplněny a používaly se výhradně pro účely normalizace.

- asociace má vazbu na atribut – asociace může sama o sobě nést informace
- je možná vazba m:n – v relačním modelu díky nutnosti používat k vazbě cizí klíč bylo možné realizovat pouze vazbu 1:m

Z uvedených skutečností vyplývá, transformace jakého druhu musejí být provedeny. Jsou to:

- nahrazení konstruktů ekvivalentními
- nahrazení cizích klíčů asociacemi a nahrazení „vazebních“ tabulek asociacemi s atributy
- doplnění generalizací

Transformace probíhá podle následujícího schématu. Některé dílčí transformace jsou prováděny na základě transformačních pravidel (transformačního modelu), jiné na základě uživatelských značek:

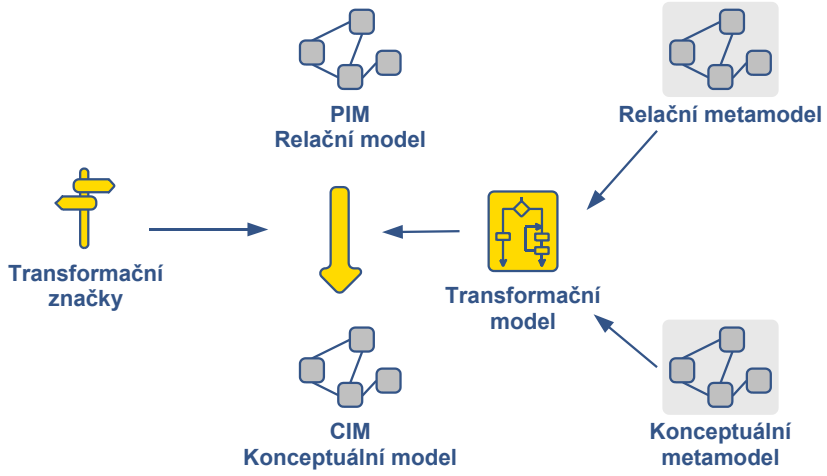


schéma 21: transformace do CIM

9.1.1 Záměna ekvivalentních konstruktů a nahrazení cizích klíčů asociacemi

Jak již bylo uvedeno, konceptuální model nezná celkové schéma, a proto je každé schéma databáze třeba transformovat samostatně, do oddělených cílových modelů. Tyto modely jsou ovšem navzájem mezi sebou na metamodelové úrovni nerozlišitelné. Z elementu DBSchema se nevytváří žádný element konceptuálního modelu. Transformace probíhá podle následujících pravidel:

1. Pro každé schéma relace je vytvořena stejnojmenná entita.

2. Na každé nové entitě existuje stejná sada atributů, jako na původním schématu relace. Tyto atributy mají shodné jméno a typ.

3. Každý atribut, pokud byl v relačním modelu členem primárního klíče, je označen jako identifikátor příslušné entity.

4. Všechny cizí klíče jsou transformovány v binární asociace tak, že na straně cizího klíče není kardinalita definována a na straně primárního klíče je rovna jedné. Jméno asociací je odvozeno z názvů navázaných entit.

5. Atributy všech cizích klíčů jsou odstraněny.

λ-kalkul (legenda je uvedena v kapitole *Přílohy*):

1: $\forall R$:
$R \Rightarrow E$;
$E \triangleleft name := R \triangleleft name$

2: $\forall R\ (R \Rightarrow E)\ \forall A \in R$:
$R \triangleleft A \Rightarrow E \triangleleft AE$;
$E \triangleleft AE \triangleleft name := R \triangleleft A \triangleleft name \wedge$
$(E \triangleleft AE \triangleleft domain := R \triangleleft A \triangleleft type$

3: $\forall PK \in R\ (R \Rightarrow E)\ \forall A \in PK \wedge A \in R \wedge A \Rightarrow AE$:
$AE \in Ident(E)$

4: $\forall FK \in R\ (R \Rightarrow E)$:
$FK \Rightarrow Assoc$;
$Assoc \triangleleft E_i := FK \triangleleft R \wedge$
$Assoc \triangleleft E_j := FK \triangleleft R_o \wedge$
$Assoc \triangleleft name := (\lambda x\ \lambda y \mid x \triangleleft concat: y) \triangleleft: E_i\ E_j \wedge$
$Assoc \triangleleft E_i \triangleleft Card = \varnothing \wedge$
$(Assoc \triangleleft E_j \triangleleft Card := 1$

5: $\forall E\ \ \forall AE\ ((A \in FK) \Rightarrow AE)$:
$|E \triangleleft AE| = \varnothing$

Pro úplnost transformace je třeba ověřit, zda jsou přeneseny všechny elementy zdrojového modelu a zda jsou využity všechny výrazové prostředky cílového metamodelu. Vzhledem k tomu, že transformace v této kapitole zajišťuje vznik entit, atributů, asociací a kardinalit, je důkaz omezen pouze na tyto elementy.

Důkaz úplnosti, krok 1: **Veškerá schémata relace byla transformována do entit**, tj. neexistuje ani jedno schéma relace, pro které by nebyla vytvořena entita. Vlastnostmi schématu relace jsou: jméno, sada vlastněných atributů, sada vlastněných klíčů. **Pro každou vytvořenou entitu bylo přeneseno jméno**, ostatní vlastnosti schématu relace jsou transformovány v dalších krocích. Při splnění uvedeného je tato transformace v tomto kroku úplná.

Důkaz úplnosti, krok 2: **Pro každý atribut každého schématu relace byl** v příslušných (stejnojmenných) entitách **vytvořen příslušný atribut entity**. Vlastnostmi atributů schématu relace jsou jméno, typ a příslušnost do klíče, resp. cizího klíče. **Pro každý vytvořený atribut bylo přeneseno jméno a typ**, ostatní vlastnosti atributů jsou transformovány v dalších krocích. Při splnění uvedeného je tato transformace v tomto kroku úplná.

Důkaz úplnosti, krok 3: Atributy (nejen) každého primárního klíče byly v předchozím kroku transformovány ve stejnojmenné atributy entit. **Veškeré atributy všech entit, které původně tvořily primární klíč v příslušných schématech relace, byly označeny jako identifikátory příslušných entit**. Primární klíč nemá žádné vlastnosti, které by bylo potřeba transformovat. Při splnění uvedeného je tato transformace v tomto kroku úplná.

Důkaz úplnosti, krok 4: **Každý cizí klíč byl transformován v asociaci, ke každé asociaci byly doplněny obě strany kardinalit.** Vlastnostmi cizích klíčů jsou: atributy cizího klíče, atributy protilehlého klíče, přeneseně také vlastní a protilehlé schéma relace. Atributy v cílovém modelu s asociací nesouvisí, ta je vázána na entity. Asociace v cílovém modelu je navázána na entity, jejichž protějšky (schémata relace) ve zdrojovém modelu vlastnily atributy cizího, resp. protilehlého klíče. Při splnění uvedeného je tato transformace v tomto kroku úplná.

Důkaz úplnosti, krok 5: **Všechny atributy, které byly původně členy cizích klíčů, byly odstraněny.** Při splnění uvedeného je tato transformace v tomto kroku úplná. Jsou sice rušeny některé elementy zdrojového modelu, ale informace, kterou tyto elementy nesly (odkaz na „primární“ schéma relace), byla uložena do jiného elementu (asociace).

Celkový důkaz úplnosti: **Všechny elementy, které jsou v této kapitole transformovány, byly přeneseny ze zdrojového do cílového modelu včetně svých vlastností. Nové elementy, které s uvedenou transformací souvisejí, a které nemají obraz v elementu zdrojového modelu, byly ve všech případech vytvořeny. Popisovaná transformace je za splnění těchto podmínek úplná.**

OCL zápis vycházející z metamodelů:

pre1:

```
Entity.allInstances()->isEmpty()
```

pre2:

```
Entity.allInstances()->notEmpty() and
Entity.allInstances().attribute->isEmpty()
```

pre3:

```
Entity.allInstances().attribute->notEmpty() and
Entity.allInstances().attribute->select(a | a.isKey
= true)->isEmpty()
```

pre4:

```
Entity.allInstances().attribute->notEmpty() and
Association.allInstances()->isEmpty()
```

pre5:

```
Association.allInstances()->notEmpty()
```

post:

```
--1
RelationSchema.allInstances()->iterate(rs |
Let newEntity = Entity.oclIsNew() in
newEntity.name = rs.name) and

--2
RelationSchema.allInstances()->iterate(rs |
Let properEntity = Entity.allInstances()->select(pe
| pe.name = rs.name) in
rs.ownedAttribute->iterate(oa |
Let newAttribute = pe.attribute.oclIsNew() in
newAttribute.name = oa.name and
newAttribute.domain = oa.type))

--3
RelationSchema.allInstances()->iterate(rs |
Let properEntity = Entity.allInstances()->select(pe
| pe.name = rs.name) in
rs.ownedKey->select(ok | ok.primaryKey =
true).keyMember->iterate(km |
Let properAttribute = pe.attribute->select(att |
att.name = km.name) in
properAttribute.isKey = true))

--4
ForeignKey.allInstances->iterate(fk |
Let singleEntity = Entity.allInstances()->select(e |
```

```
e.name = fk.referencedKey.relationSchema.name) in
Let multipleEntity = Entity.allInstances()->select(e
| e.name = fk.keyMember->any().relationSchema.name)
in
Let newAssociation = Association.oclIsNew() in
newAssociation.entity-> one(e |e = singleEntity and
Cardinality.number=1) and
newAssociation.entity-> one(e |e = multipleEntity
and Cardinality.number.oclIsUndefined()) and
newAssociation.entity-> size()=2 and
newAssociation.name =
singleEntity.name.concat(multipleEntity.name)) and


--5
RelationSchema.allInstances()->iterate(rs |
rs.ownedAttribute->select(oa |oa.foreignKey-
>notEmpty())->iterate(fka |
Entity.allInstances()->select(e | e.name =
rs.name).attribute->excludes(att | att.name =
fka.name)
```

V tomto okamžiku je v konceptuálním modelu struktura přímo odvozená z relačního. Všechny asociace jsou tedy typu 1:m a chybí generalizace.

9.1.2 Nahrazení schémat „vazebních" relací asociacemi s atributy

V předchozím kroku jsme z entit, které původně obsahovaly atributy cizích klíčů, tyto atributy odebrali a nahradili je asociací. Proto se nyní v konceptuálním modelu mohou objevovat entity, které žádné atributy nemají (byly všechny odstraněny). Některá vazební schémata relace však mohla obsahovat informace s touto vazbou související. Všechna tato to schémata se vyznačují specifickou vlastností: primární klíč byl tvořen výhradně cizími klíči, a pokud v nich byly obsaženy další významové atributy, pak jako neklíčové. Proto je třeba provést další transformace, které:

- entity bez atributů převedou na prosté asociace a
- entity s atributy, z nichž žádný není klíčový, převedou na asociace a atributy původní entity na atributy asociace.

