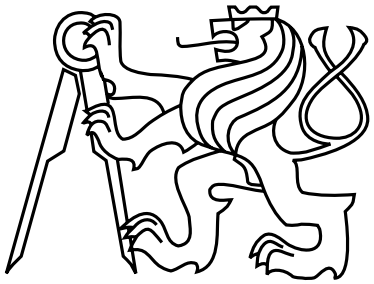
Zadání práce

Navažte na existující projekt MigDB. Vytvořte formální model v jazyce Z pro aplikační úroveň frameworku. Tento model rozšiřte o popis chování aplikace (metody a jejich volání) a zhodnoťte omezení či dopad změn chování vzhledem k datovému modelu aplikace. Pro refaktoringy definované Fowlerem definujte jejich sémantiku v jazyce Z vzhledem k vytvořenému modelu. U vzniklého modelu ověřte vhodné vlastnosti (úplnost, správnost atd.).

České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů



Diplomová práce

MigDb - formální specifikace

Bc. Petr Tarant

Vedoucí práce: Ing. Ondřej Macek

Studijní program: Otevřená informatika, Magisterský

Obor: Softwarové inženýrství

4. června 2014

Poděkování

Chtěl bych poděkovat panu Ing. Ondřeji Mackovi za nekonečnou trpělivost při práci se mnou a výborné mentorování celého týmu The MigDB. Dále všem, kteří se se mnou v průběhu několika posledních let podíleli na vývoji, jmenovitě Bc. Jiřímu Ježkovi, Bc. Martinu Lukešovi, Bc. Martinu Mazanci a také pánům Pavlu Moravcovi a Davidu Harmancovi, Ph.D., za jejich ochotu, čas a dobrou spolupráci.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 30. 4. 2014 .....

Abstract

Project Database Migration attempts to enrich the process of automatic database migration in a revolutionary way. Our goal is to change the database schema corresponding with the changes made in the application object model without any data loss. My role in the project focuses mainly on expanding the existing formal meta-model of the application by elements which represent behavior of class, i.e. methods and expressions. Furthermore, my work consists of defining the transformation rules which these new meta-model elements can change during the evolution of the application. Thanks to this extension, it will be possible to create quite complex transformations in a single instrument.

Abstrakt

Projekt Migrace databáze se snaží revolučním způsobem obohatit automatické migrování databází. Cílem je změna databázového schématu na základě změn na aplikační úrovni a to bez ztráty jakýchkoliv dat. Má práce na projektu spočívá v rozšíření stávajícího formální meta-modelu aplikace o elementy, reprezentující objektové chování tříd, tedy metody a výrazy. Dále pak ve vytvoření transformačních pravidel, které tyto elementy dokáží měnit v rámci evolučního procesu aplikace. Díky tomuto rozšíření bude v budoucnu možné vytvářet zcela komplexní transformace v jednom jediném nástroji.

Obsah

Seznam obrázků

Kapitola 1

Úvod

Projekt Migrace Databáze se již několik let zabývá problematikou evolučních změn v životním cyklu aplikace. Snahou je vytvořit takový automatizovaný proces, který vyřeší objektově-relační mapování bez ztráty informací v relační databázi. Unikátnost projektu tkví právě v zachování instancí i přes změnu databázové struktury a aplikační logiky.

V reálném provozu tato automatizace změn v aplikaci znamená nejen statické přegenerovávání databáze v závislosti na stavu aplikační vrstvy, ale také generování příslušných SQL dotazů, které pohybují instancemi na té nejnižší úrovni.

Evoluční proces chápeme jako sekvenci stavů, které konvergují k vyžadovanému výsledku. Kvůli urdržení software v konzistentním stavu a zajištění zachování instancí, je nutné postupovat po co nejatomičtejších krůčcích. Cestu z původního stavu A do finálního stavu B je tedy nutné rozsekat na mnoho malých mezistavů, které reprezentují jednu drobnou změnu skrze všechny vrstvy aplikace (od objektové vrstvy až po SQL přesuny přímo na úrovni relační databáze).

Evoluce je konstruována na abstraktní modelové úrovni, která se skládá ze dvou základních vstev - aplikační a databázové. Transformací těchto modelových vrstev na základně meta-modelových operací dochází k požadované simulaci evolučního procesu.

Snahou této práce je rozšíření aplikačního meta-modelu o schopnost chování na modelové úrovni. Jedná se tedy o rozšíření perzistentních entit o metody, respektive výrazy a konstruktory. Rozšíření struktury frameworku musí stále dodržovat pravidla konzistence a neporušenost informace v databázi.

V této práci budete seznámeni se současným stavem projektu, strukturou frameworku The MigDB a dalšími konkurenčními řešeními. Dále bude blíže popsána metoda pro formální popis modelu a samotná konstrukce rozšíření stávájícího statického modelu, a to od definování meta-modelu až po vlastní transformačí pravidla na aplikační úrovni.

Cílem práce není vytvořit kompletní sadu transformačních pravidel, která bude kopírovat nejběžnější katalogy refaktoringů. Snahou je vytvořit formální meta-model, který bude do budoucna snadno rozšířitelný o jakékoliv dodatečné vlastnosti a potřebné funkce.

1.1 Historie projektu

Projekt vznikl v roce 2011 ve spolupráci s firmou CollectionsPro, s.r.o. (Dále jen CP) a Katedry počítačů fakulty Elektrotechnické Českého vysokého učení techniského v Praze. V prvním roce se tým, pracující na projektu, soustředil hlavně na adaptaci s technologiemi a vývojovým prostředím. Jejich výstupem bylo vytvoření funkčního vývojového systému, který se skládal z vybraných programovacích jazyků a k nim odpovídajícím IDE.

Další fází vývoje byla samotná implementace meta-modelové struktury a tvorba transformací. Jelikož je systém, který využívá firma CP, postavený na velice abstraktní meta-modelové struktuře, musel framework, který do ní bude později implementován, odpovídat jejich modelovým pořadavkům. Z toho vyplývá i struktura frameworku The MigDB, která se skládá z několika vrstev. Transformace pak bylo nutné implementovat nejen na jednotlivých vrstvách, ale i skrze celou strukturu.

Výsledky dosavadní práce byly v roce 2012 prezentovány jako case-study na prestižní modelové konferenci Code Generation 2012 [16] v Anglickém Cambridge.

V současné době se na základě projektu vypracovává několik diplomových prací. Bc. Martin Lukeš [11] ve své práci dotahuje k dokonalosti celý koncept transformací a meta-modelových operací na úrovni programovacích jazyků. Bc. Martin Mazanec [12] se na druhou stranu zabývá tvorbou nového, uživatelsky přívětivého jazyka pro obsluhu transformačních operací frameworku.

Tato práce navazuje na již definovaný formální model frameworku, který je vytvořený v jazyce Z.

Už od svého počátku je celý vývoj The MigDB [19] brán jako ryze experimentální projekt. V současné době ještě není prokázáno, že budeme do budoucna schopni vytvořit tak univerzální množinu operací, že budou schopné pokrýt úplně všechny transformační modulace, které mohou v průběhu vývoje software nastat.

Kapitola 2

Účel

V současné době umí framework modelovat a posléze migrovat pouze plně statické systémy. V aplikačním meta-modelu zcela chybí entity, které by zahrnovaly metody a s tím další spojené prvky jako výrazy nebo konstruktory.

Tehdy, kdy se mezi meta-modelové entity přiřadí i prvky definující chování na aplikační úrovni, modely, které bude umět framework transformovat, se tím v mnohém přiblíží zkutečným systémům. Jde totiž o to nabídnout uživateli maximum refaktorovacích možností, které bude moci nad svým software vykonávat. Přidáním práce s metodami a výrazy se funkcionalita frameworku vysloveně znásobí.

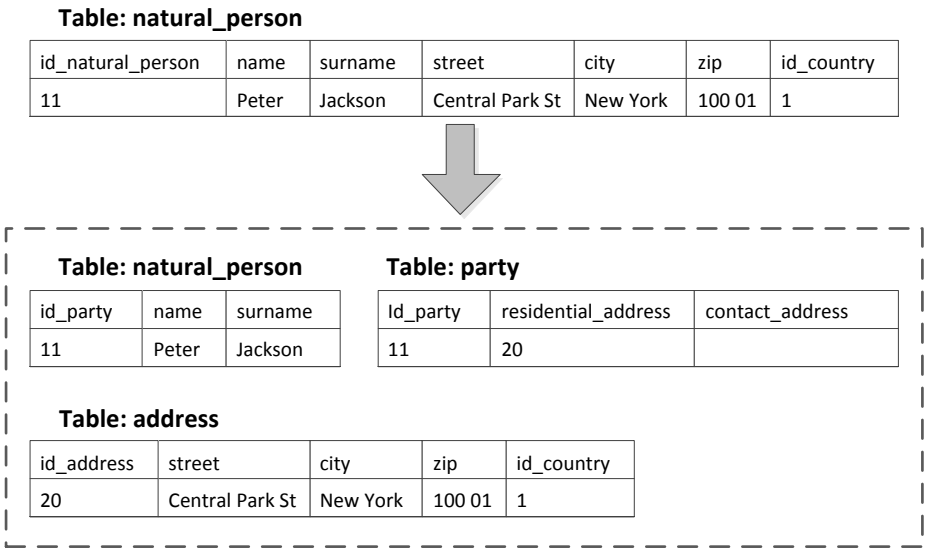
V budoucnu tak bude možné plně využít potenciálu refaktoringových katalogů, a to jen za použití frameworku The MigDB. Programátor bude schopen obsluhovat celý evoluční proces své aplikace pouze skrze jednu konzoli a příkazy bude psát pouze jedním jazykem. V podstatě se bude jednat o kombinaci migračních nástrojů z IDE, které pomáhají programátorům v základních refaktoringových operacích a objektově-relačních frameworků, které řeší distribuci této změny skrze všechna patra aplikace.

Přidání metod a výrazů do formálního modelu s sebou přináší zcela novou dimenzi problémů. Musí být zajištěna typová konzistence, neboli věc, která se u statických elementů musela řešit jen v souvislosti s existencí či neexistencí používané třídy. U metod například musí být kontrolována celá řada typových omezení od návratové hodnoty, až po výraz, který má uložený ve svém těle.

2.1 Datová a informační konzistence

Nejdůležitější podmínkou pro jakékoliv rozšíření frameworku The MigDB je zachování datové a informační konzistence. Datová konzistentnost v relačních databázích označuje stav, kdy konkrétní data splňují pravidla a omezení, která jsou na ně kladena [6]. Aplikace však v rámci transformací mohou pravidla porušit a dostat se tak i do nekonzistentního stavu. Často se jedná o problém s vazbami skze cizí klíče.

Informační konzistence na druhou stranu neznamená porušení databázových pravidel. Jde o udržení logiky informací. Data se v databázi mohou nejrůznějším způsobem přesouvat, ale stále musí být zaručeno, že budou držet stejnou informační hodnotu (obrázek 2.1).



Obrázek 2.1: Zachování datové informace - Informační hodnota dat o Petru Jacksovoni zůstala stejná i po transformačním rozdělení do tří tabulek.

2.2 Rozšířený model a datovo-informační konzistence

Otázkou je, zda se po rozšíření meta-modelu o metody a výrazy může stát, že by byla nějakým způsobem narušena datová či informační konzistence.

Frameworky jako Hibernate [1] nebo JPA [2] pro své objektově-relační mapování využívají takzvané POJO [3] objekty. Jedná se o zcela jednoduché třídy, u kterých platí jedno základní pravidlo - co není anotované, není Entity Managerem persistované. Díky tomu metody a konstruktory zcela odpadají ze hry. Neexistuje totiž žádný způsob, kterým by se dala anotovat metoda tak, aby se v rámci objektově-relačního mapování uložila do databáze. Jinak řečeno v nejrozšířenějších JAVA frameworcích (zabývajících se mapováním) v dnešní době práce s metodami zcela odpadá.

Evoluce software se dá chápat dvěma způsoby. Prvním typem evoluce, nebo v tomto případě spíše synchronizace, je evoluce "vertikální", kdy dochází k propagaci změn skrze všechny vrstvy aplikace - ORM [13]. Zástupci jsou například již zmínění Hibernate nebo JPA. Druhým způsobem je evoluce "horizontální", kdy se model přesouvá z jednoho stavu do druhého, a to včetně ORM transformací v rámci každého stavu. The MigDB framework jde právě touto cestou.

Takzvaná horizontální evoluce je složená ze sady stavů, které obsahují vertikální evoluci, tedy objektově-relační mapování. To z podstaty věci nepodporuje perzistenci metod, protože

[1] Hibernate: http://www.hibernate.org/
[2] JAVA Persistence API: http://goo.gl/FMc5vL
[3] Plain Old JAVA Object: http://www.martinfowler.com/bliki/POJO.html

neexistuje žádný rozumný způsob jak a proč ukládat metody do databáze. Díky tomu se ale dostáváme k odpovědi na otázku, zda mohou metody narušit datovou či informační konzistenci dat v databázi. Nemohou. Jde totiž o vlastnosti, které jsou na databázové vrstvě zcela nezávislé.

Metody jako takové sice využívají atributy, parametry a vrací návratové hodnoty, ale nic z toho nemění. Neexistuje metoda, která by uměla smazat atribut, nebo ukončit existenci třídy. Všechny tyto prvky pouze využívá ke své vlastní práci. Dá se tedy říci, že vůči persistovaným entitám jsou metody jen read-only.

U výrazů je odtrženost od databáze ještě vyšší, protože ty dokonce vytváří ještě jednu abstraktní vrstvu nad entitami aplikační vrstvy. Nutno říci, že také pouze read-only. Neexistuje žádný výraz, který by stejně jako u metod mohl smazat atribut nebo třídu.