Převodem „vazebních“ entit na asociace tak mj. získáme mezi nově asociovanými entitami vazbu m:n.

Transformační pravidla pak vypadají takto:

1. Každá entita, která nemá žádné atributy (tj. vznikla transformací schématu vazební relace bez dodatečných informací), je transformována na stejnojmennou asociaci. Původní asociace, které vázaly „vazební“ entitu s jinými entitami jsou odstraněny a nově vytvořená asociace je na ně přímo navázána. Nové vazby jsou typu m:n.

2. Každá entita, která nemá žádné identifikátory (tj. vznikla transformací schématu vazební relace s dodatečnými informacemi), je transformována na stejnojmennou asociaci. Této asociaci jsou vytvořeny atributy transformací z atributů zdrojových schémat relací. Původní asociace, které vázaly „vazební“ entitu s jinými entitami jsou odstraněny a nově vytvořená asociace je na ně přímo navázána. Nové vazby jsou typu m:n.

λ-kalkul (legenda je uvedena v kapitole *Přílohy*):

1: $\forall E\ (E \triangleleft AE = \emptyset)$:
$E \Rightarrow Assoc_i$;
$E \triangleleft name := Assoc_i \triangleleft name \wedge$
$Assoc_i \triangleleft Assoc = \emptyset \wedge$
$Assoc_i \triangleleft E := E \triangleleft Assoc \triangleleft E \wedge$
$Assoc_i \triangleleft Card = \emptyset$

2: $\forall E\ (Ident(E) = \emptyset)\ Assoc_i \neq Assoc$:
$E \Rightarrow Assoc_i$;
$E \triangleleft name := Assoc_i \triangleleft name \wedge$
$Assoc_i \triangleleft AE := E \triangleleft AE \wedge$
$Assoc_i \triangleleft Assoc = \emptyset \wedge$

$Assoc_i \lhd E := E \lhd Assoc \lhd E \wedge$
$Assoc_i \lhd Card = \varnothing$

Důsledkem uvedených kroků je neexistence entit bez atributů nebo identifikátorů:

$\forall E_i\,(E_i \lhd AE = \varnothing)\, \forall E_j\,(Ident(E_j) = \varnothing)$:
$E_i = \varnothing \wedge E_j = \varnothing$

Důkaz úplnosti 1: Tato transformace je úplná, pokud (v souladu se zápisem výše) **transformuje všechny entity bez atributů na asociace**. Byly sice odstraněny některé elementy, ale informace, kterou nesly, byla doplněna do nových asociací.

Důkaz úplnosti 2: Tato transformace je úplná, pokud (v souladu se zápisem výše) **transformuje všechny entity bez identifikátorů na asociace** a zároveň **přenáší veškeré atributy původní entity na vytvořenou asociaci**. Byly sice odstraněny některé elementy, ale informace, kterou nesly, byla doplněna do nových asociací.

OCL zápis vycházející z metamodelů:

```
pre1:
Entity.allInstances()->select(e | e.attribute-
>isEmpty())->notEmpty()

pre2:
Entity.allInstances()->select(e | e.attribute-
>select(a | a.isKey)->isEmpty())->notEmpty()

post:
--1
Let emptyEntity = Entity.allInstances()->select(e |
e.attribute->isEmpty()) in
emptyEntity->iterate(ee |
Let newAssociation = Association.oclIsNew() in
newAssociation.name = ee.name and
ee.association.entity->iterate(ae |
Association.allInstances()->select(ass | ass.entity
```

```
= ee + ae)->isEmpty() and newAssociation.entity-
>includes(ae) and
newAssociation.entity->select(e | e =
ae).Cardinality.number.oclIsUndefined())) and


--2
Let unidentifiedEntity = Entity.allInstances()-
>select(e | e.attribute->select(a | a.isKey)-
>isEmpty()) in
unidentifiedEntity->iterate(ue |
Let newAssociation = Association.oclIsNew() in
ue.association.entity->iterate(ae |
Association.allInstances()->select(ass | ass.entity
= ue + ae)->isEmpty() and
newAssociation.entity->includes(ae) and
newAssociation.entity->select(e | e =
ae).Cardinality.number.oclIsUndefined()) and
newAssociation.attribute = ue.attribute;
```

Po provedení uvedené transformace jsou všechny „vazební“ entity převedeny na asociace. Zbývá tedy jen doplnit generalizace tam, kde to odpovídá modelovanému universu.

9.1.3 Doplnění generalizací

Jak je uvedeno v kapitole *Principy databázového návrhu*, mapování generalizací do relačního modelu může být realizováno třemi různými způsoby:

- Struktura generalizace se „zploští“ a všechny „entity“ (nejedná se o entity EER modelu) z různých stupňů generalizace se stanou součástí jediného schématu relace. Hodnoty atributů, které jsou původně definovány jen pro některé entity, nejsou v datech záznamů o jiných entitách zastoupeny (hodnoty jsou nedefinované – NULL). Tento přístup dále označujme jako „izomorfní“.

- Struktura generalizace je zachována tak, že atributy předka na nejvyšší úrovni jsou uloženy do jedné relace, která je pak zakomponována do ostatních mezirelačních vztahů. Dále jsou vytvořena relační schémata,

která mají vazbu na schéma předka a obsahují jen data speciální pro potomka. Pro každý typ potomka je vytvořeno zvláštní schéma relace. Stejným postupem jsou vytvářena schémata relací pro potomky potomků. Na tento přístup budeme dále odkazovat jako na „strukturovaný".

- Struktura generalizace je promítnuta do různých schémat relací, aniž by tato schémata měla mezi sebou formální vztah. Každé schéma relace odpovídá různému typu entity bez ohledu na stupeň generalizace. Schémata relace nejsou vytvářena pro abstraktní předky. Každé schéma relace je pak do celkové struktury databáze zakomponováno izolovaně podle toho, jaké vztahy s ostatními schématy relace má daný konkrétní potomek. Toto řešení je dále nazýváno „izolované".

Podle struktury schémat relací a cizích klíčů lze některé přístupy rozpoznat a vyslovit tak hypotézu o existenci generalizace, ovšem díky tomu, že tyto „vzory" mohou vznikat i k jiným účelům, musí být uživatel vždy tím, kdo navrženou hypotézu potvrdí. Protože v různých případech mohou být hypotézy o generalizaci diametrálně různě úspěšné, je nejjistějším způsobem uživatelská identifikace generalizací. O možnostech nápovědy uživateli (tj. vytváření hypotéz) pojednávají zvláštní kapitoly uvnitř popisu jednotlivých řešení generalizace.

9.1.3.1 Izomorfní generalizace

Tento přístup se vyznačuje tím, že všichni potenciální předci a potomci jsou sdruženi v jedné entitě. Na uživateli tedy je, aby:

- označil atributy k přiřazení konkrétním potomkům,

- takto postupoval až do definice konečných potomků (potomků bez dalších potomků, tzv. listů).

Takovéto třídění atributů probíhá podobným způsobem, jako identifikace implicitních klíčů popsaná výše. Uživateli jsou k dispozici následující značky (tagged values):

Značka (tag)	**Hodnota**
IsoParentName	jméno rodičovské entity – důležité pro situace, kdy je na úrovni bezprostředně vyšší více entit
IsoSiblingName	požadované jméno entity na dané úrovni generalizace

tabulka 8: značky izomorfních generalizací

Těmito značkami se označují jednotlivé atributy rozdělované entity. Po doplnění značek u všech vybraných rozkládaných entit je provedena transformace podle následujících pravidel:

1. Pro každé různé `IsoSiblingName` byla vytvořena stejnojmenná entita.

2. Každá nově vytvořená entita má všechny atributy, které ve značce `IsoSiblingName` nese jméno této entity.

3. Každá nově vytvořená entita má vazbu typu generalizace na entitu, jejíž jméno je uvedeno ve značce `IsoParentName` jejích atributů.

λ-kalkul (legenda je uvedena v kapitole *Přílohy*):

$1: \forall E \quad (E \triangleleft AE \triangleleft taggedValue \triangleleft IsoSiblingName \neq \emptyset)$:
$new(E_j)$;
$|E_j| = |Set(E \triangleleft AE \triangleleft taggedValue \triangleleft IsoSiblingName)| \wedge$
$E_j \triangleleft name := E \triangleleft AE \triangleleft taggedValue \triangleleft IsoSiblingName$

$2: \forall \text{new}(E_j)\ \forall E \neq E_j$:
$(E \triangleleft AE \mathbin{//} (\lambda x \mid E \triangleleft AE \triangleleft IsoSiblingName) \triangleleft: E_j \triangleleft name) \Rightarrow E_j \triangleleft AE$

$3: \forall \text{new}(E_j)\ \forall E \neq E_j$:
$new(Gen)$;
$|Gen| = |E_j| \wedge$
$(E_j \succ E \mathbin{//} (\lambda x \mid E \triangleleft name = x) \triangleleft: E_j \triangleleft AE \triangleleft taggedValue \triangleleft IsoParentName)$

Důsledkem uvedených kroků je skutečnost, že žádná dvojice předek-potomek nemá společné atributy:

$\forall E_j\ \forall E\ (E_j \succ E)$:
$E_j \triangleleft AE \cap E \triangleleft AE = \emptyset$

Důkaz úplnosti: Tato transformace je úplná, pokud (v souladu se zápisem výše) transformuje všechny označené elementy (zajištění úplnosti označení je ponecháno na uživateli). **Pro každou zvláštní značku potomka je potomek vytvořen včetně přenosu všech atributů. Pro všechny potomky je vytvořena vazba generalizace na příslušného předka.**

OCL zápis vycházející z metamodelů:

```
pre:
--1
Generalization.allInstances()->isEmpty() and

--2
EntityAttribute.allInstances()->select(att |
att.taggedValue.name = IsoParentName)->notEmpty()

post:
Let newEntities = Entity.allInstances() -
Entity.allInstances()@pre in
```

```
--1
newEntities.name = Entity.allInstances().attribute-
>collect(att | att.taggedValue.IsoSiblingName)-
>asSet() and


--2
newEntities->iterate(ne | ne.attribute =
Attribute.allInstances()->select(att | att.
taggedValue.IsoSiblingName = ne.name)) and


--3
newEntities->iterate(ne | ne.predecessor.name =
ne.attribute->any().taggedValue.IsoParentName) and


newEntities->iterate(ne |
ne.predecessors().attribute-
>excludesAll(ne.attribute))
```

V kódu je používána metoda predecessors() pro zjištění sady předků dané entity. Její OCL vyjádření je následující:

```
context Entity
def: predecessors(): Set(Entity) =
if self.predecessor->notEmpty
then resultSet = self.predecessor;
resultSet->including(predecessor.predecessors())
else resultSet
endif
```

9.1.3.1.1 Návrh hypotézy generalizace

V tomto případě není možné kompetentně posoudit, která z entit by mohla generalizaci skrývat. Vodítkem by mohla být poměrně široká sada atributů entity. Z analýzy dat příslušné relace, v které by se sledovaly sady atributů, jejichž hodnota je pro značnou část záznamů nedefinovaná, by bylo možné hypotézu navrhnout. Ovšem vzhledem k vágně definovaným pojmům „poměrně široká sada atributů“ a „značná část záznamů“ a také s přihlédnutím k faktu, že ani ideálně rozložená data relace nemusí znamenat

skrytou generalizaci, je v tomto případě navržení i schválení hypotéz ponecháno výhradně na uživateli.

9.1.3.2 Strukturovaná generalizace

V tomto případě již rozdělování entit není potřeba – každý předek a potomek je reprezentován zvláštní entitou. Tyto entity je třeba pouze rozpoznat a určit jejich umístění v generalizační hierarchii. Uživatel tedy označuje celé entity. Označovat však může je ty entity, mezi kterými je definovaná asociace:

Značka (tag)	**Hodnota**
GenParentName	jméno rodičovské entity

tabulka 9: značky strukturovaných generalizací

Pomocí následující transformace bude vazba asociace nahrazována vazbou generalizace:

1. Každá entita se značkou `GenParentName` má vazbu typu generalizace na entitu, jejíž jméno je uvedeno ve stejné značce.

2. Binární asociace mezi entitami je odstraněna.

λ-kalkul (legenda je uvedena v kapitole *Přílohy*):

1: $\forall E_i\,(E_i \triangleleft taggedValue \triangleleft GenParentName \neq \varnothing)$:
$new(Gen)$;
$|Gen| = |E_i| \wedge$
$(E_i \succ E\,//\,(\lambda x \mid E \triangleleft name = x) \triangleleft: E_i \triangleleft taggedValue \triangleleft GenParentName)$

2: $\forall Assoc\ \forall E_i\ \forall E_j\,(E_i \succ E_j)$:
$(Assoc\,//\,(\lambda x \mid Assoc \triangleleft E = x) \triangleleft: Set\{E_i; E_j\}) = \varnothing$

Důkaz úplnosti: Tato transformace je úplná, pokud (v souladu se zápisem výše) transformuje všechny označené elementy (zajištění úplnosti označení je ponecháno na uživateli). **Pro každou zvláštní značku předka je vytvořena vazba generalizace na příslušného předka a původní asociace je odstraněna.**

OCL zápis vycházející z metamodelů:

```
pre:
--1
EntityAttribute.allInstances()->select(att |
att.taggedValue.name = GenParentName)->notEmpty()


post:
Entity.allInstances() = Entity.allInstances()@pre
and
Attribute.allInstances() =
Attribute.allInstances()@pre and


--1, 2
Let ancestors = Entity.allInstances()->select(e |
e.taggedValue->includes(GenParentName)) in
ancestors->iterate(a | a.predecessor.name = a.
taggedValue.GenParentName and
a.association->select(ass | ass.entity = {a;
a.predecessor}->isEmpty())
```

9.1.3.2.1 Návrh hypotézy generalizace

Pro strukturovaný způsob promítnutí generalizace do relačního schématu je příznačné, že přestože vazby mezi schémata relace jsou realizované cizími klíči a jsou tedy obecně typu 1:m, je faktická vazba typu 1:1 – cizí klíč potomka odkazující na předka je zároveň primárním klíčem potomka. Jsou tedy hledána taková schémata relací, u kterých má potenciální potomek cizí klíč odkazující na potenciálního předka; tento cizí klíč je zároveň jeho vlastním primárním klíčem. Vzhledem k odstranění cizích klíčů v entitách je nutné hledat v relačním modelu.

9.1.3.3 Izolovaná generalizace

Stejně jako u strukturované generalizace je i zde každý prvek generalizace reprezentován samostatnou entitou. Přímá vazba mezi entitami je možná, ale není vyžadována. Nevyjadřuje totiž vztah nahrazující generalizaci, ale možné jiné asociace:

Značka (tag)	**Hodnota**
IndParentName	jméno rodičovské entity

tabulka 10: značky izolovaných generalizací

Pomocí následující transformace bude mezi entitami vytvářena nová vazba generalizace:

1. Každá entita se značkou `IndParentName` má vazbu typu generalizace na entitu, jejíž jméno je uvedeno ve stejné značce.

λ-kalkul (legenda je uvedena v kapitole *Přílohy*):

$$
\begin{aligned}
&1:\ \forall E_i\,(E_i \triangleleft taggedValue \triangleleft IndParentName \neq \varnothing):\\
&new(Gen);\\
&|Gen| = |E_i| \wedge\\
&(E_i \succ E\,//\,(\lambda x \mid E \triangleleft name = x) \triangleleft: E_i \triangleleft taggedValue \triangleleft GenParentName)
\end{aligned}
$$

Důkaz úplnosti: Tato transformace je úplná, pokud (v souladu se zápisem výše) transformuje všechny označené elementy (zajištění úplnosti označení je ponecháno na uživateli). **Pro každou zvláštní značku předka je vytvořena vazba generalizace na příslušného předka.**

OCL:

```
pre:
--1
EntityAttribute.allInstances()->select(att |
att.taggedValue.name = IndParentName)->notEmpty()


post:
--1
Entity.allInstances() = Entity.allInstances()@pre
and
Attribute.allInstances() =
Attribute.allInstances()@pre and


Let ancestors = Entity.allInstances()->select(e |
e.taggedValue->includes(GenParentName)) in
ancestors->iterate(a | a.predecessor.name = a.
taggedValue.IndParentName) and


ancestors->iterate(a | a.predecessors().attribute-
>excludesAll(a.attribute))
```

Pozn.: V kódu je použita metoda `predecessors()`, která je definována v kapitole *Izomorfní generalizace*.

9.1.3.3.1 Návrh hypotézy generalizace

Protože izolované entity mezi sebou nemají vazby, podle kterých by bylo možné generalizaci odvodit, je pro návrh hypotéz použit jiný princip. Izolované entity totiž obsahují atributy, jejichž jistá část je pro hierarchický strom generalizace shodná. Tato část narůstá směrem ke konečným, listovým potomkům. Primární klíč je vždy shodný. Hledáme tedy takové entity, které mají shodné klíčové prvky a sady opakujících se atributů. Podle relativního množství atributů v opakující se skupině usuzujeme na úroveň entity v generalizaci.

9.2 Konstrukce objektového modelu (CIM → PIM)

9.2.1 Cílový PIM model

Uložení dat v objektové databázi založené na třídách[15] tak, jak je popsáno v kapitole *Objektové databázové systémy*, kopíruje reprezentaci dat v aplikační vrstvě. Uživateli databáze zůstává víceméně skryto, jestli je objekt transientní (dočasný, uložen pouze v paměti), nebo persistentní (uložen v databázi). Pokud požaduje uložení objektu, nemusí narozdíl od relačních databází složitě mapovat mezi aplikačním obrazem objektu a jeho databázovým obrazem. Pomocí rozšíření, které je pro většinu objektových databází pro nejpoužívanější programovací jazyky k dispozici, pouze systému sdělí, zda chce objekt uložit, či nikoli. Stejně snadné je i vyvolání objektu z databáze do paměti.

Cílovým PIM objektovým modelem je tedy stejný model, jaký je používán pro modelování struktury aplikací, pokud jsou implementovány v objektovém jazyku. Jako cílový model budeme tedy používat Class pohled na model UML. Tento model není třeba ani rozšiřovat o žádné elementy, které by základnímu UML modelu byly cizí.

Mapování uvedené v kapitole *Principy databázového návrhu* na první pohled naznačuje, že dochází k určitým změnám v uložení dat, nicméně toto uložení se netýká jen objektových databází, ale objektového přístupu obecně. Ani

[15] V současnosti mezi takové komerční produkty patří např. Cache, O2, Jasmine apod. Kromě těchto existují ještě ODB s jiným datovým modelem (a různou „čistotou" objektového přístupu): mírně upravená Gemstone využívající navíc kolekce objektů (collection-based), objektově-relační databáze Oracle 11g, či různé DB s XML modely používané pro popis hierarchických struktur.

objekty v paměti aplikace nejsou fyzicky uloženy tak, jak je uvedene v třídním pohledu na UML model, např. asociace a agregace je řešena odkazem, kdy místo atributu je odkaz na OID cílového objektu (případně OID objektu dále odkazujícího na sadu objektů), generalizace není do uložení dat promítnuta vůbec (popisuje vztah mezi třídami, nikoli mezi uloženými objekty). Narozdíl od relačních databází je toto uložení zcela transparentní a uživateli zůstává skryto.

Cílový model je tedy zjednodušenou verzí UML specifikace, která je vidět na následujícím schématu:

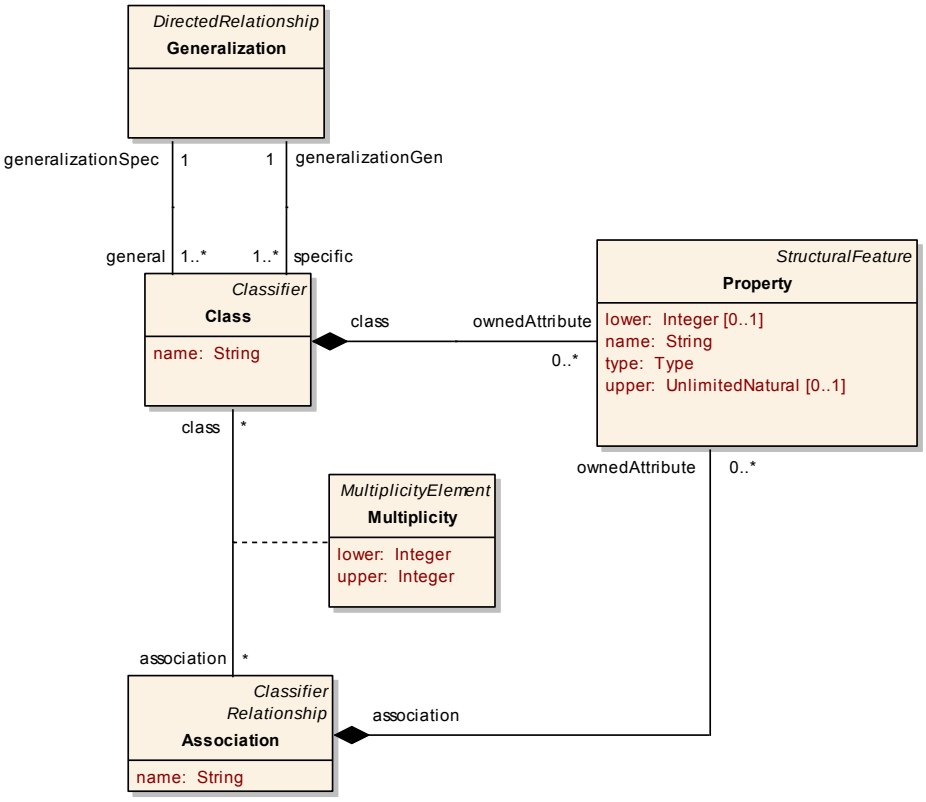


schéma 22: objektový metamodel

9.2.2 Transformace z EER modelu

Proces transformace z ERR modelu do objektového modelu je jednoznačný a plně automatizovatelný – k přenesení všech informací není třeba uživatelských zásahů. Teprve po vytvoření objektového modelu může uživatel doplnit některé informace, pro jejichž zachycení je EER model nedostatečný. Těmi jsou zejména spodní meze multiplicit (viz níže).