Kapitola 3

Současný stav

V dnešní době neexistuje žádný oficiálně představený framework nebo nástroj, který by umožňoval dynamické migrování aplikací s ohledem na instance uložené v databázi. Pro programátory to tedy znamená v lepší případě správu skriptů, které jsou přímo vytvořené na míru lokálnímu databázovému systému, nebo v horším případě psaní nekonečného množství SQL příkazů, které zařídí přesun dat do cílové struktury. V obou případech se však nejedná o změny v rámci evolučního procesu aplikace. Jde pouze o vytvoření nové aplikace s novou databází, do které se přesouvají data z původní struktury.

Následky takového přístupu jistě pocítila každá větší firma či veřejný podnik. Oba přístupy jsou totiž zatíženy obrovským ryzikem lidského pochybení, které v takových případech jen zřídkakdy nenastane. Není se ani čemu divit. V okamžiku, kdy dochází k ručnímu migrování distribuované firemní databáze s několika miliony záznamů, musí být kontrolováno tolik databázových omezení, že toho lidská pozornost ani nemůže být schopná.

Jako příklad velkého migrování databáze na základě změny aplikace si můžeme uvést snahu o reformování centrálního informačního systému registru vozidel Ministerstva dopravy v roce 2013 [20].

3.1 Konkurenční řešení

Dosud není znám žádný nástroj, který by se snažil o dynamické změny databází na základě změn v aplikaci. Existují však projekty nebo patenty, které řeší alespoň "statické"migrování, tedy přesun dat z databáze A do databáze B.

Jedním takovým patentem je například americký Database Migration[21]. Ten je založený na vytváření dočasných tabulek, které slouží jako mapování mezi oběma databázemi. Celý proces přesunu a generování tabulek je řízen speciálními skripty, které vytváří pohledy nad daty a nastavují data do správných formátů.

Mezi frameworky a nástroji v dnešní době je však většina těch, které se o instance ve svých strukturách vůbec nezajímají. Do jazyka JAVA implementovaný framework JPA například datové instance vůbec neuvažuje. Na druhou stranu jeho schopnost převést anotované POJO objekty do relačních databází skrze Entity Manager je v rámci ORM téměř dokonalá. JPA dokáže reflektovat jakékoliv změny na aplikační úrovni a vytvořit tomu přesnou databázovou

předlohu. Bohužel však při jakékoliv změně třídy na aplikační úrovni se automaticky smaže a předělá i celá databázová tabulka.

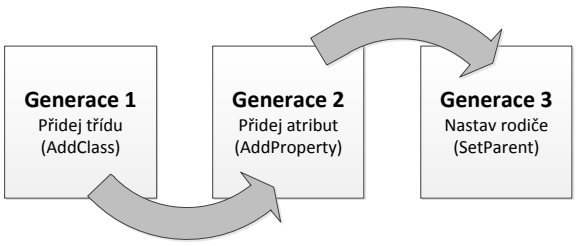
Další frameworky a nástroje, které se soustředí výhradně jen na objektově-relační mapování a instance v databázi nemilosrdně ničí, jsou například Hibernate a OpenJPA [1] pro jazyk JAVA , LiteSQL [2] a ODB [3] pro C++, Zend Framework [4] pro PHP, ActiveRecord [5] pro Ruby on Rails a v neposlední řadě například Entity framework [6] od Microsoftu. Poslední dva jmenovaní zástupci migračních frameworků sice ničí instance v databázi, portfólio operací se u nich však neomezuje pouze na ORM mapování. ActiveRecord a Entity framework nabízí i evoluční transformace v podobě vytvoření třídy a podobně.

3.2 Řešení The MigDB

Celý framework pracuje na meta-modelové a modelové úrovni abstrakce. Základním hybným kamenem celého systému jsou transformace mezi jednotlivými úrovněmi systému. The MigDB framework se skládá ze dvou základních modelových vrstev - Aplikační a Databázové. Na každé vrstvě dochází k horizontálním Model-to-model transformacím, kdy se přechází z jednoho stavu/generace modelu do druhého.

Aby se dodržela maximální datová a informační konzistence, jsou kroky na jednotlivých vrstvách co nejatomičtější, protože jen při takových krocích se dá zajistit neustálá konzistence jak aplikační vrstvy, tak databáze.

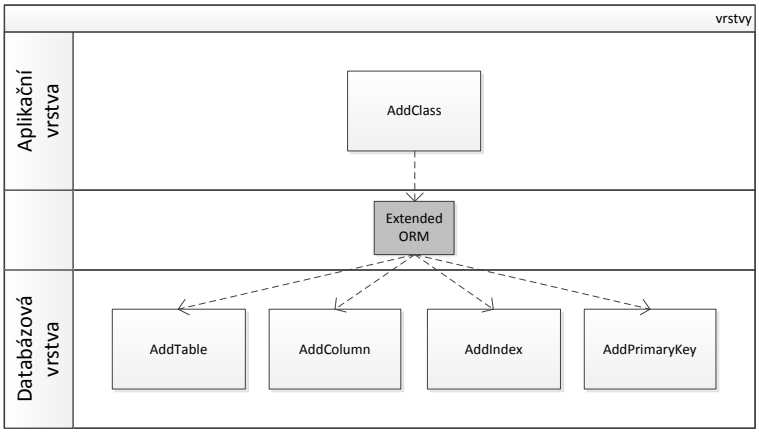
Pokud tedy chceme na aplikační úrovni vytvořit nějakou změnu, poslouží nám k tomu sada operací, které aplikační meta-model nabízí. Sestavíme tak sekvenci operací, které postupně model dokonvergují do požadovaného stavu. Každé aplikování operace tak vytvoří nový stav celého systému v evolučním procesu-viz obrázek 3.1.



Obrázek 3.1: Rozdělení evolučního procesu na generace

[1] OpenJPA: http://openjpa.apache.org/
[2] LiteSQL: http://sourceforge.net/apps/trac/litesql/
[3] ODB: http://organizersdb.org/
[4] Zend Framework: http://framework.zend.com/
[5] ActiveRecord: http://ar.rubyonrails.org/
[6] Entity Framework: http://msdn.microsoft.com/en-us/library/aa697427(v=vs.80).aspx

Každá operace z aplikačního meta-modelu, která je aplikována na aplikační model, má své odpovídající databázové operace, které se formou ORM automaticky vygenerují, aby přizpůsobily databázovou vrstvu aplikačním změnám. Je třeba si uvědomit, že mapování z aplikačních operací na databázové není jedna ku jedné. Databázových operací se může z důvodu jedné drobné aplikační změny vygenerovat celá sada jako na obrázku 3.2.



Obrázek 3.2: Mapování operací

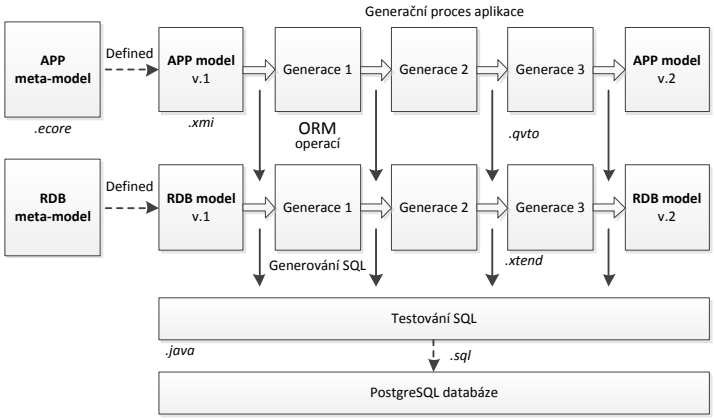
Celá struktura frameworku i s použitými technologie pak vypadá při svém chodu jako na obrázku 3.3. Zde je jasně vidět role databázového i aplikačního meta-modelu, které definují, jakých tvarů a hodnot mohou modely nabývat. Mezi jednotlivými stavy/generacemi evoluce pak dochází jak k horizontální Model-to-model transformacím, tak k vertikálním změnám skrze všechny vrstvy software, tedy ORM.

Pro definování meta-modelů se využívá technologie Ecore neboli Core EMF, která je součástí Eclipse Modeling Frameworku [7]. Ecore je vizuální nástroj, díky kterému lze snadno a rychle definovat modely a meta-modely bez znalosti nových jazyků. Jedná se tedy pouze o vizualizaci XMI [15]. Modely samotné jsou pak exportované do jazyka JAVA.

Pro Model-to-model transformace je v projektu využíván jazyk QVT (Query/View/Transformation) [14]. Jedná se o imperativní transformační jazyk definovaný OMG. Nejsilnější stránkou jazyka je podpora komponenty OCL [17], která dovoluje mnohem intuitivnější a snadnější práci s elementy v modelu.

Aby modelové vrstvy dokázaly komunikovat s reálnou databází, je nutné vytvářet Motel-to-text transformace, které budou z vygenerovaných databázových modelových operací vytvářet SQL příkazy. K tomuto účelu slouží jazyk Xtend [9], který je z rodiny jazyků Xtext [10]. Xtend je staticky-typový funkcionální a objektově orientovaný jazyk postavený nad jazykem JAVA. Nejsilnější stránkou jazyka Xtend je polymorfismus nad parametrem a obohacování řetězců (rich-string).

Framework je technologicky velmi nehomogenní. Aby došlo ke spolupráci všech zmíněných technologií, využívá se jazyka MWE2 [8]. Jedná se o deklarativní, externě nastavitelný nástroj, který slouží k popisu libovolné objektové struktury. V jazyce MWE2 lze technologie obalovat do komponent, které mezi sebou komunikují svými vstupy a výstupy (sloty). Ze sekvence komponent pak vzniká pracovní tok, který uvádí celý framework do chodu.



Obrázek 3.3: Struktura frameworku

Kapitola 4

Formální metody verifikace

Formální metody jsou speciální návrhové techniky, které k tvorbě systémů využívají přísně specifikované matematické modely. Na rozdíl od ostatních návrhových systémů používají formální metody matematický důkaz jako dolňující prvek k testování. Formální specifikace obecně popisuje, co musí systém dělat bez toho, aniž by bylo řečeno, jak to má dělat [18].

Jak se systémy stávají stále složitějšími, narůstá také poptávka po udržení bezpečnosti. Tu mohou formální metody zajistit na ještě vyšší úrovni abstrakce [3].

Největší rozdíl oproti standardním metodám však je hlavně ve verifikačních schématech, ve kterých musí být striktně prokázáno, že základní principy systému jsou správné ještě před jeho přijetím [2]. Tradiční modely pro ověření konzistence a validity systému využívají rozsáhlé testy chování. Ty však, stejně jako ostatní tradiční testovací techniky, prokáží pouze to, že systém funguje v daném testovacím scénáři. Nezískáme však informace o tom, že systém funguje, i pokud pracuje mimo testovací scénář.

Je důležité si uvědomit, že formální metody nikdy nenahradí standardní způsob testování software. Oba způsoby by se měly při vývoji doplňovat. Formální verifikace například nedokáže opravit špatný návrh systému, ale dokáže poukázat na problémy v uvažování, které by jinak mohly zůstat přehlédnuty.

Podle [3] se dá proces návrhu formálními metodami rozdělit na tři základní kroky.

1. **Formální specifikace**
   Jde o velice důslednou snahu o popis systému za pomoci nejrůznějších modelovacích jazyků. Základní vlastností těchto jazyků je přísně ukotvená gramatika, která dovoluje uživateli nadefinovat nejrůznější složité struktury jen z předdefinovaných typů. Celá tato fáze je vlastně snaha o napasování problému ze zkutečného světa do striktně pracující algebry.

2. **Verifikace**
   Formální metody kladou obrovský důraz na správnost a průkaznost systému. Takže už při samotné tvorbě a definování modelu vytváří vývojář určitá tvrzení, teorémy. V okamžiku, kdy je schopen potvrdit, že nad daným modelem tvrzení platí, provedl verifikaci systému.

3. **Implementace**
   U specifikovaného a verifikovaného modelu jde už jen o převedení algebraického zápisu do konkrétního programovacího jazyka s vlastní syntaxí a sémantikou.

4.1 Jazyk Z

Formální specifikační jazyk Z [18] je pojmenovaný podle Zermelo-Fraenkelově množinové teorii[1]. V roce 1974 ho jako potřebný specifikační nástroj použil Jean-Raymond Abrial [1] ve svém článku Data semantics. Tento jazyk pak požívala University of Grenoble skoro 10 let pro své výzkumy. Jako "Jazyk Z"notaci Abrial pojmenoval až ve svých poznámkách na přelomu osmdesátých let.

Jazyk Z je pouze notačním nástrojem, není exekuční a také se nedá považovat za programovací jazyk. Mezi jeho základní vlastnosti patří používání lambda kalkulu [2], axiomatické množinové teorie a predikátová logika prvního řádu. Všechny výrazy (schémata, entity) v Z notaci jsou typované.

Jazyk Z je postavený na modelech. Systém je pak modelován za pomoci stavů, tedy sadě stavových proměnných a jejich hodnot a operací, které můžou stavy měnit. Silnou zbraní notace jsou schémata, za pomoci kterých se vytváří vzory deklarací a omezení.

Jelikož je jazyk Z pouze notací, je takovým způsobem používána i v této práci. Existuje sice nástroj pro dokazování, který se snaží o ověřování zadaných teorémů, ale k jeho zprovoznění jsou třeba na dnešní dobu skoro nesplnitelné požadavky v podobě Pythonu 1.2 nebo 8 bit jádra systému [3].

4.2 Používané funkce jazyka Z

Jak již bylo řečeno jazyk Z je silně typový. Umí pracovat jen s těmi typy, které jsou v modelu deklarované. Základní deklarace nových typů může být zapsána například takto.

$$[PERSON, DOCUMENT]$$

Vytváření axiomatických popisů se v notaci Z zapisuje takto.

$$\begin{array}{|l} maxUsers : \mathbb{N} \\ \hline maxUsers < 100 \end{array}$$

Deklarace nad čarou říká, že *maxUsers* je nezáporné číslo. Predikát pod čarou omezuje hodnoty deklarace. Položky v axiomatickém popisu jsou vždy konstanty.

Schéma pak definuje nějaký důležitý koncept, spojení několika proměnných dohromady.

$$\begin{array}{|l} \textit{INHERITANCE} \\ \hline parent : CLASS \\ child : CLASS \\ \hline \end{array}$$

[1] Jean-Raymond Abrial: http://www.ae-info.org/ae/User/Abrial_Jean-Raymond
[2] Lambda Calculus: http://plato.stanford.edu/entries/lambda-calculus/
[3] Z/EVES: http://oracanada.com/z-eves/welcome.html

4.2.1 Predikáty a výrazy

Základním stavebním kamenem jazyka Z je definování predikátů a výrazů. Vlastnosti výrazů používané v textu jsou následující:

- $\mathbb{P}\, P$ značí potenční množinu. Tedy množinu všech podmnožin množiny P.
- Lokální definice se značí znakem $\bullet$.
- Sekvence se značí ve zkosených závorkách jako $\langle E \rangle$.
- Dekorace proměnných může být vstupní $x?$, výstupní $x!$ nebo vytvoření kopie pro další práci x'.

Nejpoužívanější značky mezi predikáty v textu jsou:

- Rovnost $a = b$, nerovnost $x \neq y$, členství $a \in S$, nečlenství $a \notin S$.
- Z logických operátorů to jsou negace $\neg\ P$, konjunkce $P \land Q$, disjunkce $P \lor Q$, implikace $P \Rightarrow Q$, ekvivalence $P \Leftrightarrow Q$
- Pro všechna ST se značí $\forall\, ST \bullet P$ a existuje ST se značí $\exists\, ST \bullet P$.

4.2.2 Množiny a sekvence

Většina prvků v modelu MigDB je uložena v nějaké sekvenci nebo množině. Pokud v ní sama uložena není, tak určitě nějakou vlastní. Proto je práce s množinami v invariantech a transformacích velice důležitá.

Z množinových operací jsou nejvyužívanějšími:

- Prázdná množina $\varnothing$, podmnožina $S \subset T$.
- Sjednocení $S \cup T$, průnik $S \cap T$, rozdíl $S \setminus T$

Při práci se sekvencemi se nejvíce využívají následující funkce:

- Spojení dvou sekvencí tak, že sekvence t bude připojena za sekvenci s $s \frown t$.
- Získání prvního prvku $head\ s$, posledního prvku $last\ s$.
- Všech prvků kromě prvního $tail\ s$, všech prvků kromě posledního $front\ s$.
- Extrakce subsekvence ze sekvence $U \upharpoonright s$.
- Ověření, zda sekvence existuje na konci jiné sekvence s suffix t.
- Ověření, zda sekvence existuje na začátku jiné sekvence $s \subseteq t$.
- Ověření zda se sekvence nachází v libovolném místě v jiné sekvenci $s \in t$.
- Počet prvků v množině se určuje znakem $\#$.

Kapitola 5

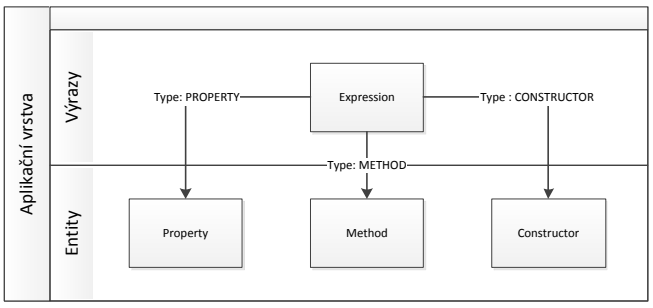
Model rozšířený o chování

Při rozšiřování meta-modelu o entity reprezentující chování na aplikační vrstvě (metody, výrazy, konstruktory), se vycházelo z původního aplikačního meta-modelu definovaného pro The MigDB framework.

Jelikož je však meta-model nyní psaný v jazyce Z, muselo dojít k velkému množství úprav, které si notace žádá. V prvé řadě bylo v meta-modelu nutné nadefinovat vazby mezi jednotlivými entitami speciálními schématy. V okamžiku, kdy se vazby tvořily skrze property, docházelo k validačním chybám kvůli pořadí definovaných entit. Například metoda ve svém těle obsahuje výraz. To znamená, že musí být výraz definovaný dříve než metoda. Problém je ale v tom, že výraz v sobě může obsahovat metodu. Dostáváme se tak do jakéhosi definičního cyklu, který rozsekne až vytvoření speciálního schématu *BODYOFMETHOD* (6.2.6), které je definované později a spojuje metody s jejich tělem.

Vedle notačních úprav meta-model prošel ještě celou řadou drobných logických změn. V původním meta-modelu jsou definované speciální typy atributů. Podle toho, zda jsou primitivního typu, nebo zda jsou součástí obousměrné či jednosměrné vazby. Po úpravě může být v modelu pouze atribut, který je jednosměrným ukazatelem. Neexistují tedy žádné primitivní typy, vše je objekt.

Dále do modelu přibývají nové entity. Základem rozšíření jsou metody, které mají stejně jako v běžných programovacích jazycích parametry, návratovou hodnotu a tělo, které obsahuje sekvenci příkazů.



Obrázek 5.1: Představa dvouvrstvého rozdělení aplikační vrstvy na výrazy a zbylé entity

Aby bylo tělo metody čím naplnit, jsou nově v meta-modelu výrazy, neboli expressions (viz 6.2.8). Pokud se však na strukturu entit v meta-modelu podíváme s nadhledem a velice abstraktně, zjistíme, že výrazy mezi tyto entity nepatří. Výrazy s ostatními entitami nespolupracují, ostatní entity obalují a posléze využívají. Vzniká tak zcela nové rozdělení aplikační vrstvy, které se dělí na entity a výrazy. Výraz ve svém těle může mít uložený atribut třídy, parametr metody, volání konstruktoru nebo celou metodu samotnou. Celé rozložení prvků na entity a výrazy lépe ukazuje obrázek 5.1.

Všechny tyto změny vedly k tomu, aby bylo vytvořeno co nejzákladnější jádro objektového programovacího jazyka podle vzoru modelu Featherweight JAVA [5].

Featherweight JAVA (FJ) je minimalistickým jádrem pro modelování typového systému jazyka JAVA. Jedná se o malý model, který má definovou svou syntax a sémantiku. Jeho hlavním cílem je vytvoření důkazu typové *soundness*, neboli "dobře typovaný program se nikdy nezasekne". Toto tvrzení se dá smysluplně dokázat pouze v okamžiku, kdy je programovací jazyk zredukován o všechny zbytečné prvky až na samotnou "kost". Musí ale platit podmínka, že ono zbylé jádro jazyka musí být schopné vytvořit jakýkoliv konstrukt, který lze napsat ve "velkém"programovacím jazyce. Tedy pro FJ musí platit, že jakýkoliv kód napsaný v JAVA lze napsat i ve FJ a obráceně.

5.1 Formální zápis modelu

Featherwight JAVA pro formální zápis používá vlastní matematickou notaci, která je v textu přehlednější než predikátový zápis jazyka Z. Matematický zápis a notace jazyka Z jsou však ekvivalentní-viz 5.1.3. V této kapitole bude tedy nejprve celý meta-model popsán v matematickém zápise, který díky svému kompaktnějšímu zápisu dovoluje stručně popsat nejzákladnější entity a nejdůležitější typová omezení.

Velkou odlišností zápisu od konvenčních formálních notací je zápis množin. Pokud chceme říci, že je nějaký prvek množinou, zapíšeme ho s čárkou nad $\overline{C}\ \overline{f}$ - Toto například znamená množinu fieldů f, kde má každý svůj typ C.

Pokud se v zápisu objeví velká písmena jako například A, B, C, jedná se o definování typů; f a g definují práci s atributy; m s metodami; x práci skrze proměnné; d a e výrazy.

5.1.1 Syntaxe

Základním stavebním kamenem každého jazyka je jeho syntax. Syntax aplikačního meta-modelu frameworku The MigDB obsahuje definování tříd, atributů, metod, konstruktorů a výrazů. Z velké části definice modelu vychází právě z modelu FJ[5].

$$L ::= class\ C\ extends\ C\ \{\ \overline{C}\ \overline{f};\ K\ \overline{M}\ \} \tag{5.1}$$

L je množina všech tříd. V zápise se říká, že každá třída může mít nějakého předka $extendsC$. Obsahuje množinu atributů $\overline{C}\ \overline{f}$; , jeden konstruktor K a množinu tříd $\overline{M}$.

$$K ::= C(\overline{C}\ \overline{f})\{\ this.\overline{f} = \overline{f};\ \} \tag{5.2}$$

K je množinou všech konstruktorů. Každý konstruktor má stejné jméno jako jeho vlastnící třída C. Dále obsahuje přesně tolik parametrů, v takovém pořadí a s takovými typy, kolik je ve třídě atributů $\overline{C}\ \overline{f}$. Tělo konstruktoru pak už jen obsahuje řádky, které přiřazují parametry k atributům $this.\overline{f} = \overline{f};$.

$$M ::= C\ m(\overline{C}\ \overline{x})\{\ return\ e;\ \} \tag{5.3}$$

M je množinou všech metod v modelu. Každá metoda má svou návratovou hodnotu C, název m, množinu parametrů $\overline{C}\ \overline{x}$ a tělo, které obsahuje pouze jediný řádek s klíčovým slovem *return*.

$$e ::= x \mid e.f \mid e.m(e) \mid newC(\overline{e}) \tag{5.4}$$

e reprezentuje výraz. Ten může nabývat několika různých podob. Buď se jedná o parametr (proměnnou), atribut třídy, volání metody nebo konstruktor.

Všechna zde definovaná syntaktická pravidla musí být nad modelem kontrolována. V každém novém stavu musí být ověřeno, že jsou všechny elementy v předepsaném stavu. Konkrétně to například znamená, že konstruktory mají stejný název jako jejich vlastnící třída. Počet parametrů se rovná počtu atributů a souhlasí i typy. Počet přiřazení v těle konstruktoru souhlasí s počtem parametrů, respektive s počtem atributů (musí být také dokázáno, že parametry a atributy v přiřazení, skutečně patří dané třídě a existují). Omezení pro všechny elementy modelu budou rozebrány později 6.4.

5.1.2 Typová pravidla

U rozšíření meta-modelu o metody jsou největším problémem typové kontroly. Zajištění typové konzistence se stejně jako zajištění správné syntaxe musí řešit při každém novém stavu modelu.

V typových pravidlech jsou z důvodu jednoduššího zápisu použity modelové operace pro práci s entitami. Ve zkratce se jedná o *mBody*() která získá tělo metody; *mType*() pro získání typu metody; *cParent*() získá předka dané třídy; *cAncestors*() vrací všechny předky dané třídy až ke kořenové třídě; *cChilds* vrací množinu všech tříd, které jsou potomky dané třídy; *cProperties*() vrací množinu atributů třídy; *cMethods*() množinu metod dané třídy; *coBody* vrací tělo konstruktoru. Konkrétní rozbor jednotlivých pomocných operací je v kapitole 6.3.

$$\frac{\Gamma\ \vdash\ e_0\ :\ C_0 \qquad cProperties(C_0)\ =\ \overline{C}\ \overline{f}}{\Gamma\ \vdash\ e_0.f_i\ :\ C_i} \tag{5.5}$$

Zápis $\Gamma\ \vdash\ e_0\ :\ C_0$ v překladu znamená "v prostředí Γ má výraz e_0 typ C_0". Prostředí Γ je konečné mapování mezi proměnnými a jejich typy. Dle tohoto pravidla máme výraz typu C_0 a přistoupíme-li k jeho atributu, bude se jednat vždy jen o atribut z třídy C_0, která má sadu atributů $cProperties(C_0)$ a jeho typ bude C_i.

$$\frac{\Gamma\ \vdash\ e_0\ :\ C_0 \qquad mType(m, C_0)\ =\ \overline{D} \to C \qquad \Gamma\ \vdash\ \overline{e_0}\ :\ \overline{C_0} \qquad \overline{C} <: \overline{D}}{\Gamma\ \vdash\ e_0.m(\overline{e})\ :\ C} \tag{5.6}$$

Toto pravidlo definuje typ návratové hodnoty. Pokud budou všechny zadané parametry typově správné $\Gamma \vdash \overline{e_0} : \overline{C_0} \quad \overline{C} <: \overline{D}$ a bude volána skutečně metoda, která se nad daným objektem dá volat $mtype(m, C_0) = \overline{D} \to C$, pak bude typ metodou navráceného objektu C.

$$\frac{cProperties(C) = \overline{D}\ \overline{f} \quad \Gamma \vdash \overline{e} : \overline{C} \quad \overline{C} <: \overline{D}}{\Gamma \vdash new\ C(\overline{e}) : C} \tag{5.7}$$

Poslední typové pravidlo pro výrazy se zabývá konstruktory. V okamžiku, kdy do parametru konstruktoru zadáme výrazy, které vrací správné typy $\Gamma \vdash \overline{e} : \overline{C} \quad \overline{C} <: \overline{D}$, vrátí konstruktor objekt takového typu, jako je jeho vlastnící třída.