Transformace probíhá podle následujícího schématu. Vlastní transformace je provedena výhradně podle transformačního modelu. Uživatel pak výsledný model může upravit a doplnit o konstrukty vlastní pouze cílovému modelu:

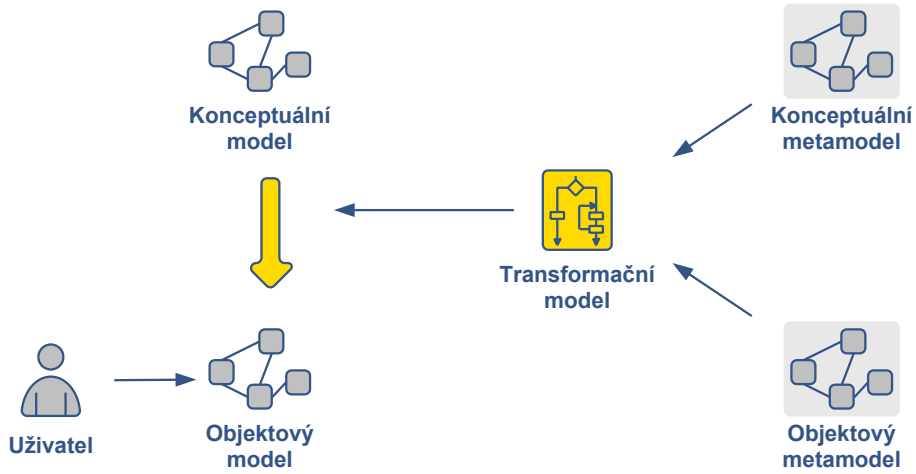


schéma 23: transformace do PIM

Transformační pravidla jsou následující:

1. Každá entita byla transformována do třídy, včetně všech svých atributů.

2. Veškeré asociace byly přeneseny včetně všech atributů a kardinalit. Související kardinality jsou přeneseny tak, že každá kardinalita původní asociace tvoří horní mez příslušné multiplicity.

3. Veškeré generalizace byly přeneseny.

λ-kalkul (legenda je uvedena v kapitole *Přílohy*):

1: $\forall E$:
$E \Rightarrow C$;
$C \triangleleft A := E \triangleleft AE$

2: $\forall Assoc$:
$Assoc \Rightarrow AssocC;$
$AssocC \triangleleft A := Assoc \triangleleft AE \wedge$
$AssocC \triangleleft Mult \triangleleft upper := Assoc \triangleleft Card$

3: $\forall$ *Gen*:
$Gen \Rightarrow GenC$;
$GenC \triangleleft general := Gen \triangleleft ancestor \wedge$
$GenC \triangleleft specific := Gen \triangleleft predecessor$

Důkaz úplnosti: Tato transformace je úplná, pokud (v souladu se zápisem výše) transformuje všechny elementy zdrojového modelu. **Všechny entity, asociace a generalizace jsou včetně všech svých vlastností transformovány.**

OCL:

```
post:
--1
Entity.allInstances()->iterate(e |
Let newClass = Class.oclIsNew() in
e.name = newClass.name and
e.attribute->iterate(a |
Let newAtt = Property.oclIsNew() in
newAtt.class = newClass and
newAtt.name = a.name and
newAtt.type = a.domain)) and
```

```
--2
Association.allInstances()->iterate(ass |
Let newAss = Association.oclIsNew() in
ass.name = newAss.name and
ass.attribute->iterate(a |
Let newAtt = Property.oclIsNew() in
newAtt.association = newAss and
newAtt.name = a.name and
newAtt.type = a.domain) and
ass.entity->iterate(e |
e.name = newAss.class.name and
e.Cardinality.number =
newAss.class.Multiplicity.upper)) and

--3
Generalization.allInstances()->iterate(gen |
Let newGen = Generalization.oclIsNew() in
gen.predecessor.name = newGen.general.name and
gen.ancestor.name = newGen.specific.name)
```

9.2.2.1 Doplnění multiplicit

Vhledem k tomu, že v konceptuálním modelu je kardinalita definována pouze jako 1 nebo n, je i do objektového modelu takto přenesena. Objektový model ale podporuje multiplicitu libovolným určením její dolní a horní meze. Proto je vhodné objektový model dále zpřesnit. Toto zpřesnění však pro svoji jednoduchost probíhá přímou úpravou modelu, tj. nejsou vytvářeny uživatelské značky, které by byly dále transformovány.

9.3 Konstrukce objektového datového modelu z relačního (PIM → PIM)

9.3.1 Kritika EER modelu

Z nahrazení jednoduchého ER modelu rozšířeným je patrný vliv objektových technologií i na konceptuální úrovni (je to také důkaz toho, že objektové pojetí je reálnému světu blíž). Přesto však ani rozšířený ER model nemá prostředky k tomu, aby zachytil některé detaily, které je třeba specifikovat na

nižší úrovni (směrem k implementaci). Paradoxně se tak setkáváme s tím, že i směrem od relačního modelu ke konceptuálnímu dochází ke ztrátě informace.

Další skutečností je, že se v současnosti díky masovému a téměř monopolnímu rozšíření objektového přístupu a technologií již od počátku implementace informačního systému počítá s cílovou – objektovou platformou. V praxi lze velmi zřídka narazit na správce požadavků, který by při modelování používal ER diagramy. Mnohem častěji jsme svědky využívání UML v nejranějších fázích projektů, vytváření takzvaných analytických modelů. Vyjadřovací schopnost těchto modelů nezávisí na dostupných prostředcích UML, které jsou nepřeberné, ale pouze na uvážení jejich autora. Tyto modely jsou dále zpřesňovány a může z nich být nakonec až vygenerován programový kód.

Z těchto důvodů je vhodné položit si otázku, splňuje-li ještě ER model to, k čemu byl původně určen. Odpověď je asi individuální, záleží na volbě každého, kdo konceptuální modely vytváří. Nakolik je objektový model považován za výpočetně nezávislý a nakolik „jen" za platformně nezávislý, je ponecháno na zvážení čtenáři.

9.3.1.1 Přímá transformace

Pokud bychom se na základě informací uvedených v předchozí kapitole rozhodli transformovat relační model do objektového přímo, tj. bez vytváření konceptuálního modelu, je třeba příslušným způsobem upravit transformace, kterými vytváříme konceptuální model. Tyto transformace jsou trojího druhu:

- záměna ekvivalentních konstruktů,
- nahrazení schémat vazebních relací a

- doplnění generalizací.

Následující tabulka uvádí nutné změny v transformacích, které jsou popsány v kapitole *Konstrukce konceptuálního modelu (PIM → CIM)*:

Element relačního metamodelu	**Element konceptuálního metamodelu**	**Element objektového metamodelu**	**Změny v transformaci**
RelationSchema	Entity	Class	záměna elementů
Attribute	EntityAttribute	Property	záměna elementů
ForeignKey	Association	Association	záměna elementů, přímé naplnění multiplicit
-	Generalization	Generalization	záměna elementů
Key	-	-	záměna elementů
Key {primaryKey = true}	EntityAttribute {isKey = true}	-	klíče nejsou transformovány
RelationSchema {vazební, viz transformace do CIM}	Association	Association	záměna elementů

tabulka 11: značky izolovaných generalizací

10 Závěr, zhodnocení, další práce

Tato práce popisuje metodu, jak relační databázový model transformovat do objektového. Výhodou je fakt, že relační model může být nekvalitní, tj. může obsahovat nerelevantní nebo implicitní struktury a nemusí splňovat základní normální formy. Další výhoda spočívá v dosažení relativně vysoké míry automatizace, kde interakce uživatele je co možná nejvíce omezena a mnoho potřebných informací, které relační datový model explicitně nenese, je doplněno z jiných zdrojů. Takto popsaná transformace pokládá solidní základ navazující migraci dat, která může využít (v CASE nástroji) evidovaných tras přechodu původního atributu do cílového.

Řešení je výhodné také z důvodu vytvoření konceptuálního modelu, který může být v konkrétních situacích cenný. V opačném případě, kdy by vytvoření tohoto modelu bylo zbytečné a přínos uživateli by spočíval výhradně ve vytvoření objektového modelu, je popsána změna pravidel pro přímou transformaci.

Metoda transformace je založena na standardních platformách (MDA, UML, OCL), které jsou v současnosti dostatečně podpořeny komerčními produkty.

V neposlední řadě bylo dokázáno, že popsaná transformace je úplná, pokud jsou postupně transformovány všechny elementy zdrojového modelu a přitom nedochází ke ztrátě informace. Toto tvrzení vychází z indukce, tj. pokud je každá dílčí transformace úplná (což bylo v předchozích kapitolách dokázáno), potom i jejich libovolná kombinace je úplná.

Autor plánuje na předloženou práci navázat mj.:

- vytvořením nástroje pro překlad výrazů jazyka OCL to transformačního jazyka nástroje Enterprise Architect, aby bylo možné transformace modelů vyjadřovat výhradně v OCL; překlad do nativního jazyka CASE nástroje pak proběhně automaticky,

- vytvořením nástroje pro zadávání uživatelských značek – tento nástroj by měl zajistit konzistenci značek a jejich hodnot, což v současnosti musí zajistit uživatelů.

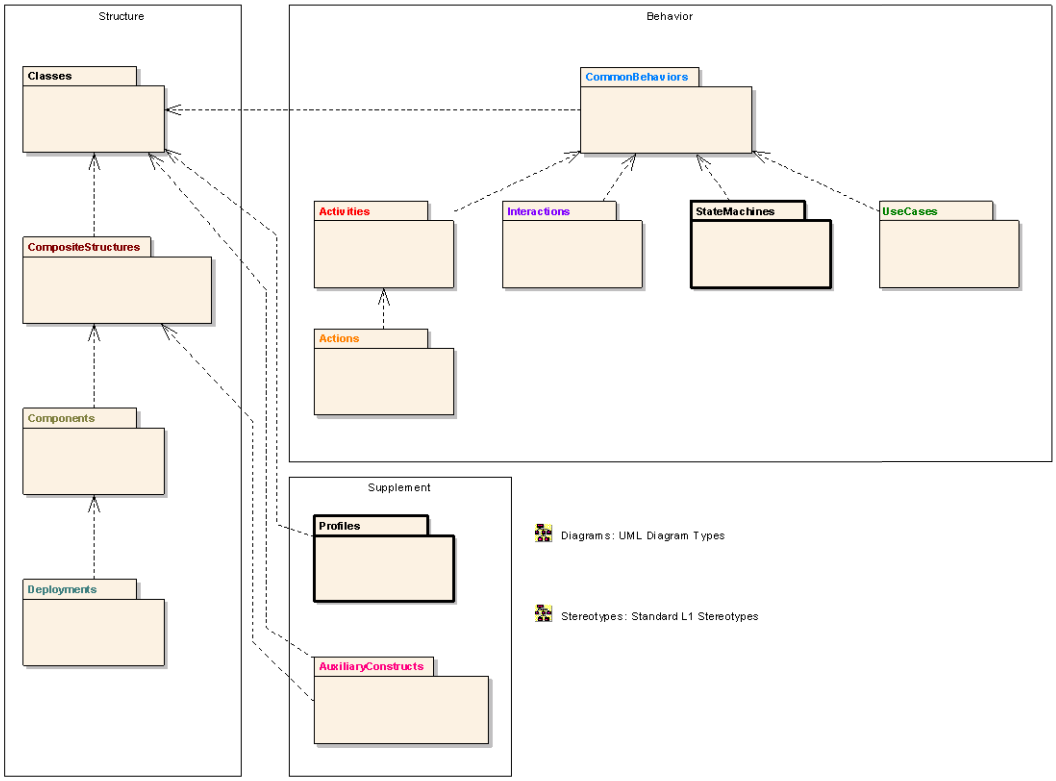
11 Přílohy

11.1 Jazyk UML

Níže uvedené diagramy pochází z vizuálního nástroje pro výuku jazyka UML, který byl vytvořen z [OMG_05] autorem této práce. Jedná se o UML metamodel, jehož interaktivitu zajišťuje prostředí CASE nástroje EA (Enterprise Architect).

První diagram mj. znázorňuje rozdělení UML jazyka do oblastí (Structure, Behavior a Supplement) a vztahy mezi příslušnými balíčky. Veškeré vazby jsou typu `PackageMerge`:

UML Superstructure



Document Format & Structure

This document contains structure diagrams taken from the UML 2.0 Superstructure, document version 05-07-04. Diagrams are extended with explanatory notes and some examples. Also, hierarchy overview (generalization) diagrams are supplied.

Each root UML package is distinguished by a unique font color, which is maintained through all sub-packages and contained elements. In addition, each class diagram is given another color, which is then used as a background color for all elements that are new to given diagram (i.e. all elements to which the diagram is a parent diagram). This makes reading diagrams easier

Packages with bold borders are not yet finished.

How To Read Diagrams

All diagrams here are structure class diagrams with only a small variety of contained elements. Noteworthy constructs used are:

Classes

Classes' attributes are only visible in their parent diagrams. Operations are defined but are not visible at all. Attribute and operation visibility can be individually toggled by pressing Ctrl-Shift-Y when an element is highlited.

Relationships

Relationships between classes are only visible in classes' parent diagrams. Some of them are derived unions, usually when related elements are abstract. Derived union means that the relationship represents an union of all matching underlying relationships (relationships between abstract elements' subclasses in other diagrams, which are marked as subsets of a given relationship). Subsetting relationships may further be subset.

In hierarchy overview diagrams only generalizations are visible. Also, being a semantic equivalent to generalization, element merging (via package merge) is represented by a general dependency relationship in hierarchy overview diagrams.

Superstructure 2.0 Versions

Since UML Superstructure 2.0 version 03-08-02 (which the exams are based on), many errors have been corrected, but also the structure has been slightly changed. Diagrams that are affected in a significant manner are accordingly marked. You are expected to compare these with those in version 03-08-02 to see the difference. However, in almost all cases (except for Dependencies package) the most recent version contains all information necessary to pass the test.

schéma 24: struktura UML metamodelu

Např. balíček `Classes` je dále složen z vlastněných balíčků `Kernel`, `Dependencies`, atd.:

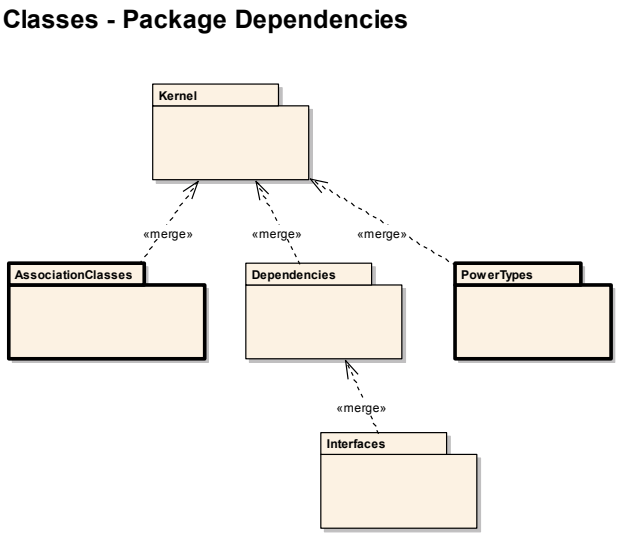


schéma 25: struktura základních balíčků

Balíčky na nejnižší úrovni granularity již obsahují přímo UML elementy. Pro přehlednost je ale jako další mezivrstva definována sada pohledů (diagramů) na daný balíček. Tato vrstva však už není pro vlastní model podstatná. Např. balíček `Kernel` obsahuje tyto pohledy:

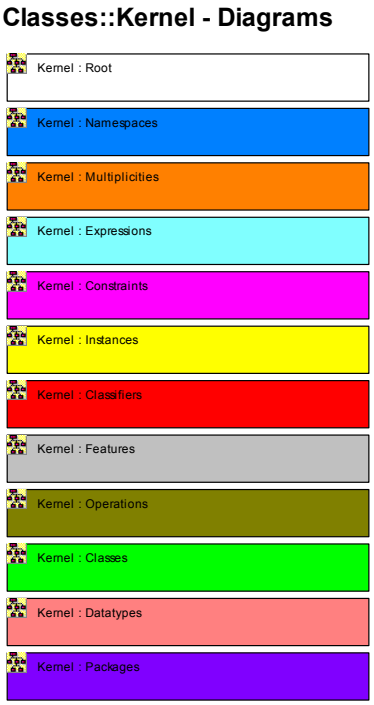


schéma 26: diagramy balíčku

Každý diagram pak popisuje jemu příslušnou oblast, zde např. diagram `Operations`. Na třídě `Operation` je vidět, že má vlastnost `isQuery`, která říká, jestli je operace invazivní (např. veškeré operace modelu OCL invazivní nejsou):

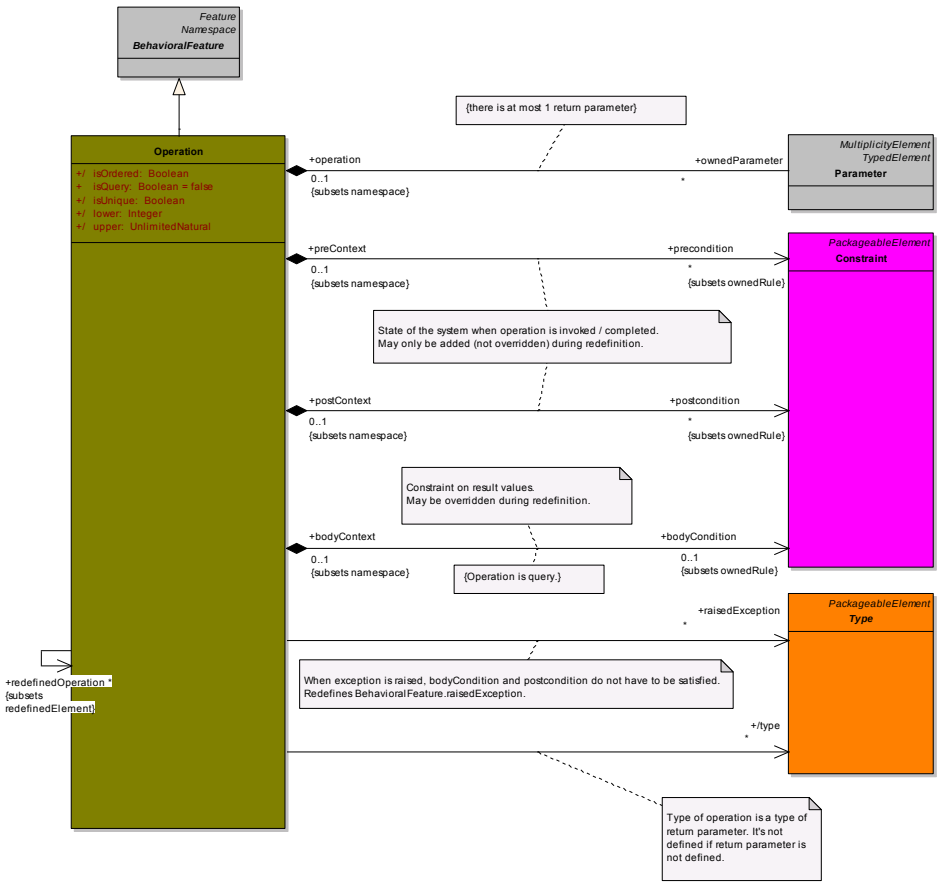


schéma 27: konkrétní diagram

11.2 Jazyk OCL

I zde uvedené diagramy pocházejí z nástroje, který autor vytvořil pro výuku UML a OCL. Předlohou pro vytvoření bylo [OMG_05-2] a [WARM03]. První diagram ukazuje možnosti použití OCL v rámci UML. Tam, kde UML definuje element `ValueSpecification`, lze použít výrazu jazyka OCL ([OMG_03] toto dokonce explicitně doporučuje):