Vedle výrazů musí být typová pravidla nastavená i pro metody a třídy.

$$\frac{\overline{x} : \overline{C}, this : C \vdash e_0 : E_0 \quad E_0 <: C_0 \quad mType(n, C) = \overline{C} \to C_0}{C_0\ m(\overline{C}\ \overline{x})\{\ return\ e_0 :\}\ OK\ IN\ C} \tag{5.8}$$

V okamžiku kdy má metoda správné parametry a výraz v jejím těle vrací správný typ, je metoda dobře otypovaná ve třídě C, tedy - $OK\ IN\ C$.

$$\frac{K = C(\overline{D}\ \overline{g},\ \overline{C}\ \overline{f})\{super(\overline{g});\ \ this.\overline{f} = \overline{f};\ \} \quad cProperties(D) = \overline{D}\ \overline{g} \quad \overline{M}\ OK\ IN\ C}{class\ C\ extends\ D\ \{\ \overline{C}\ \overline{f};\ \ K\ \overline{M}\ \}} \tag{5.9}$$

U třídy je z typového hlediska velice důležité pochopit i to, že může mít nějakého rodiče a ten má také své atributy $cProperties(D) = \overline{D}\ \overline{g}$. Pokud má třída dobře definovaného rodiče, správně formulovaný konstrukor a metody, které jsou ve stavu OK, pak je celá třída správně otypovaná.

5.1.3 Mapování do Z

Matematický zápis FJ a notace Z je ekvivalentní. Vše, co lze zapsat v jedné notaci lze zapsat i ve druhé. Například definice třídy. V matematickém zápise vypadá následovně.

$$L ::= class\ C\ extends\ C\ \{\ \overline{C}\ \overline{f};\ K\ \overline{M}\ \}$$

V jazyce Z je tento jediný řádek rozobjektován do několika provázaných schémat. Prvním a tím nejdůležitějším je třída samotná.

CLASS

$label : LABEL$

Každý *label* třídy je ekvivaletní k jeho jménu, tedy C. Možnost dědičnosti, tedy *extendsC* reprezentuje schéma *INHERITANCE*, kde dochází k propojení třídy a jejího rodiče skrze nové vazební schéma.

INHERITANCE
parent : *CLASS*
child : *CLASS*

Množinu atributů $\overline{C}\ \overline{f}$; reprezentuje vazební schéma, které tvoří spojení mezi atributy a třídou v zadaném prostředí.

PROPERTYOFCLASS
class : *CLASS*
property : *PROPERTY*

I konstruktor a metody jsou definovány jako vazba mezi entitami modelu. Tedy zápis $K\ \overline{M}$ se v jazyce Z dá reprezentovat takto.

METHODOFCLASS
class : *CLASS*
method : *METHOD*

CONSTRUCTOROFCLASS
constructor : *CONSTRUCTOR*
class : *CLASS*

Nutno podotknout že tento zápis v jazyce Z není jediný správný. Notace i přes svou velikou typovou konzervativnost dovoluje obrovské vyjadřovací množnosti. Kdyby byla situace v meta-modelu jiná a nebylo by třeba vazebních schémat, zápis by mohl být vložený třeba i do jediné entity reprezentující třídu.

CLASS
label : *LABEL*
fields : $\mathbb{P}$ *PROPERTY*
constructor : *CONSTRUCTOR*
methods : $\mathbb{P}$ *METHOD*

Tento zápis je mnohem přehlednější, ale bohužel v meta-modelu kvůli cyklickým závislostem není realizovatelný.

Kapitola 6

Definování Modelu

6.1 Vlastnosti modelu

Formální meta-model aplikační vrstvy je sestaven s ohledem na co největší abstraktnost a jednoduchost. Veškeré komplexnější prvky jazyka (jako například vlákna), nebyly na úkor přísnější typové bezpečnosti zařazeny do meta-modelu.

Meta-model je jakousi obdobou podmnožiny jazyka JAVA. Základní vlastností a zároveň velkou rozdílností meta-modelu od jazyku JAVA je neexistence primitivních typů. Vše v modelu je objekt.

V rámci objektového modelování jsou zakázané vlastnosti jako zapouzdřování, kovariance, kontravariance, přetěžování metod, přepisování metod, abstrakce, interface, balíčky, enumerace, statické atributy a metody, vnitřní třídy, vícenásobná dědičnost a přetypování.

Pokud bychom se chtěli co nejvíce přiblížit reálné podobě softwarových systémů, je možné jednotlivé vlastnosti do meta-modelu přidat.

6.2 Elementy meta-modelu

Základními elementy meta-modelu jsou třídy s jejich atributy a metodami. Každá metoda pak ve svém těle obsahuje výraz. Jelikož je vše v modelu objekt, tak atributy automaticky vytváří asociace mezi třídami.

6.2.1 Bool

Abychom byli schopni na úrovni meta-modelu nadefinovat některé atributy elementů, je třeba pomocného typu *BOOL*.

$$BOOL ::= TRUE \mid FALSE$$

6.2.2 Označení

Každý element modelu (třída, atribut, metoda) má své unikátní označení, identifikátor. Díky tomuto označení lze elementy rozlišovat a porovnávat. Na úrovni kódu si toto označení spíše než hash, přesně identifikující objekt, můžeme představit jako název daného elementu.

$[LABEL]$

6.2.3 Kardinalita

Díky atributům třídy lze v modelu vytvářet asociace. Je však důležité definovat možné kardinality této asociace. V praxi se běžně setkáváme s kardinalitami definovanými reálným číslem. V rámci zjednodušení modelu jsou však poskytovány pouze dvě možnosti, ONE a $MANY$. Kde ONE představuje vazbu $1:1$ a $MANY$ vazbu $1:N$.

$$CARDINALITY ::= ONE \mid MANY$$

6.2.4 Třída

Schéma $CLASS$ definuje kostru prázdné třídy. Taková třída obsahuje pouze svůj identifikátor, který je typu $LABEL$.

$CLASS$
$label : LABEL$

Model nepodporuje hodnotu *null*. Pokud tedy v modelu nastane například situace, kdy třída nemá žádného předka, vracíme meta-modelovou hodnotu $NULLCLASS$.

$$NULLCLASS : CLASS$$

Vytváření hierarchií je jedním ze základů objektového modelování. V modelu je proto velice důležité udržet si informaci o dědičných vazbách mezi třídami.

Meta-model k obdobným situacím poskytuje speciální vazební schémata, kterými pak na úrovni modelu drží spojení jednotlivých elementů. Každé vazební schéma si vždy pamatuje danou n-tici prvků, čímž mezi nimi vytvoří pomyslnou vazbu (nemusí se nutně jednat o vazbu hierarchickou).

Dědičnost je v modelu znázorněna pomocí dvojice rodič a potomek, která je uložena ve schématu $INHERITANCE$. Jelikož v modelu není povolena vícenásobná dědičnost, schéma obsahuje pouze dvojici rodič *parent* a potomek *child*.

$INHERITANCE$
$parent : CLASS$
$child : CLASS$

6.2.5 Atribut

Model je zcela objektový. Každý atribut třídy se tak automaticky stává asociací s jinou třídou. Proto je u každého atributu nutné zároveň nastavit i jeho kardinalitu *upper* (viz 6.2.3). Unikátnost názvu atributu v rámci třídy a hierarchie je kontrolována za pomoci identifikátoru *LABEL*. Cílová (*target*) třída určuje typ atributu.

Asociace v modelu jsou pouze jednosměrné. Tedy znalost asociace si drží pouze třída, vlastnící daný atribut.

PROPERTY
label : *LABEL*
upper : *CARDINALITY*
target : *CLASS*

Vazbu mezi třídou a atributem drží schéma *PROPERTYOFCLASS*.

PROPERTYOFCLASS
class : *CLASS*
property : *PROPERTY*

6.2.6 Metoda

Metody tříd jsou definované jako schémata, která si drží množinu parametrů *params* a návratovou hodnotu *return*. Každá metoda má také svůj jedinečný identifikátor *label*, díky kterému můžeme například zajistit zákaz přepisování a přetěžování metod v rámci třídy a hierarchie.

METHOD
label : *LABEL*
params : $\mathbb{P}$ *PROPERTY*
return : *CLASS*

V modelu je u návratové hodnoty metody zakázané vracet typ *void*. Metoda musí vždy obsahovat návratovou hodnotu nějakého existujícího typu. Jinak řečeno metoda ve svém těle v jazyce JAVA musí vždy obsahovat klíčové slovo *return*.

Vazbu mezi metodou a třídou znázorňuje schéma *METHODOFCLASS*, kde *class* je vlastnící třída a *method* je konkrétní metoda.

METHODOFCLASS
class : *CLASS*
method : *METHOD*

V těle metody se nachází složený výraz reprezentující logiku a chování metody (Expression - viz 6.2.8). Vazbu mezi tímto složeným výrazem a metodou definuje schéma *BODYOFMETHOD*.

Složeným výrazem je v meta-modelu myšlena sekvence v řádku po sobě jdoucích příkazů. V objektově orientovaných jazycích jde o volání skrze "tečkovou"notaci.

BODYOFMETHOD
method : *METHOD*
body : seq *EXPRESSION*

6.2.7 Konstruktor

Konstruktor je speciální typ metody, který má vždy stejný název *label*, jako jeho vlastnící třída. Jelikož je v meta-modelu zakázané přetěžování, v každé třídě je povolen pouze jeden konstruktor.

Každý konstruktor má právě tolik parametrů *params* a právě takového typu, jako jsou atributy třídy. Typ konstruktoru *type* je vždy typem třídy, která ho vlastní.

Tělo konstruktoru je jediné místo v modelu, kde je ve výrazu povoleno přiřazení (vlastnosti výrazů 6.2.8). Tělo obsahuje právě tolik přiřazení, kolik má konstruktor parametrů.

CONSTRUCTOR
label : *LABEL*
params : $\mathbb{P}$ *PROPERTY*
type : *CLASS*

Každé přižazení, tedy každý řádek konstruktoru, představuje právě jedno schéma *CONSTRUCTORSBODY*.

CONSTRUCTORSBODY
constructor : *CONSTRUCTOR*
assigned : *ASSIGNEDEXPR*

V modelu má každá třída právě jeden konstruktor. Tuto vazbu znázorňuje schéma *CONSTRUCTOROFCLASS*, které obsahuje vlastnící třídu a k ní přiřazený konstruktor.

CONSTRUCTOROFCLASS
constructor : *CONSTRUCTOR*
class : *CLASS*

6.2.8 Výraz

Tělo každé metody může obsahovat pouze jeden jediný složený výraz, který je v modelu reprezentovaný sekvencí elementárních výrazů. V jazkyce JAVA si tuto situaci můžeme představit tak, že každá metoda obsahuje pouze jeden řádek kódu začínající klíčovým slovem *return* a všechny elementární výrazy jsou objektově volané skrze tečkovou notaci.

```
return new Human(name, this.age).setName(new String("P"));
```

Výrazy mohou nabýt čtyř forem - atribut třídy, parametr metody, volání metody nebo konstruktor vytvářející objekt.

Každý výraz má svůj typ *out*. Typem je myšlena třída, kterou daný výraz vrací jako výsledek svého zpracování. V sekvenci po sobě jdoucích elementárních výrazů si pak můžeme dovolit tvrdit, že typ celého složeného výrazu definuje jeho poslední člen.

Všechny typy výrazů jsou v meta-modelu reprezentovány jedním elementem *EXPRESSION*, který zcela zobecňuje práci se skutečnými elementy modelu. Zobecnění spočívá v tom, že všechny čtyři elementární typy výrazů považujeme za volání, které má vstupní parametry *in*, výstup *out* a typ *caller*, nad kterým mohou být volány.

V případě metod a konstruktorů se nejedná o složitou představu. Oba elementy mají parametry a výstupní hodnotu. Metody mají hodnotu *return* a konstruktory mají typ třídy, kterou sami vytváří *type*. V případě atributů třídy a parametrů je vstupní sekvence parametrů prázdná a výstupní hodnotu určuje typ *PROPERTY*.

EXPRESSION

$out : CLASS$
$caller : CLASS$
$type : (PROPERTY \lor CONSTRUCTOR \lor METHOD)$

Aby nedocházelo k cyklickým chybám v definování schémat je nutné *in* sekvenci všech vstupních výrazů, tedy parametrů výrazu, uložit do vazebního schématu *INSEQOFEXPR*.

INSEQOFEXPR

$in : \text{seq}\, EXPRESSION$
$e : EXPRESSION$

Atribut *type* značí, jaký element v modelu daný výraz představuje. Může se jednat o konstruktor, metodu nebo atribut či parametr (obojí je typu *PROPERTY*). Tuto skutečnost v jazyce Z zachycuje zápis $(PROPERTY \lor CONSTRUCTOR \lor METHOD)$, kde $\lor$ znamená právě ono "nebo". Kdybych se snažil vytvořit podobný konstrukt v objektově orientovaném jazyce, využil bych dědičnosti a vytvořil tři typy výrazů mající jednoho předka. Dědičnost bohužel jazyk Z nepodporuje. Tento zápis sice ano, ale neumí s ním pracovat tak, jak je zamýšleno. Jazyk Z v tuto chvíli očekává, že všechny prvky, uložené v *type* budou právě typu $(PROPERTY \lor CONSTRUCTOR \lor METHOD)$, ne však pouze jedním z nich. Toto je tedy pouze velice kompromisní řešení, které občas validátor ve výjovovém prostředí označí jako syntaktickou chybu.

Každý výraz v modelu musí mít určeno, nad čím smí být volán - *caller*. Například metody nebo atributy mohou být volány pouze nad objekty třídy, která je vlastní.

Pro představu se v následujícím výrazu snažíme z listové struktury všech lidí získat člověka, jehož jméno vygeneruje generátor. Generátor k vytvoření jména potřebuje vědět první písmeno a maximální délku jména. V tomto případě je prvním písmenem P a maximální délka je pět.

```
return listOfHumans.getHumanByName(
                        nameGenerator(
                                new String("P"),
                                new Integer(5)));
```

Objekt *listOfHumans* je typu *HumanList*. Veřejná metoda poskytovaná rozhraním třídy *HumanList* je *getHumanByName*(). Tato metoda vrací objekt typu *Human* a jako parametr má jméno, které musí být typu *String*. Druhou metodou, která se v příkladu vyskytuje, je metoda *nameGenerator*(), která vrací objekt typu *String* a jejími parametry jsou počáteční písmeno jména, které musí být typu *String* a maximální délka jména, která musí být uložena v podobě objektu typu *Integer*.