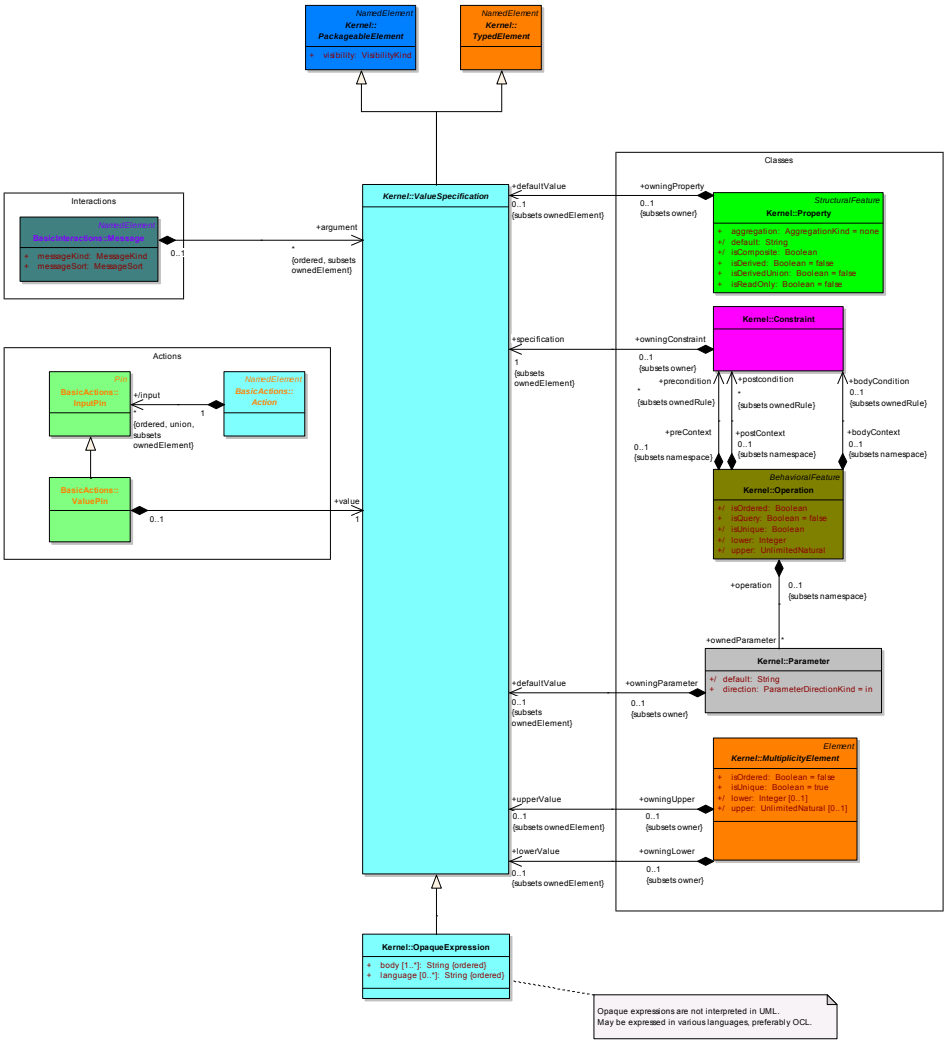


schéma 28: použití OCL v UML

Ze schématu je patrné, že použití je různorodé a zahrnuje např. definici implicitních hodnot atributů, parametrů zpráv, ale zejména omezujících podmínek. Jediným způsobem, jak popsat operaci v OCL, je tedy deklarace vstupních a výstupních podmínek operace, neboť pro přímý popis těla metody (implicitně invazivní) by bylo zapotřebí použít jazyka imperativního.

Na dalším schématu jsou vidět elementy jazyka OCL a operace, které je možné ve výrazech na daný element použít:

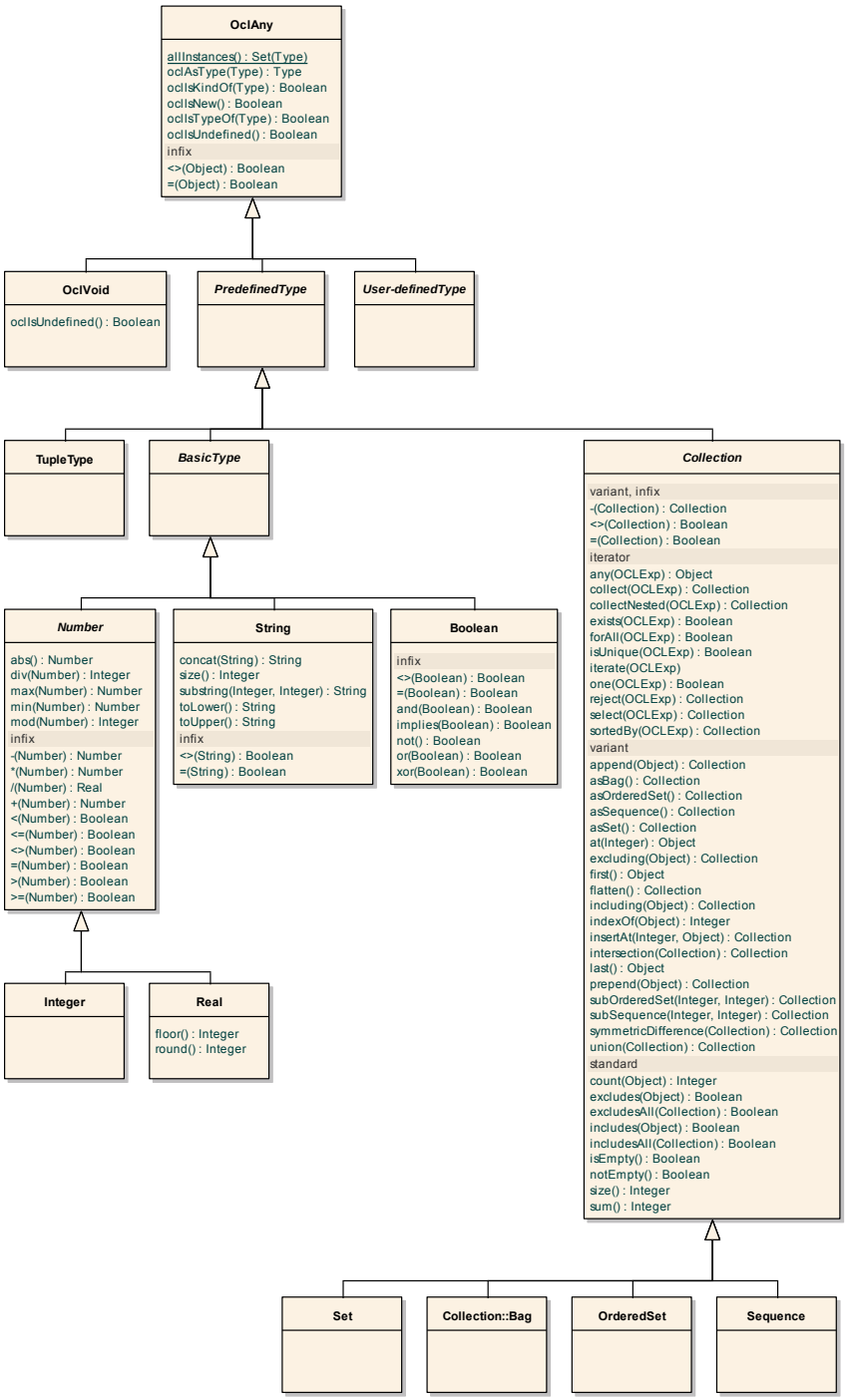


schéma 29: operace na elementech OCL

11.3 Transformace v CASE nástroji

Jako modelovací a transformační nástroj autor používá Enterprise Architect společnosti Sparx Systems. Následující ukázky jsou ale realizovatelné obecně v jakémkoli CASE nástroji, který podporuje MDA a UML 2.0.

Následující obrázek ukazuje možnosti vytvoření vlastních transformačních šablon (součástí produktu jsou předem vytvořené šablony pro nejčastější použití – šablony potřebné pro tuto práci je třeba vytvořit ručně). Jazyk je jednoduchý, ale bohužel proprietární (nejde o nějaký ze standardu, např. QVT – Query View Tranform):

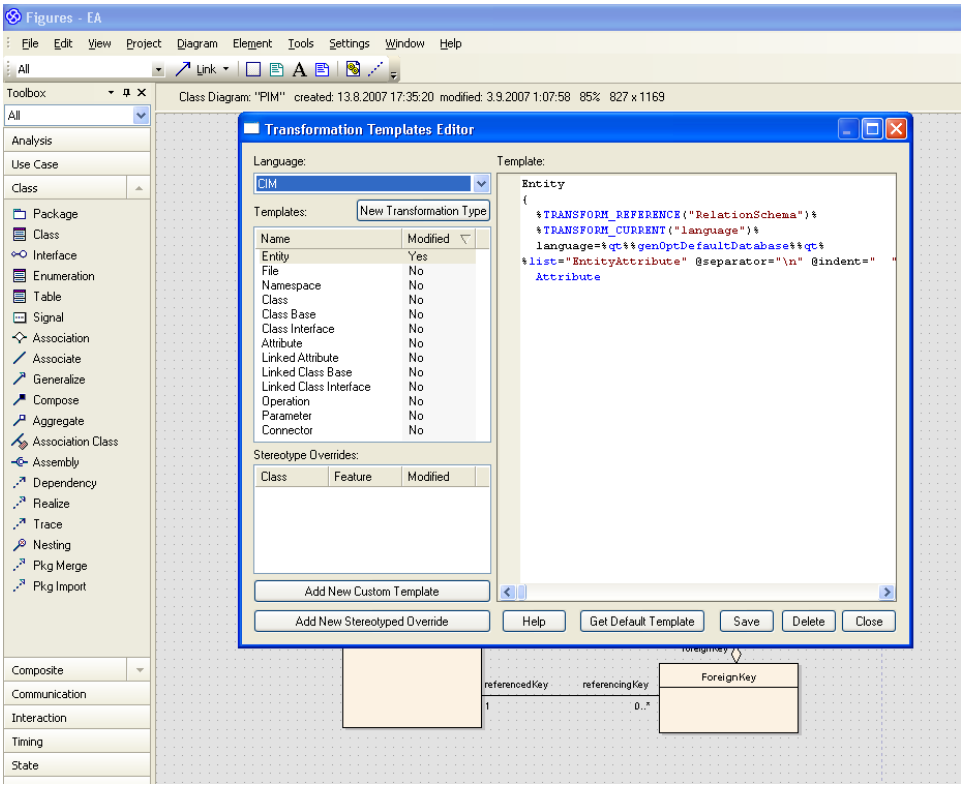


schéma 30: transformace v CASE nástroji

11.4 Překlad z jazyka OCL

V současnosti existují nástroje, které překládají specifikaci metod vyjádřenou v OCL do implementačního jazyka. Např. OCLTF (OCL Translation Framework) podporuje překlad OCL kódu metod při dopředné transformaci do programovacího jazyka modelu PSM. Např. překlad metody přidávající zákazníka do poštovního seznamu firmy je následovný:

OCL:

```
Context MailList::enroll(c: Customer)
pre : not participants->includes(c)
post: participants = participatns@pre->including(c)
```