Pro metodu *getHumanByName*() je typ calleru *List*, jeho výstupem je pak typ *Human*. Metoda *nameGenerator*() má v calleru uložený typ třídy, která ji vlastní a jako výstup generuje objekt typu *String*.

6.2.8.1 Variable expresion

Pokud se v sekvenci složeného výrazu využije parametr metody, jde o variable expression a v kolonce *type* entity bude typ *PROPERTY*.

V reálném kódu jde například o nastavovací metody (*setters*), které zajišťují přístup k privátním třídním proměnným a dovolují změnu jejich hodnot. V kontextu meta-modelu však setter nemůže fungovat stejně jako například v jazyce JAVA. Každá metoda v modelu má pouze jediný řádek a ten začíná klíčovým slovem *return*, každá metoda tedy musí vracet objekt nějakého typu.

Řešením pro metody s obdobnou funkcí jako jsou settery je vracet nový objekt, který kopíruje ten původní a změní jen požadovaný atribut.

```
Human setName(String name){
        return new Human(name, this.age);
}
```

V příkladu je nastaveno nové jméno člověka (třída *Human*). Každý člověk v programu si s sebou nese atributy jméno (*name*) a věk (*age*). Použití atributu *name* v konstruktoru nového objektu *Human* je variable expression.

6.2.8.2 Field expression

Pokud se v sekvenci složeného výrazu využije atribut třídy, jde o field expression a v kolonce *type* entity bude stejně jako u variable expression *PROPERTY*.

Využití atributu a parametru ve složeném výrazu je z hlediska modelu totožná věc. V obou případech je hodnotou *out* typ objektu.

V kódu si lze představit stejnou situaci jako u variable expression. Field expression je tu zastoupena použitím *this.age*.

```
Human setName(String name){
        return new Human(name, this.age);
}
```

Field expression se v kódu obecně pozná podle výskytu klíčového slova *this*. Existují však i situace, kdy přistupujeme k atributu třídy bez použití *this* (viz 6.2.8.2).

```
new Human(first, second).name
```

6.2.8.3 Call expression

Ve složeném výrazu můžeme volat metody nad atributy třídy nebo parametry. V takovém případě bude u expression typ *METHOD*.

V následujícím příkladu se snažíme objektu *Human* nastavit nové jméno. Metoda *setName*() reprezentuje volání call expression.

```
new Human(name, this.age).setName(new String("Petr"))
```

6.2.8.4 Create expression

Pokud ve složeném výrazu dochází k vytváření objektů, typem expression bude *CONSTRUCTOR*.

Jelikož je meta-model definovaný zcela objektově, vytváření objektů je ve výrazech zcela běžnou praxí. Vytvoření instance *Human* je create expression stejně jako vytvoření objektu *String* a *Integer*.

```
new Human(new String("Peter"),new Integer(15))
```

6.2.8.5 Assigned expression

Jedná se o specální typ výrazu, který se vyskytuje pouze v konstruktorech. Nikdy není součástí sekvence složeného výrazu.

Jediným účelem přiřazovacího výrazu je přiřazení hodnoty parametru *param* k atributu *field* v těle konstrukoru třídy.

ASSIGNEDEXPR
param : *PROPERTY*
field : *PROPERTY*

V kódu se jedná o jeden řádek konstruktoru, kde se vyskytuje přiřazení hodnoty parametru *param* k danému atributu *field*.

```
this.name = name;
```

6.2.8.6 Složený výraz

V těle metody se tedy může vyskytnout buď kombinace elementárních výrazů nebo každý samostatně.

Model výrazů je oproti reálným programovacím jazykům velice omezen. V metodách nelze přiřazovat, inicializovat, používat aritmetické nebo logické operátory, vytvářet smyčky

nebo rozhodovací stromy. Na druhou stranu model nemá žádný problém při práci s rekurzí a metodami. Pokud tedy například chceme sečíst dvě čísla, můžeme vše řešit skrze třídy a metody.

Důležitou součástí složeného výrazu, neboli sekvence výrazů, je možnost zanořování. Každá entita *EXPRESSION* v sobě obsahuje množinu parametrů *in*. Tyto parametry jsou také typu *EXPRESSION*. To nám dovoluje vytvářet libovolně zanořené struktury výrazů skrze parametry.

Díky zanořování skrze parametry si můžeme dovolit psát výrazy jako v následujícím příkladu.

```
new Human(name, this.age).setName(
                        generateName(
                                new String("P"),
                                new Integer(5)));
```

V tomto výrazu se vyskytuje vytvoření objektu (create expression), použití parametru (variable expression), použití atributu třídy (field expression) a použití volání metody (call expression). Zároveň dochází k zanořování výrazů skrze parametry metody.

6.2.9 Aplikace

Aplikace je v prvé řadě definována jako množina tříd, atributů, metod a konstruktorů. Dále obsahuje množiny vazeb mezi elementy, tedy množinu dědičností, těl metod, těl konstruktorů a výrazů.

U každé aplikace je definován její typ, což je hodnota určující, zda se jedná o aplikaci validní *CONSISTENTSTATE*, či nevalidní *FAILSTATE*.

$$STATE ::= CONSISTENTSTATE \mid FAILSTATE$$

APPLICATION

$classes : \mathbb{P}\ CLASS$
$properties : \mathbb{P}\ PROPERTY$
$proprtiesOfClasses : \mathbb{P}\ PROPERTYOFCLASS$
$methods : \mathbb{P}\ METHOD$
$methodsOfClasses : \mathbb{P}\ METHODOFCLASS$
$constructors : \mathbb{P}\ CONSTRUCTOR$
$constructorsOfClasses : \mathbb{P}\ CONSTRUCTOROFCLASS$
$constructorsBodies : \mathbb{P}\ CONSTRUCTORSBODY$
$inheritances : \mathbb{P}\ INHERITANCE$
$bodyOfMethods : \mathbb{P}\ BODYOFMETHOD$
$expressions : \mathbb{P}\ EXPRESSION$
$inSeqOfExpressions : \mathbb{P}\ INSEQOFEXPR$
$assignedExpressions : \mathbb{P}\ ASSIGNEDEXPR$
$type : STATE$

6.2.10 Typ metody

V meta-modelu je nadefinováno i speciální schéma *METHODTYPE*, které reprezentuje datový typ metody jako množinu typů parametrů (*params*) a typ návratové hodnoty (*return*).

METHODTYPE

$$params : \mathbb{P}\, CLASS$$
$$return : CLASS$$

6.3 Modelové operace

V meta-modelu je definováno několik operací, které se dále využívají během zajištění konzistentnosti modelu a poskytují veliký komfort při práci.

6.3.0.1 mBody

Operace mBody nám slouží k získání sekvence *EXPRESSION*, která se vyskytuje v těle metody. Tato operace je tedy funkcí, která má na vstupu danou metodu a vlastnící třídu a vrací složený výraz uložený v těle, tedy $(METHOD \times CLASS) \rightarrow seq\ EXPRESSION$.

$$mbody\ : (METHOD \times CLASS) \rightarrow \text{seq}\, EXPRESSION$$

$$\begin{aligned}
&\forall\, m : METHOD;\ c : CLASS;\ exp : \text{seq}\, EXPRESSION \bullet \\
&mbody\ (m, c) = exp \Rightarrow \\
&\quad \exists\, bom : BODYOFMETHOD;\ moc : METHODOFCLASS \bullet \\
&\quad moc.class = c \land moc.method = m \land \\
&\quad bom.method = m \land bom.body = exp
\end{aligned}$$

6.3.0.2 mType

Meta-modelová operace *mType* nám slouží k získání datového typu metody. Operace má na vstupu danou metodu a vlastnící třídu a vrací typ *METHODTYPE*, tedy $(METHOD \times CLASS) \rightarrow METHODTYPE$.

$$mtype\ : (METHOD \times CLASS) \rightarrow METHODTYPE$$

$$\begin{aligned}
&\forall\, m : METHOD;\ c : CLASS;\ type : METHODTYPE;\ a : APPLICATION \bullet \\
&mtype\ (m, c) = type \Rightarrow \\
&\quad m.return = type.return \land \quad (\forall\, p : PROPERTY \bullet \\
&\quad p \in m.params \Rightarrow \\
&\qquad p.target \in type.params)
\end{aligned}$$

6.3.0.3 cParent

Operace, která vrátí předka dané třídy. Jedná se tedy o funkci, kde je vstupem třída a celá aplikace a výstupem je nadřazená třída, tedy $(CLASS \times APPLICATION) \rightarrow CLASS$.

$$
\begin{array}{|l}
cparent\ : (CLASS \times APPLICATION) \rightarrow CLASS \\
\hline
\forall\, c, par : CLASS;\ a : APPLICATION \bullet \\
cparent\ (c, a) = par \Leftrightarrow \\
\quad \exists\, i : INHERITANCE \bullet \\
\quad i.parent = par \land i.child = c \land cparent\ (c, a) = NULLCLASS \Leftrightarrow \\
\quad\quad \forall\, i : INHERITANCE \bullet \\
\quad\quad i.parent = par \land i.child \neq c
\end{array}
$$

6.3.0.4 cAncestors

Meta-modelová operace pro získání všech předků třídy. Vstupem této operaci je třída a aplikace a výstupem je sekvence tříd, tedy $(CLASS \times APPLICATION) \rightarrow \text{seq}\, CLASS$.

$$
\begin{array}{|l}
cancestors\ : (CLASS \times APPLICATION) \rightarrow \text{seq}\, CLASS \\
\hline
\forall\, c : CLASS;\ a : APPLICATION;\ out : \text{seq}\, CLASS \bullet \\
cancestors\ (c, a) = \langle\rangle \Leftrightarrow cparent\ (c, a) = NULLCLASS \land \\
cancestors\ (c, a) = \langle cparent\ (c, a)\rangle \frown cancestors\ (cparent\ (c, a), a) \Leftrightarrow \\
\quad cparent\ (c, a) \neq NULLCLASS
\end{array}
$$

6.3.0.5 cChilds

Pokud je třeba získat všechny potomky dané třídy, slouží k tomu operace *cchilds*. Vstupem této operace je třída a aplikace. Výstupem pak množina všech potomků, tedy $(CLASS \times APPLICATION \rightarrow \mathbb{P}\, CLASS)$.

$$
\begin{array}{|l}
cchilds\ : CLASS \times APPLICATION \rightarrow \mathbb{P}\, CLASS \\
\hline
\forall\, c_p : CLASS;\ a : APPLICATION \bullet \\
cchilds(c_p, a) = \{c : CLASS \mid \\
\quad \exists\, i : INHERITANCE \bullet i.parent = c_p \land i.child = c \land i \in a.inheritances\}
\end{array}
$$

6.3.0.6 cProperties

Operace pro získání všech atributů třídy. Vstupem je znovu třída a aplikace a výstupem množina atributů, tedy $(CLASS \times APPLICATION) \rightarrow \mathbb{P}\, PROPERTY$.

$$
\begin{array}{|l}
cproperties\ : (CLASS \times APPLICATION) \rightarrow \mathbb{P}\, PROPERTY \\
\hline
\forall\, c : CLASS;\ a : APPLICATION \bullet \\
cproperties\ (c, a) = \{p : PROPERTY \mid p \in a.properties \land \\
\quad \exists\, poc : PROPERTYOFCLASS \bullet poc.class = c \land poc.property = p\}
\end{array}
$$

6.3.0.7 cMethods

Aby byla sada operací kompletní, je v meta-modelu operace cMethod, která získá všechny metody zadané třídy. Vstupem je třída a aplikace a výstupem množina metod, tedy $(CLASS \times APPLICATION) \rightarrow \mathbb{P}\, METHOD$.

$$cmethods : (CLASS \times APPLICATION) \rightarrow \mathbb{P}\, METHOD$$

$$\begin{aligned}
&\forall\, c : CLASS;\ a : APPLICATION \bullet \\
&cmethods\ (c, a) = \{m : METHOD \mid m \in a.methods \wedge \exists\, met : METHODOFCLASS \bullet \\
&\quad met \in a.methodsOfClasses \wedge met.class = c \wedge met.method = m\}
\end{aligned}$$

6.3.1 coBody

Jednu pomocnou operaci mají i konstruktory. Operace coBody získá všechna přiřazení, která se v těle metody vyskytují. Jedná se tedy o operaci, která má na vstupu konstruktor a na výstupu množinu všech přiřazení, tedy $(CONSTRUCTOR) \rightarrow \mathbb{P}\, ASSIGNEDEXPR$.

$$cobody : (CONSTRUCTOR \times APPLICATION) \rightarrow \mathbb{P}\, ASSIGNEDEXPR$$

$$\begin{aligned}
&\forall\, co : CONSTRUCTOR;\ a : APPLICATION \bullet \\
&cobody\ (co, a) = \{as : ASSIGNEDEXPR \mid as \in a.assignedExpressions \wedge \\
&\quad \exists\, cb : CONSTRUCTORSBODY \bullet cb.constructor = co \wedge cb.assigned = as\}
\end{aligned}$$

6.4 Aplikační invarinaty

Pro zajištění konzistence modelu po a před transformací musí být definována omezení, která model musí splňovat. Oproti statickému modelu jsou v dynamickém modelu nejzásadnější rozšíření v oblasti metod a výrazů.

Ve výrazu, kde dochází k vytváření objektu nějaké třídy, musí být zajištěno, že konstruktor, který daný objekt sestavuje v aplikaci, skutečně existuje. To znamená, že konstruktor musí být součástí množiny všech konstruktorů aplikace (*constructors*).

ConstructorInExpressionMustExistsInv

$$\begin{aligned}
&a? : APPLICATION \\
&\hline
&\forall\, e : EXPRESSION;\ c : CONSTRUCTOR \bullet \\
&e.type = c \Rightarrow c \in a?.constructors
\end{aligned}$$

Modelová situace v JAVA může vypadat následovně. V metodě *foo*() se vytváří objekt typu *A*. Musí tedy být ověřeno, že konstruktor pro třídu *A* skutečně existuje.

```
class A {}
class B {
        A foo() {
                return new A();
        }
}
```

Pokud se ve výrazu vyskytuje nějaký třídní atribut, musí být zajištěno, že tento atribut skutečně existuje buď ve třídě, ve které se vyskytuje metoda, nebo ve třídách předků.

FieldOrVariableInExpressionMustExistsInv

$a? : APPLICATION$

$\forall e : EXPRESSION;\ m : METHOD;\ c : CLASS;$
$anc : \text{seq}\ CLASS;\ p : PROPERTY \bullet$
$e.type = p \land e \in \text{ran}(mbody\ (m, c)) \land anc = cancestors\ (c, a?) \Rightarrow$
$(\exists poc : PROPERTYOFCLASS \bullet$
$p = poc.property \land (poc.class \in \text{ran}(anc) \lor c = poc.class))$
$\lor (p \in m.params)$

Pokud jde o případ, kdy se ve výrazu vyskytuje atribut třídy, jde pro bližší představu o *this.c* v příkladu níže. Aby byl konzistentní, musí být zajištěno, že atribut c skutečně existuje.

```
class B{
        C c;
        A foo() {
                return new A(this.c);
        }
}
```

V příkladu níže je v metodě *getD*() využíván parametr metody *foo*(). V rámci zachování konzistence modelu musí platit, že parametr d skutečně existuje.

```
class B{
        C c;
        A foo(D d) {
                return c.getD(d);
        }
}
```

V případě, že je ve výrazu volána nějaká metoda nad objektem, musí platit, že metoda ve třídě objetu skutečně existuje.

MethodInExpressionMustExistInv

$a? : APPLICATION$

$\forall e : EXPRESSION;\ m : METHOD;\ tar : CLASS;$
$anc : \text{seq}\ CLASS;\ p : PROPERTY \bullet$
$e.type = m \Rightarrow$
$\exists moc : METHODOFCLASS \bullet$
$m = moc.method \land anc = cancestors\ (tar, a?) \land$
$(tar = moc.class \lor moc.class \in \text{ran}(anc))$

Situace může vypadat například takto. V metodě *foo*() je nad objektem c zavolána metoda *createA*(). Musí být tedy zajištěno, že metoda *createA*() skutečně existuje.

```
class B{
        C c;
```

```
        A foo() {
                return c.createA();
        }
}
```

V těle konstruktoru, tedy v assigned expression, je nutné ověřit, že přiřazovaný parametr skutečně v hlavičce konstruktoru existuje.

ParameterInAssignedExpressionMustExistsInv

$a? : APPLICATION$

$\forall e : ASSIGNEDEXPR;\ c : CONSTRUCTOR;\ param : PROPERTY \bullet$
$param = e.param \Rightarrow e.param \in c.params$

V příkladu níže se tedy jedná o to, aby v konstruktoru skutečně existoval parametr *name*, který na pravé straně výrazu přiřazujeme k atributu *this.name*.

```
Human (String name) {
        this.name = name;
}
```

V modelu je třeba ověřit i opačnou situaci, tedy že atribut, ke kterému přiřazujeme parametr, skutečně ve třídě existuje.

FieldInAssignedExpressionMustExistsInv

$a? : APPLICATION$

$\forall e : ASSIGNEDEXPR;\ co : CONSTRUCTOR;\ c : CLASS;\ p : PROPERTY \bullet$
$p = e.field \Rightarrow$
$\quad \exists cb : CONSTRUCTORSBODY \bullet$
$\quad co = cb.constructor \land e = cb.assigned \land$
$\quad\quad \exists coc : CONSTRUCTOROFCLASS \bullet$
$\quad\quad co = coc.constructor \land c = coc.class \land$
$\quad\quad\quad \exists poc : PROPERTYOFCLASS \bullet$
$\quad\quad\quad p = poc.property \land c = poc.class$

Aby v modelu nevznikaly osamělé, tedy nikým nevlastněné metody, musí platit pravidlo, že každá metoda v modelu je vlastněna právě jednou třídou.

MethosAreOwnedByClassInv

$a? : APPLICATION$

$\forall m : METHOD \bullet$
$m \in a?.methods \Leftrightarrow$
$\quad \exists moc : METHODOFCLASS \bullet$
$\quad moc \in a?.methodsOfClasses \land m = moc.method$

Model se dostane do nekonzistentního stavu nejen kvůli osamělým metodám, ale i kvůli nikým nevlastněným konstruktorům. Každý konstruktor musí mít právě jednu třídu, která ho vlastní.

ConstructorsAreOwnedByClassInv

$a? : APPLICATION$

$\forall\, c : CONSTRUCTOR \bullet$
$c \in a?.constructors \Leftrightarrow$
$\quad \exists\, coc : CONSTRUCTOROFCLASS \bullet$
$\quad coc \in a?.constructorsOfClasses \land c = coc.constructor$

V případě konstruktorů platí ještě jedno pravidlo. Každá třída smí mít pouze jeden konstruktor. V modelu to tedy znamená povolení pouze jedné vazby *CONSTRUCTOROFCLASS* mezi třídou a konstruktory.

ClassHaveOneConstructorInv

$a? : APPLICATION$

$\forall\, coc1, coc2 : CONSTRUCTOROFCLASS \bullet$
$coc1 \in a?.constructorsOfClasses \land coc2 \in a?.constructorsOfClasses \Rightarrow$
$\quad coc1.class \neq coc2.class$

Stejné pravidlo jako u konstruktorů musí platit i pro všechny výrazy v modelu. Každý výraz musí být součástí právě jedné sekvence výrazů uložené v nějakém těle metody.

Existuje však výjimka. Jak již bylo řečeno, výraz vytváří abstraktní vrstvu nad elementy modelu. Pokud tedy někde chceme použít metodu $a(Cc, Dd)$, musíme vytvořit výraz, který sice bude s touto metodou svázaný prostřednictvým *type*, ale ve skutečnosti převezme její vlastnosti a bude simulovat její použití.

Převzetí vlastností znamená, že podle počtu parametrů v metodě vznikne počet prvků v *in* sekvenci výrazů. Podle návratové hodnoty metody se nastaví *out* a podle třídy, která metodu vlastní, se nastaví *caller*.

U metody $a(Cc, Dd)$ se pak vytvoří dva prvky typu *EXPRESSION* v sekvenci *in*. Do těchto prvků se může vložit další výraz, do jehož parametru se stejným způsobem může vložit další atd. Tím vniká parametrické větvení výrazů.

Zde už ale narážíme na ty výrazy, které nutně nemusí být uložené v nějaké sekvenci těla metody. Jsou to právě ty *EXPRESSION*, které simulují vstupní výrazy v sekvenci *in*. Tedy například výraz, který naplní v metodě $a()$ parametr c nebo d. Zaručení, zda výstup tohoto podvýrazu odpovídá typu parametru metody, se řeší až později (6.4.1)).

ExpressionIsOwnedByBodyOfMethodInv

$a? : APPLICATION$

$\forall\, e : EXPRESSION \bullet$
$e \in a?.expressions \Leftrightarrow$
$\quad (\exists\, bom : BODYOFMETHOD \bullet$
$\quad bom \in a?.bodyOfMethods \land e \in \mathrm{ran}(bom.body))$
$\quad \lor\, (\exists\, expr : EXPRESSION;\ es : INSEQOFEXPR \bullet$
$\quad\quad es.e = expr \land e \in \mathrm{ran}(es.in))$

U všech metod v modelu musí platit, že typy parametrů v modelu skutečně existují. To znamená, že všechny třídy zastupující typy parametrů jsou v množině tříd aplikace (*classes*).

TypeOfMethosParametersMustExistsInv

$$a? : APPLICATION$$

$$\forall\, m : METHOD;\ p : PROPERTY \bullet$$
$$p \in m.params \Rightarrow p.target \in a?.classes$$

U parametrů metod je také nutné zajistit unikátnost jejich názvů. V hlavičce metody, tedy v množině *params*, nesmí být dva parametry se stejným názvem.

MethodParametersMustHaveUniqueNamesInv

$$a? : APPLICATION$$

$$\forall\, m : METHOD;\ p_1, p_2 : PROPERTY \bullet$$
$$p_1 \in m.params \land p_2 \in m.params \land p_1 \neq p_2 \Rightarrow p_1.label \neq p_2.label$$

Stejné typové omezení jako u parametrů platí i s návratovou hodnotou metody. Třída, reprezentující typ návratové hodnoty, musí být uložena v množině všech tříd aplikace (*classes*).

ReturnTypeOfMethodMustExistsInv

$$a? : APPLICATION$$

$$\forall\, m : METHOD;\ c : CLASS \bullet$$
$$c = m.return \Rightarrow c \in a?.classes$$

V modelu je zakázané přetěžování metod, a to jak v rámci třídy, tak v rámci hierarchie. Musí tedy platit maximální unikátnost všech metod v hierarchii. Toto pravidlo je zajištěno vyžadováním unikátního jména, a to jak v rámci třídy, tak v rámci hierarchie.

NoMethodOverloadingInClassInv

$$a? : APPLICATION$$

$$\forall\, m_1, m_2 : METHODOFCLASS \bullet$$
$$m_1 \in a?.methodsOfClasses \land m_2 \in a?.methodsOfClasses \land$$
$$m_1.class = m_2.class \land m_1 \neq m_2 \Rightarrow m_1.method.label \neq m_2.method.label$$

NoMethodOverloadingInHierarchyInv

$$a? : APPLICATION$$

$$\forall\, c_1, c_2 : CLASS;\ anc : \mathrm{seq}\, CLASS;\ m_1 : METHOD \bullet$$
$$anc = cancestors\ (c_1, a?) \land c_2 \in \mathrm{ran}(anc) \land$$
$$c_1 \in a?.classes \land m_1 \in cmethods\ (c_1, a?) \Rightarrow$$
$$m_1 \notin cmethods\ (c_2, a?)$$

V modelu je zakázána i další vlastnost objektového programování, a to přepisování metod v rámci hierarchie. S tímto omezením jde ruku v ruce také zákaz kovariance a kontravariance. Všechny tyto vlastnosti lze kontrolovat skrze unikátnost názvů metod.

NoOverridingOfMethodsInv

$$
\begin{array}{l}
a? : APPLICATION \\
\hline
\forall\, c_1, c_2 : CLASS;\ anc : \mathrm{seq}\, CLASS;\ m_1 : METHOD \bullet \\
anc = cancestors\ (c_1, a?) \land c_2 \in \mathrm{ran}(anc) \land \\
c_1 \in a?.classes \land m_1 \in cmethods\ (c_1, a?) \Rightarrow \\
\quad m_1 \notin cmethods\ (c_2, a?)
\end{array}
$$

Svá omezení v modelu mají i konstruktory tříd. První nutnou vlastností konstruktoru je shodný název s vlastnící třídou.

NamesOfConstructorAndOwnedClassAreEqualInv

$$
\begin{array}{l}
a? : APPLICATION \\
\hline
\forall\, c : CLASS;\ coc : CONSTRUCTOROFCLASS \bullet \\
coc.class = c \Rightarrow coc.constructor.label = c.label
\end{array}
$$

Dalším omezením u konstruktorů je vyžadování stejného počtu parametrů, jako je počet atributů třídy. Parametry a atribudy se dále musí shodovat svým názvem a typem.

ConstructorParamsAndClassPropertiesAreEqualsInv

$$
\begin{array}{l}
a? : APPLICATION \\
\hline
\forall\, c : CLASS;\ properties : \mathbb{P}\, PROPERTY \bullet \\
properties = cproperties(c, a?) \Rightarrow \\
\quad \exists\, coc : CONSTRUCTOROFCLASS \bullet \\
\quad c = coc.class \land properties = coc.constructor.params \Rightarrow \\
\quad\quad \forall\, p_1, p_2 : PROPERTY \bullet \\
\quad\quad p_1 \in properties \land p_2 \in coc.constructor.params \land p_1.label = p_2.label \Rightarrow \\
\quad\quad\quad p_1.target = p_2.target
\end{array}
$$

Každý konstruktor má svůj typ a ten musí odpovídat třídě, která ho vlastní. Toto omezení zabraňuje vytvoření situací, kdy by v modelu byl vytvořen konstruktor, který by vracel objekty jiného typu, než je jeho vlastnící třída.

ConstructorTypeAndOwnedClassAreEqualsInv

$$
\begin{array}{l}
a? : APPLICATION \\
\hline
\forall\, c : CLASS;\ co : CONSTRUCTOR \bullet \\
c = co.type \Rightarrow \\
\quad \exists\, coc : CONSTRUCTOROFCLASS \bullet \\
\quad coc.class = c \land coc.constructor = co
\end{array}
$$

Posledním omezením pro konstruktory je ověření, zda se počet přiřazení v těle konstruktoru rovná počtu parametrů konstrukotru (z předešlých omezení tedy i počtu atributů třídy).

NumbeOfExpressionInConstructorIsValidInv

$a? : APPLICATION$

$\forall\, co : CONSTRUCTOR \bullet$
$\#(cobody(co, a?)) = \#(co.params)$

6.4.1 Typová kontrola

Všechna omezení se až do tohoto okamžiku týkala převážně udržení modelu v konzistentním stavu. Ověřovalo se, zda skutečně v modelu existují všechny elementy, které figurují v daných výrazech, třídách, konstruktorech nebo atributech.