Java:

```
Void enroll(Customer c) {
  assert( ! participants.contains(c) );
  old_participants = new ArrayList(participants);
  // ...
  // body of operation
  // ...
  assert( participants = old_participants.add(c) );
}
```

11.5 Transformace PSM → PIM

Založení schématu databáze je popsáno následující podmínkou. Je předpokládáno založení pouze jednoho schématu databáze (další schémata databáze by byla zakládána v cyklu). Vnější proměnné, vstupující z PSM modelu vyjádřeného kódem, předchází otazník:

```
pre:
„create schema schema_name"
```

```
post:
Let newSchema:DBSchema = DBSchema.oclIsNew() in
newSchema.name = ?schema_name
```

Založení schématu relace a jeho vnitřní struktury (tj. atributů) je popsáno další podmínkou. Schéma relace je vytvořeno v právě založeném schématu databáze. Ve výstupních podmínkách jsou používané proměnné definované v předchozích podmínkách:

```
context: newSchema

pre:
„create table table_name (column_name column_type,
...)"

post:
ownedRelationSchema->size() =
ownedRelationSchema@pre->size()+1

Let newRS:RelationSchema =
ownedRelationSchema.oclIsNew() in
Let i=?(column_name column_type, ...)->all() in
i->iterate(i | Let newAtt:Attribute =
newRS.ownedAttribute oclIsNew() in
newAtt.name = i.?column_name and
newAtt.type = i.?column_type)
```

Dále jsou vytvořeny unikátní sady atributů, tedy klíčů, pro právě založenou relaci. Vyjmenované atributy jsou obsazeny do nově vytvořeného klíče. Příznak `primaryKey` není nastaven. Operátor `@pre` označuje stav objektu před transformací:

```
context: newRS

pre:
„alter table table_name add unique (column_name,
...)"

post:
ownedKey->size() = ownedKey@pre->size()+1
```

```
Let newKey:Key = ownedKey.oclIsNew() in
not newKey.primaryKey and
Let i=?(column_name, ...)->all() in
i->iterate(i | newKey.keyMember =
newKey@pre.keyMember->append( ownedAttribute-
>select(att | att.name = i.?column_name)))
```

Je vytvořen primární klíč pro právě založenou relaci tak, že jsou do něj obsazeny uvedené atributy schématu relace a klíči je nastaven příznak `primaryKey`:

```
context: newRS

pre:
„alter table table_name add primary key
(column_name, ...)"

post:
ownedKey->size() = ownedKey@pre->size()+1

Let newKey:Key = ownedKey.oclIsNew() in
newKey.primaryKey and
Let i=?(column_name, ...)->all() in
i->iterate(i | newKey.keyMember =
newKey@pre.keyMember->append( ownedAttribute-
>select(att | att.name = i.?column_name)))
```

Dále je navázána právě založená relace na existující relaci pomocí cizího klíče. Je vytvořen nový cizí klíč, jehož členy se stávají atributy A schématu relace A. Tento cizí klíč pak odkazuje na existující klíč schématu relace B, který je definován výčtem atributů B:

```
context: newSchema

pre:
„alter table table_nameA add foreign key
(column_nameA, ...) references table_nameB
(column_nameB, ...)"

post:
ownedKey.referencingKey->size() =
ownedKey@pre.referencingKey ->size()+1
```

```
Let newFK:ForeignKey = ForeignKey.oclIsNew() in
Let referencedRS:RelationSchema =
ownedRelationSchema->select(rs | rs.name =
?table_nameB in
Let referenceKey:Key = referencedRS.ownedKey-
>select(key | key = ?(column_nameB, ...) in
Let referencingRS:RelationSchema =
ownedRelationSchema->select(rs | rs.name =
?table_nameA in


Let i=?(column_nameA, ...)->all() in
i->iterate(i | newFK.keyMember =
newFK@pre.keyMember->append( ownedAttribute-
>select(att | att.name = i.?column_nameA)))
```

Nyní tedy existuje relační model PIM, který byl vytvořen z upraveného kódu PSM.

11.6 Doplnění implicitních klíčů

Nejdříve jsou odstraněny relace, u kterých nejsou požadovány další transformace:

```
context: DBSchema


pre:
ownedRelationSchema->select(rs | rs.taggedValue.name
= Abandoned)


post:
self.ownedRelationSchema->excludesAll(rs |
rs.taggedValue.name = Abandoned and
rs.taggedValue.dataValue = true)
```

Poté jsou zpracovány značky pro klíče (candidate keys). Jako první je požadovaný klíč vytvořen a poté je postupně naplněn atributy daného schématu relace, které jsou pro to označeny. Pořadí atributů v klíči je dáno příslušnou značkou:

```
context: RelationSchema
```

```
pre:
ownedAttribute->select(att | att.taggedValue.name =
CKSequence)->notEmpty()


post:
ownedKey->size() = ownedKey@pre->size()+1


Let newKey:Key = ownedKey.oclIsNew() in
Let newKeyMems:Set = ownedAttribute->select(att |
att. taggedValue.name = CKSequence) in


ownedKey->at(ownedKey->size()) = newKey and
newKeyMems->iterate(i | newKey.keyMember-
>at(i.taggedValue.dataValue) = i)
```

Nakonec jsou zpracovány cizí klíče. Nejdříve jsou získána jména schémat relací, které obsahují protilehlé klíče. Pro každý jejich odkazovaný klíč je vytvořen cizí klíč složený z označených atributů:

```
context: DBSchema


pre:
ownedRelationSchema.ownedAttribute->select(att |
att.taggedValue.name = FKSequence)->notEmpty()


post:
Let fKSeq:Set = ownedRelationSchema.ownedAttribute-
>select(att | att.taggedValue.name = FKSequence)-
>collect(att |
att.taggedValue.FKSequence.dataValue)->asSet() in


fKSeq->iterate(seq | Let atts:Set = select (att |
ownedRelationSchema.ownedAttribute.taggedValue.FKSeq
uence. dataValue = fKSeq) in


Let newFK:ForeignKey = ForeignKey.oclIsNew() in


Let rs:RelationSchema = ownedRelationSchema-
>select(rs | rs.name = atts-
>any().taggedValue.ReferencedCKOwner.dataValue) in


newFK.referencedKey.relationSchema = rs and
newFK.referencedKey = rs.ownedKey->at(atts-
```

```
>any().taggedValue.ReferencedCKSequence.dataValue)
and
newFK.keyMember = att
```

Po této transformaci existuje model PIM doplněný o implicitní klíče a cizí klíče. Zároveň byly odebrány nežádoucí schémata relací.

11.7 Normalizace

1 NF je definována existencí primárního klíče (a atomičností hodnot, která je předpokládána). Proto u každého schématu relace, které nemá definovaný primární klíč, bude takto nastaven libivolný z klíčů. Pokud ani klíč neexistuje, bude vytvořen jako sada všech atributů schématu:

```
context:
RelationSchema

pre:
is1NFCompliant = false

post:
if ownedKey->isEmpty() then
Let newPK:Key = ownedKey.oclIsNew() in
newPK.keyMember = ownedAttribute
endif

ownedKey->one(pk | pk.primaryKey)
```

Všechna schémata relace, která nedodržují 2NF, a obsahují tedy determinant tvořený částí primárního klíče, musejí být podle tohoto determinantu rozděleny. Determinant tvořený částí primárního klíče původního schématu relace se stane primárním klíčem nové relace. Všechny (parciální) dependanty tohoto determinantu jsou přesunuty do nového schématu relace a atributy determinantu v původní relaci se stávají cizím klíčem odkazujícím na primární klíč nové relace.

Zároveň musejí být přeneseny funkční závislosti, aby i na nově vytvořené relaci mohla být ověřena HNF.

```
context:
DBSchema

pre:
Let wrongDet:Determinant = incompletePKDeterminants-
>any() in
Let wrongRS = relationSchema->select(RS | RS.name =
???) in
ownedRelationSchema.is1NFCompliant = false[16]

post:
Let newRS:RelationSchema =
ownedRelationSchema.oclIsNew() in
Let newPK:Key = newRS.ownedKey.oclIsNew() in
Let newDet:Determinant = :Determinant.oclIsNew() in
Let newPFD:PartialFunctionalDependency =
:PartialFunctionalDependency.oclIsNew() in
Let newFK:ForeignKey = :ForeignKey.oclIsNew() in

newRS.name =
wrongRS.name.concat(wrongDet.determinantMember ...)
and
newRS.ownedAttribute =
wrongDet@pre.determinantMember +
wrongDet@pre.dependancy.dependant and
newDet.determinantMember =
wrongDet@pre.determinantMember and
newPK.keyMember = wrongDet@pre.determinantMember and
wrongRS.ownedAttribute-
>excludesAll(wrongDet@pre.dependancy.dependant) and
newFK.referencedKey = newPK and
newFK.keyMember = wrongDet@pre.determinantMember
```

Transformace do 3NF je podobná jako u druhé normální formy s tím rozdílem, že tentokrát je původní, nepřípustný determinant neklíčový.

---

[16] V této i dalších transformacích je kód podmínky nahrazen odkazem na ní. V praxi však musí být rozepsán, protože v dalším kódu je odkazováno na proměnné v ní uvedené.

```
context:
DBSchema


pre:
Let wrongRS = relationSchema->select(RS | RS.name =
???) in
Let wrongDet:Determinant =
wrongRS.ownedAttribute.determinant->any(det |
det.determinantMember <> self.primaryKeyAttributes)
in
ownedRelationSchema.is1NFCompliant = false


post:
Let newRS:RelationSchema =
ownedRelationSchema.oclIsNew() in
Let newPK:Key = newRS.ownedKey.oclIsNew() in
Let newDet:Determinant = :Determinant.oclIsNew() in
Let newPFD:PartialFunctionalDependency =
:PartialFunctionalDependency.oclIsNew() in
Let newFK:ForeignKey = :ForeignKey.oclIsNew() in

newRS.name =
wrongRS.name.concat(wrongDet.determinantMember ...)
and
newRS.ownedAttribute =
wrongDet@pre.determinantMember +
wrongDet@pre.dependancy.dependant and
newDet.determinantMember =
wrongDet@pre.determinantMember and
newPK.keyMember = wrongDet@pre.determinantMember and
wrongRS.ownedAttribute-
>excludesAll(wrongDet@pre.dependancy.dependant) and
newFK.referencedKey = newPK and
newFK.keyMember = wrongDet@pre.determinantMember
```

Transformací uvedenou v 3NF jsme zároveň docílili dosažení BC normální formy, další normalizační transformace jsou zbytečné.