Typová kontrola je samozřejmě také součástí udržení konzistence. Jedná se však o trochu odlišný přístup k problému. Místo toho, aby se ověřovala elementární proveditelnost, ověřuje se proveditelnost typová. Jinými slovy, zda je model nejen z vlastní struktury proveditelný a odpovídá meta-modelu, ale také zda dodržuje typová pravidla.

V případě dynamického modelu je nejdůležitější částí typové konzistence ověřování výrazů a metod. Každý výraz v modelu musí mít podle meta-modelu definovaný svůj typ. V typové kontrole musíme ověřit, že se deklarovaný typ výrazu skutečně rovná typu, který daný výraz vrací. U metod je důležité ověřit, zda výraz, který je uložen v jejich těle, koresponduje s návratovou hodnotou.

Typ výrazu je vždy roven typu posledního prvku v sekvenci výrazů. Pokud tedy chceme ověřit, zda je výraz v těle metody roven návratové hodnotě, musíme se zaměřit na poslední prvek v sekvenci.

K nerovnosti typů může dojít pouze v případě, že typ výrazu je podtypem typu návratové hodnoty.

ReturnTypeAndExpressionTypeAreEqualInv

$a? : APPLICATION$

$\forall\, m : METHOD;\ rType : CLASS;\ childs : \mathbb{P}\, CLASS \bullet$
$rType = m.return \Rightarrow$
$\quad \exists\, bom : BODYOFMETHOD \bullet$
$\quad bom.method = m \land childs = cchilds\ (rType, a?) \land$
$\quad ((last\ (bom.body)).out = rType \lor (last\ (bom.body)).out \in childs)$

Invariant pro existenci typu parametru a návratové hodnoty si představíme na dalším příkladu. U metody *foo*() musí být zajištěno, že v množině všech tříd aplikace existuje jak třída D, která představuje typ parametru, tak třída A, která je návratovým typem. Typ Expression v těle metody pak musí být roven návratové hodnotě metody nebo být třídou, která je jejím potomkem.

```
class B {
        C c;
        A foo(D d) {/* Expression */}
}
```

Při ověřování typové konzistence modelu dochází k zajímavé vazbě mezi metodou a výrazem uloženým v jejím těle. Pokud totiž nastane případ, kdy sekvence výrazů končí voláním metody, znamená to, že je typ výrazu závislý na těle této metody. Tím se dostáváme do jakéhosi pomyslného kruhu, kdy je typ výrazu definován typem metody a ten zase typem výrazu. K vyřešení tohoto problému by bylo třeba vytváření stromových struktur a rozkládání všech zanořených těl metod.

V meta-modelu je ale všechno jinak. Kruh se rozbíjí hned při svém vzniku. Každá metoda má ve svém schématu nastavenou návratovou hodnotu *return*. Při kontrole konzistence modelu je pak ověřeno, zda všechny výrazy v tělech metod vrací deklarovanou návratovou hodnotu. Díky této znalosti se už nemusíme nadále zabývat rozebíráním výrazů v metodách a sestavováním stromů. Lze jednoduše říct, že metoda vrací typ XY, protože je tak deklarováno v modelu aplikace a ten je v konzistentním stavu.

I přes tuto vlastnost meta-modelu je však nutné výrazy z různých hledisek ověřovat.

V každé sekvenci výrazů, která se v modelu v tělech metod vyskytuje, je nutné ověřit, zda by vůbec mohla existovat. Tedy zda nedochází k volání metod nad objekty, které je vůbec nemají atd. Základní vlastností sekvence výrazů je, že se musí rovnat *out* hodnota předešlého výrazu s *caller* hodnotou následujícího.

ValidExpressionSequenceInv

$$a? : APPLICATION$$

$$\forall e_1, e_2 : EXPRESSION;\ expr : \operatorname{seq} EXPRESSION;\ n_1, n_2 : \mathbb{N} \bullet$$
$$n_1 < \#expr \land n_2 = n_1 + 1 \land expr(n_1) = e_1 \land expr(n_2) = e_2 \Rightarrow$$
$$e_1.out = e_2.caller$$

Kvůli možnosti větvení a zanořování výrazů skrze parametry metod a konstruktorů je nutné ověřit, že podvýraz, jehož výstup je vstupem do parametru, typově odpovídá.

ValidExpressionSubsequenceInv

$$a? : APPLICATION$$

$$\forall e : EXPRESSION;\ es : INSEQOFEXPR;$$
$$expr : \operatorname{seq} EXPRESSION;\ m : METHOD;\ c : CONSTRUCTOR \bullet$$
$$e \in \operatorname{ran}(expr) \land es.e = e \land (e.type = m \lor e.type = c) \land es.in \neq \langle\rangle \Rightarrow$$
$$\forall ex : EXPRESSION \bullet$$
$$ex \in \operatorname{ran}(es.in) \Rightarrow$$
$$\exists p : PROPERTY \bullet$$
$$(p \in m.params \lor p \in c.params) \land$$
$$p.target = ex.out$$

U všech výrazů musí být také ověřeno, zda je hodnota *caller*, *out* a počet prvků v *in* skutečně odpovídající.

ValidExpressionPropertiesInv

$$
\begin{array}{l}
a? : APPLICATION \\
\hline
\forall\, e : EXPRESSION;\ es : INSEQOFEXPR;\ m : METHOD; \\
\qquad c : CONSTRUCTOR;\ p : PROPERTY;\ cl : CLASS \bullet \\
es.e = e \Rightarrow \\
\quad e.type = m \lor e.type = c \Rightarrow \\
\quad\quad (\#(es.in) = \#(m.params) \lor \#(es.in) = \#(c.params)) \land \\
\quad\quad\quad (e.out = m.return \lor e.out = c.type) \land \\
\quad\quad\quad ((\exists\, moc : METHODOFCLASS \bullet \\
\quad\quad\quad\quad moc.method = m \land moc.class = cl \Rightarrow e.caller = cl) \\
\quad\quad\quad \lor (\exists\, coc : CONSTRUCTOROFCLASS \bullet \\
\quad\quad\quad\quad coc.constructor = c \land coc.class = cl \Rightarrow e.caller = cl)) \\
\quad \lor e.type = p \Rightarrow \\
\quad\quad \#(es.in) = 0 \land e.out = p.target \land \\
\quad\quad \exists\, poc : PROPERTYOFCLASS \bullet \\
\quad\quad\quad poc.property = p \land poc.class = cl \Rightarrow e.caller = cl
\end{array}
$$

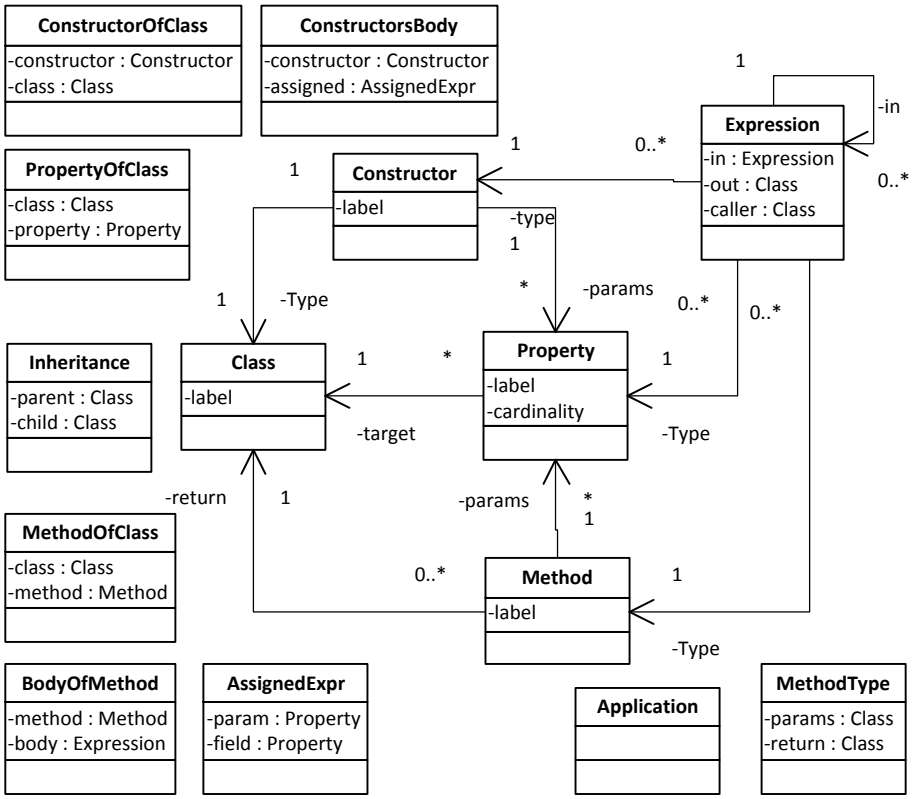
6.5 Konzistence modelu

Všechna předešlá omezení spolu se statickými omezeními modelu nám zajistí, že jakákoliv transformace modelu povede buď do konzistentního stavu, který bude splňovat všechna pravidla, nebo do nekonzistentního stavu, kdy model nesplňuje alespoň jedno z nich.

ApplicationIsConsistent

$a? : APPLICATION$

$ConstructorInExpressionMustExistsInv \land$
$FieldOrVariableInExpressionMustExistsInv \land$
$MethodInExpressionMustExistInv \land$
$ParameterInAssignedExpressionMustExistsInv \land$
$FieldInAssignedExpressionMustExistsInv \land$
$MethosAreOwnedByClassInv \land$
$ConstructorsAreOwnedByClassInv \land$
$ClassHaveOneConstructorInv \land$
$ExpressionIsOwnedByBodyOfMethodInv \land$
$TypeOfMethosParametersMustExistsInv \land$
$MethodParametersMustHaveUniqueNamesInv \land$
$ReturnTypeOfMethodMustExistsInv \land$
$NoMethodOverloadingInClassInv \land$
$NoMethodOverloadingInHierarchyInv \land$
$NoOverridingOfMethodsInv \land$
$NamesOfConstructorAndOwnedClassAreEqualInv \land$
$ConstructorParamsAndClassPropertiesAreEqualsInv \land$
$ConstructorTypeAndOwnedClassAreEqualsInv \land$
$NumbeOfExpressionInConstructorIsValidInv \land$
$ReturnTypeAndExpressionTypeAreEqualInv \land$
$ValidExpressionSequenceInv \land$
$ValidExpressionSubsequenceInv \land$
$ValidExpressionPropertiesInv$



Obrázek 6.1: Aplikační meta-model rozšířený o metody, konstruktory a výrazy

Kapitola 7

Pokrytí statických transformačních pravidel

Jak již bylo mnohokrát zmíněno, celá tato práce se zabývá rozšířením stávajícího formálního meta-modelu. Je proto třeba ověřit, že přidáním nových entit do meta-modelu nenastanou exekuční problémy s již definovanými transformacemi a invarianty. Jde o jakýsi regresní test, zda nově přidaná funkcionalita nebude mít vliv na tu stávající.

Meta-model je postavený velice modulárně a autonomně, proto už je předem odhadnutelné, že nově přidané entity nebudou mít vliv na žádná transformační pravidla. U těch však nyní může nastat, že v situacích, kdy bez rozšíření vedly ke konzistentní změně modelu, povedou do nekonzistetntního stavu. To však není způsobeno chybnou implementací pravidel, ale obří sadou nových invariantů, které hlídají, aby byly všechny nové elementy konzistentní.

U většiny transformací musí být ohlídáno, že nebude porušená typová konzistence výrazů a metod.

7.1 AddClass

Při vytvoření nové třídy v modelu nevzniká žádný problém s metodami ani s výrazy. Prázdná třída je pouze vytvořena a přiřazena do množiny všech tříd aplikace (*classes*).

7.2 RemoveClass

Při smazání třídy z množiny všech tříd aplikace dochází k několika problémům, které se musí kontrolovat. V metodách se jedná o následující problémy:

- Třída nesmí být typem žádného parametru

Pokud například z modelu budeme chtít smazat třídu *String*, nesmí se už nikde v modelu třída vyskytovat mezi typy parametrů (v příkladu 7.2 se jedná o parametr *name*).

```
Human setName (String name) {
        return new Human(name, this.age);
}
```

- Třída nesmí být návratovým typem metody

Obdobný případ jako u typu proměnných nastává i u návratových hodnot. Nelze vracet typ, který již v modelu neexistuje. Z teoretického hlediska tato situace nikdy nastat nemůže, protože ve výrazech je kontrolováno, zda používají existující elementy modelu. Pro ucelenost konceptu je ale vhodné mít tento problém také definovaný.

```
String setName () {
        return new String("Petr");
}
```

U výrazů se jedná o následující problém:

- Třída se nesmí vyskytovat v žádném výrazu, tedy tam, kde se vytváří nové objekty.

Do výrazů se problém s chybějící třídou dostane i skrze atributy a parametry. Tyto kolize však řeší již hotové invarianty ve statickém modelu.

7.3 AddProperty

S tvorbou atributu není žádný problém. Pokud do jakékoliv třídy přibude nový atribut, na metody ani na výrazy to nebude mít žádný vliv.

U konstruktorů se však musí zařídit, že při vytvoření nového atributu se vytvoří nový parametr v konstrukotru a také nové přiřazení v těle konstruktoru.

7.4 RemoveProperty

Mazání atributu má velký dopad na výrazy v metodách.

- Atribut se nesmí vyskytovat v žádném výrazu v rámci hierarchie. Může totiž nastat situace, kdy se odstraní z kořenové třídy nějaký atribut, který je ale použit v metodě o několik potomků níže.

Odstranění atributu třídy má také vliv na konstruktory.