11.8 Legenda λ výrazů

R	schéma relace
E	entita
A	atribut schématu relace

AE	atribut entity (nebo atribut asociace)
PK	primární klíč
FK	cizí klíč
R_o	schéma protilehlé relace (schéma relace, na kterou odkazuje cizí klíč)
Card	kardinalita v EER modelu (pokud není definována, je typu m)
Mult	multiplicita v objektovém modelu
Assoc	asociace v EER modelu
AssocC	asociace v objektovém modelu
Ident(E)	identifikátor entity
Gen	generalizace v EER modelu
GenC	generalizace v objektovém modelu
C	třída

funkce a operátory:

◁	navigace mezi operandy
◁:	aplikace proměnné na lambda výraz
≻	generalizace (potomek je v levé části)
concat	sloučení dvou textových proměnných
new	vytvoření nového elementu nebo nově vytvořený element
//	selekce

12 Literatura

[ALHA02] R. Alhajj: Extracting the EER model from a legacy relational database, *Information Systems Volume 28 , Issue 6 (September 2003),* pp. 597 – 618, ISSN:0306-4379

[AMBL03] S. Ambler: Agile Database Techniques: Effective Strategies for the Agile Software Developer, Wiley, 2003. ISBN 0471202835. Online: http://www.agiledata.org/essays/objectOrientation101.html, http://www.agiledata.org/essays/relationalDatabases.html

[ANDE94] M. Andersson.: Extracting an Entity Relationship Schema from a Relational Database through Reverse Engineering, *Proc. of the Int. Conf. on the Entity-Relationship Approach (ERA)*, Manchester, 1994, pp. 403-419

[ASTR04] I. Astrova, B. Stantic: Reverse Engineering of Relational Databases to Ontologies: An Approach Based on an Analysis of HTML Forms, *In Proceedings of the 1st European Semantic Web Symposium (ESWS)*, LNCS 3053 (2004)

[ATKI89] M. Atkinson: The Object-Oriented Database System Manifesto, Proceedings of the First International Conference on Deductive and Object-Oriented Databases, 1989

[ATKI02] C. Atkinson, T. Kühne: The Role of Metamodeling in MDA, *International Workshop in Software Model Engineering (in conjunction with UML '02)*, Dresden, Germany, October 2002

[BARB01] A. Barbar, M. Collard: Semantic Extraction: a User-Driven Method, *4th International Conference on Information Systems Modelling*, ISM'01, Republique Tchèque, 2001, pp 77-84.

[BLAH95] M. Blaha, W. Premerlani: Observed Idiosyncracies of Relational Database designs, *in Proc. of the 2nd IEEE Working Conf. on Reverse Engineering*, Toronto, July 1995, IEEE Computer Society Press

[BONC98] P. Boncz, A. Wilschut, M. Kersten: Flattening an Object Algebra to Provide Performance, *Proceedings of the Fourteenth International Conference on Data Engineering*, pp.568 – 577. 1998. ISBN:0-8186-8289-2

[CASA83] M. Casanova, J. Amaral de Sa: Designing Entity-Relationship Schemas for Conventional Information Systems, *Proceedings of the Third International Conference on Entity-Relationship Approach*, 1983

[CERI85] S. Ceri, G. Gottlob: Translating SQL into relational algebra: optimization, semantics, and equivalence of SQL queries. *IEEE Trans. Softw. Eng. 11*, 4 (Apr. 1985), 324-345.

[CODD70] E. Codd: A relational model of data for large shared data banks. *Commun. ACM 13*, 6 (June 1970), 377-387

[CODD72] E. Codd: Further normalization of the data base relational model. *In Courant Computer Science Symposium 6: Data Base Systems*, R. Rustin, Ed., Prentice-Hall, 1972, pp. 33-64.

[CODD74] E. Codd: Recent investigations in relational data base systems. *In Information Processing 74*, North-Holland, 1974, pp. 1017–1021
[DAVI00] K. Davis, P. Aiken: Data Reverse Engineering: A Historical Survey. *Proceedings of the 7th Working Conference on Reverse Engineering WCRE´Ż2000*, Brisbane, Queensland, Australia, 2000, p70-78.
[DOWE95] M. Dowell, L. Stephens, R. Bonnell: Using Metamodels as Part of a Semantic Gateway among Databases, *USC Technical Report ECE-MLD-030-95*. Online: http://www.aug.edu/~mcsmld/using-metamodels-as-part.pdf
[FAGI77] R. Fagin: Multivalued dependencies and a new normal form for relational databases. *ACM Trans. Database Syst. 2*, 3 (Sept. 1977), 262-278
[FAGI79] R. Fagin: Normal forms and relational database operators. *In Proceedings of the 1979 ACM SIGMOD Conference,* P. A. Bernstein, Ed., pp. 153-160
[FONG93] J. Fong,, M. Ho: Knowledge-Based Approach for Abstracting Hierarchical and Network Schema Semantics, *Proceedings of the Twelve International Conference on Entity-Relationship Approach*, Dallas, 1993.
[GOGO02] M. Gogolla: Meta-model Transformation of Data Models, *Workshop in Software Model Engineering in Dresden*, Germany, 2002
[GOGO05] M. Gogolla: An Example for Metamodeling Syntax and Semantics of Two Languages, their Transformation, and a Correctness Criterion, *in Proceedings of the Language Engineering for Model-Driven Software Development*, 2005
[HAIN95] J.-L. Hainaut, V. Englebert, J. Henrard, J.-M. Hick, D. Roland: Requirements for Information Systems Reverse Engineering Support, *Proc. of the Int. Working Conference on Reverse Engineering (WRCE)*, Toronto 1995.
[HAIN98] J.-L. Hainaut: Database Reverse Engineering, *Proc. of the 10th Conf. on ER Approach*, San Mateo (CA), 1998
[CHEN76] P. Chen: The Entity-Relationship Model: towards a unified view of data; *ACM TODS*, Vol. 1, Nb.1, 1976
[CHIA94] R. Chiang, T. Barron, V. Storey: Reverse Engineering of Relational Databases: Extraction of an EER Model from a Relational Database, *Data and Knowledge Engineering (DKE)*, 12, 1994 pp. 107-142
[CHIK96] E. Chikofsky: Předmluva k Data Reverse Engineering: Slaying the Legacy Dragon, McGraw-Hill, 1996, str. xiii – xvi.
[JOYN97] I. Joyner: Open Distributed Processing: Unplugged! Online: http://homepages.tig.com.au/~ijoyner/ODPUnplugged.html
[KIFE90] M. Kifer, G. Lausen, J. Wu: Logical Foundations of Object-Oriented and Frame-Based Languages. *Technical Report 90/14*, Department

of Computer Science, State University of New York at Stony Brook (SUNY), June 1990.
[KORT99] H. Korth, X. Peltier: Query Algebra for Object Oriented Databases, *in Proceedings of the Schlumberger Software Conference*, 1990
[MORA02] B. Morand: Models Transformation: From Mapping to Mediation, *Workshop in Software Model Engineering in Dresden*, Germany, 2002
[NAVA87] S. Navathe, A. Awong: Abstracting Relational and Hierarchical Data with a Semantice Data Model, *Proceedings of the Sixth International Conference on Entity-Relationship Approach*, 1987
[NUTS02] H. Nootenboom, OONF, *Nut's The Online Column 2002*. Online: http://www.sum-it.nl/en200239.html
[ODMG00] R. Cattell (eds.): The Object Database Standard: ODMG 2.0, January 2000, Morgan Kaufmann, ISBN 1-55860-647-5
[ODP_96] ISO/IEC 10746-3: Information Technology – Open Distributed Processing – Reference Model, 1996
[OMG_01] J. Miller, J. Mukerji (eds.): Model Driven Architecture (MDA), Object Management Group, Document number ormsc/2001-07-01, July 9, 2001.
[OMG_03] J. Miller, J. Mukerji (eds.): The MDA Guide, Object Management Group, Document number omg/2003-06-01, June 12, 2003.
[OMG_05] J. Miller, J. Mukerji (eds.): Unified Modeling Language: Superstructure, Object Management Group, Document number formal/05-07-04, August, 2005
[OMG_05-2] J. Miller, J. Mukerji (eds.): OCL 2.0 Specification, Object Management Group, Document number ptc/05-06-06, June, 2005
[PETI94] J.-M. Petit, J. Kouloumdjian, J.-F. Boulicaut, F. Toumani: Using Queries to Improve Database Reverse Engineering, *Proc. of the Int. Conf. on the Entity-Relationship Approach (ERA)*, Manchester, 1994, pp. 369-386
[PREM93] W. Premerlani, M. Blaha: An Approach for Reverse Engineering of Relational Databases, *in Proc. of the IEEE Working Conf. on Reverse Engineering, 1993*, IEEE Computer Society Press
[PREM94] W. Premerlani, M. Blaha: An Approach for Reverse Engineering of Relational Databases, *Communications of the ACM (CACM)*, 37(5), 1994, pp. 42-49
[PROP04] H. Proper, T. Halpin. Conceptual Schema Optimisation: Database Optimisation before sliding down the Waterfall. *Technical Report 341*, Department of Computer Science, University of Queensland, Brisbane, Australia, July 1995. Version of June 2004
[RUBI03] E. Domínguez, J. Lloret, A. Rubio, M. Zapata. An MDA-based Approach to Managing Database Evolution. Model-Driven Architecture:

Foundations and Applications; *CTIT Technical Report Series, No. 03-27*. ISSN 1381-3625

[RUND92] E. Rundensteiner, L. Bic: Set Operations in Object Based Data Models, *IEEE Transactions on Knowledge and Data Engineering, Vo. 4, No, 3,* 1992, pp. 382-398

[SHAW89] G. Shaw, S. Zdonik: A Query Algebra for Object-Oriented Databases. *Technical Report CS-89-19*, Department of Computer Science, Brown University, Providence, Rhode Island, Mar 1989.

[STON90] M. Stonebraker, L. Rowe, B. Lindsay, J. Gray, M. Carey, M. Brodie, P. Bernstein, D. Beech: Third-generation database system manifesto. *In Readings in Database Systems (2nd Ed.)*, M. Stonebraker, Ed. Morgan Kaufmann Series In Data Management Systems. Morgan Kaufmann Publishers, San Francisco, CA, 1994, pp.932-945.

[TARI97] Z. Tari, O. Bukhres, J. Stokes, S. Hammoudi: The Reengineering of Relational Databases based on Key and Data Correlations. *In Searching for Semantics: Data Mining, Reverse Engineering, etc.*, S. Spaccapietra and F. Maryanski (eds.), Chapman & Hall, 1998.

[TEOR99] T. Teorey: Database Modeling & Design, 1999 Academic Press, 3rd. Edition, ISBN 1558605002

[TSIC78] D. Tsichritzis, A. Klug: The ANSI/X3/SPARC DBMS Framework *Report of the Study Group on Database Management Systems*. Information Systems, 1:173–191, 1978.

[WARM03] J. Warmer, A. Kleppe: The Object Constraint Language Second Edition – Getting Your Models Ready For MDA, 2003 Addison Wesley, 1st. Edition, ISBN 0321179366