- Při odstranění atributu musí nastat odebrání parametru z konstruktoru a také odstranění přiřazení v těle konstrukotoru.

7.5 AddParent

Jen pro upřesnění operace přidání rodiče není akt, při kterém se vytvoří dané třídě nový prázdný rodič. Jedná se o operaci, při které se dané třídě přiřadí již plnohodnotná, atributů a metod plná třída. V takové situaci je proto nutné kontrolovat přetěžování a přepisování metod.

- Problém s přetěžováním: nová rodičovská třída nesmí obsahovat metodu se stejným názvem, ale rozdílným počtem parametrů. Když už je počet parametrů stejný, nesmí dojít ani ke kovarianci či kontravarianci, tedy zobecňování či konkretizování parametrů nebo návratových hodnot.
- Problém s přepisováním: nová rodičovská třída nesmí obsahovat typem a názvem zcela identickou metodu.

7.6 RemoveParent

Při smazání rodiče je nutné kontrolovat výrazy v těle metod.

- Se třídou nesmí být smazán žádný atribut, který je použit v nějakém výrazu.
- Se třídou nesmí být smazána žádná metoda, která je použita v nějakém výrazu.

7.7 PushDown

Tato operace představuje refaktoring, při kterém se přesune atribut z předka do potomků. Taková změna nemá na metody a výrazy žádný vliv.

7.8 PullUp

Při této změně dochází k přesunu atributu z potomka do předka. Tato změna také nemá žádný vliv na metody a výrazy. Dokonce ani na volání metod, jak ukazuje následující příklad.

Kapitola 8

Evoluční transformace

Evoluční proces software je zpravidla množina změn, která aplikaci dostane z výchozího stavu do finálního. Každý software, který chce být v průběhu času konkurenceschopný, musí za svůj životní cyklus projít mnoha takovými změnami. Tyto změny, ač se může zdát, že se liší projekt od projektu, se dají velice snadno zobecnit. Vytváří se takzvané refaktoringové katalogy, které nabízí návody, na co nejčistší migrování z jedné verze aplikace do druhé.

U každé aplikované transformace je kladen obrovský důraz na udržení konzistentnosti modelu. Žádná operace nesmí model dostat do nekonzistentního stavu. Aby se tato vlastnost u transformačních pravidel udržela, byla všechna pravidla napsána jako co nejelementárnější změny z jednoho stavu modelu do druhého. Náročnější transformace se pak vždy dají složit jako sekvence elementárních transformací.

Za pomoci těchto elementárních změn pak dochází k požadovanému evolučnímu procesu aplikace.

Evoluce je ve formálním modelu tvořena sadou aplikovaných transformací, které cestou vytváří mezistavy aplikace. Než se tedy software dostane z výchozího do finálního stavu, může cestou projít mnoha mezistádii.

8.1 Transformační pravidla

Všechna následující transformační pravidla jsou převzata z Fowlerova katalogu refaktoringů [4]. Z katalogu jsou vybrány takové transformace, které se nejvíce dotýkají dynamických vlastností modelu. U každé transformace je popsán její význam, příklad užití a vlastní pravidlo, reprezentující změnu na úrovni formálního modelu.

Každé transformační pravidlo se skládá ze dvou kroků. Prvním krokem je definice samotného transformačního pravidla - Operace s postfixem *Succ*. Druhým krokem je aplikace transformačního pravidla na model. Ten má vždy název stejný jako operace s transformačním pravidlem, ale bez postfixu.

V rámci zpřehlednění textu není u každé operace uveden druhý krok, tedy ověření konzistence modelu. Tento krok je u všech operací totožný. Jediné, co se vždy mění, je název operace, které se do modelu vkládá a vstupy, které musí být vždy totožné s operací *Succ* stejného jména.

Aby bylo možné označit nějaký výstup transformace jako nevalidní, existuje značkovací schéma *FAILSTATE*

OperationName

$$
\begin{array}{l}
\Delta APPLICATION \\
< inputs >? :< TYPE > \\
\hline
\exists\, x, y : APPLICATION \bullet \\
\quad x = \theta(APPLICATION) \land \quad y = \theta(APPLICATION)' \land \\
\quad (ApplicationIsConsistent[x/a?] \land < OperationName > Succ \land \\
\qquad ApplicationIsConsistent[y/a?] \Rightarrow y.type = CONSISTENTSTATE) \lor \\
\quad (\neg\, ApplicationIsConsistent[x/a?] \land x = y) \lor \\
\quad (ApplicationIsConsistent[x/a?] \land \neg\ < OperationName > Succ \land \\
\qquad \neg\, ApplicationIsConsistent[y/a?] \Rightarrow y.type = FAILSTATE)
\end{array}
$$

Transformace může dopadnout dvěma způsoby. Buď je validní vstup a po aplikování operace i validní výstup, nebo je validní vstup a po aplikování operace dostaneme nevalidní výstup. Existuje také i teoretická situace, kdy dostáváme již nevalidní vstup. V takovém případě se vždy výstup rovná vstupu.

8.1.1 Add Method

Operace Add method vytvoří v definované třídě novou metodu. Model po přidání musí splňovat všechny podmínky konzistence. Z těch například vyplývá, že se metody nesmí přetěžovat či přepisovat, a to jak v rámci třídy, tak v rámci hierarchie.

Každá metoda, kterou do modelu vložíme, musí být přidána do množiny všech metod v modelu *methods*. Dále musí být metodě přiřazena vlastnící třída. Vytvoří se tedy i vazební schéma *METHODOFCLASS*, které vazbu mezi elementy zajistí.

addMethodSucc

$$
\begin{array}{l}
\Delta APPLICATION \\
c? : CLASS \\
m? : METHOD \\
\hline
methods' = methods \cup \{m?\} \\
\exists\, moc : METHODOFCLASS \bullet \\
\quad initMethodOfClass[moc/moc!] \land \\
\quad methodsOfClasses' = methodsOfClasses \cup \{moc\}
\end{array}
$$

8.1.2 Add Expression

Operace Add expression vytváří výraz, který musí být přiřazen do nějaké sekvence výrazů v těle metody. Výraz je automaticky přiřazen vždy na konec sekvence. Následně musí být ověřeno, že souhlasí návaznost hodnoty *out* předešlého prvku a *caller* tohoto nového výrazu.

addExpressionSucc

$\Delta APPLICATION$
$m? : METHOD$
$e? : EXPRESSION$

$expressions' = expressions \cup \{e?\}$
$\exists\, bom : BODYOFMETHOD \bullet$
$bom.method = m? \Rightarrow$
  $bom.body = bom.body \frown \langle e? \rangle$
$\lor\ bom.method \neq m? \Rightarrow$
  $initBodyOfMethod[bom/bom!] \land$
  $bodyOfMethods' = bodyOfMethods \cup \{bom\}$

8.1.3 Add Expression Sequence

Operace Add Expression Sequence vytvoří kompletní sequenci výrazů v těle metody. Po přidání musí platit vazba *out-caller* v sekvenci a rovnost návratové hodnoty metody a *out* hodnoty posledního prvku.

addExpressionSequenceSucc

$\Delta APPLICATION$
$m? : METHOD$
$e? : \text{seq}\, EXPRESSION$

$\forall\, exp : EXPRESSION \bullet exp \in \text{ran}(e?) \Rightarrow$
  $addExpression[exp/e?]$

8.1.4 Add Constructor

Operace Add Constructor vytvoří konstruktor v definované třídě. U konstruktorů je nutné kontrolovat jejich jediněčnost v rámci třídy, rovnost názvu s názvem třídy a odpovídající počet a typ parametrů.

Při vytváření konstruktoru v modelu je třeba vytvořit vazební schéma *CONSTRUCTOROFCLASS*, které zajistí jedinečnou vazbu mezi konstruktorem a jeho třídou.

addConstructorSucc

$\Delta APPLICATION$
$c? : CLASS$
$co? : CONSTRUCTOR$

$constructors' = constructors \cup \{co?\}$
$\exists\, coc : CONSTRUCTOROFCLASS \bullet$
  $initConstructorOfClass[coc/coc!] \land$
  $constructorsOfClasses' = constructorsOfClasses \cup \{coc\}$

8.1.5 Remove Method

Operace Remove Method odstraní metodu z dané třídy. Podmínky konzistence jsou pro jakékoliv odstraňovací (prefix *remove*) operace mnohem složitější, než u operací vkládacích (prefix *add*).

Pokud chceme odebrat metodu, nejde pouze o její odstranění z odpovídajících množin. Musí se také ověřit, že metoda není volána v žádném složeném výrazu ($CALLEXPR$) v celém modelu a neexistuje žádný výraz, který by měla uložený ve svém těle.

removeMethodSucc

$$\Delta APPLICATION$$
$$c? : CLASS$$
$$m? : METHOD$$

$$methods' = methods \setminus \{m?\}$$
$$\forall\, moc : METHODOFCLASS \bullet$$
$$moc.class = c? \land moc.method = m? \land$$
$$methodsOfClasses' = methodsOfClasses \setminus \{moc\}$$

8.1.6 Remove Expression

Operace Remove Expression odstraní výraz ze sekvence výrazů v těle metody. Tato operace odstraní vždy jen poslední výraz v sekvenci výrazů.

Ze strukturálního hlediska by nebyl žádný problém, kdyby operace mohla mazat libovolný výraz ze sekvence. Výraz by se odstranil a u sekvence by se ověřilo, zda odpovídá návaznost *out-caller*.

Z logického hlediska by však došlo k drobné nekonzistenci při práci s modelem skrze operace. Při vkládání přidáváme výraz vždy jen na konec sekvence. Při odstraňování bychom tedy měli dodržovat obdobný systém a mazat výrazy od konce.

Pokud odstraníme výraz, musí být zaručeno, že poslední prvek v sekvenci vrací typ, který koresponduje s návratovou hodnotou metody.

removeExpressionSucc

$$\Delta APPLICATION$$
$$m? : METHOD$$

$$\forall\, bom : BODYOFMETHOD \bullet$$
$$bom.method = m? \land$$
$$expressions' = expressions \setminus \{last\ (bom.body)\} \land$$
$$bom.body = front\ (bom.body)$$

8.1.7 Remove Expression Sequence

Operace Remove Expression Sequence odstraní sekvenci výrazů uložených v těle metody.

removeExpressionSequenceSucc

$\Delta APPLICATION$
$m? : METHOD$
$e? : \text{seq}\, EXPRESSION$

$\forall\, bom : BODYOFMETHOD \bullet$
$bom.method = m? \land e? \,\text{suffix}\, bom.body \Rightarrow$
$(\forall\, ex : EXPRESSION \bullet$
$ex \in \text{ran}(e?) \Rightarrow$
$removeExpression)$

8.1.8 Remove Constructor

Operace Remove Constructor odstraní konstruktor ze třídy. U odstranění konstruktoru je nutné ověřit, zda není používán v nějakém složeném výrazu (*CREATEEXPR*).

removeConstructorSucc

$\Delta APPLICATION$
$c? : CLASS$
$co? : CONSTRUCTOR$

$constructors' = constructors \setminus \{co?\}$
$\forall\, coc : CONSTRUCTOROFCLASS \bullet$
$coc.class = c? \land coc.constructor = co? \land$
$constructorsOfClasses' = constructorsOfClasses \setminus \{coc\}$

8.1.9 Extract Method

Extrahování části výrazu z těla metody do nové pomocné metody je jedním z nejčastějších refaktoringů software obecně. Využívá se nejen ke zvýšení přehlednosti a čitelnosti kódu, ale také ke zvýšení efektivity kódu v podobě znovupoužití nových metod.

Drobnou odlišností transformace od standardního refaktoringu podle Fowlera [4] je umístění nové metody v modelu. Při standardním refaktoringu je extrahovaná metoda vytvořena ve stejné třídě jako metoda původní. Meta-model však takový postup ze své vlastní podstaty nedovoluje. Operace Extract Method vyjme poslední n-tici prvků ze sekvence výrazů, které jsou uloženy v těle metody. První výraz v této sekvenci musí mít definovaný *caller* a to typu jeho vlastnící třídy. Pokud nenastane situace, kdy náhodou extrahujeme novou metodu právě ve stejné třídě jako je typ *calleru* prvního výrazu, musí dojít k tomu, že nově extrahovaná metoda bude uložena ve třídě, ve které se vyskytuje první výraz. K lepší představě poslouží příklad (8.1.9)).

```
class CleverClass{
    Node getBiggestNode(){
        return getAllNodes().sortList().getFirst();
    }

    List getAllNodes(){ /* Expression */ }
}

class List{
    SortList sortList(){ /* Expression */ }
}

class SortList{
    Node getFirst(){ /* Expression */ }
}

class Node{}
```

Třída *CleverClass* poskytuje metodu *getBiggerstNode*(), která z listu všech *Node* prvků vrátí ten největší. Výraz v metodě postupuje následovně. Nejprve si vezme seznam všech *Node* prvků. Ten je uložený v objektu typu *List*. Nad tímto objektem všechny prvky srovná od největšího po nejmenší metodou *sortList*(), který seřazený seznam vrátí v podobě objektu *SortList*. Z takto seřazeného seznamu objektů už pak jen stačí vzít první prvek metodou *getFirst*(), kterou poskytuje třída *SortList*.

Úkolem operace Extract Metod je extrahování sekvence operací *sortList*().*getFirst*() do nové metody, která se bude jmenovat *getFirstInSortList*().

Pokud bychom chtěli postupovat jako při standardním refaktoringu a novou metodu *getFirstInSortList*() bychom vytvořili ve třídě *CleverClass*, situace by vypadala následovně:

```
class CleverClass{
    Node getBiggestNode(){
        return getAllNodes().getFirstInSortList();
    }

    List getAllNodes(){ /* Expression */ }

    Node getFirstInSortList(){
        return sortList().getFirst();
    }
}
```

Kompilátor při tomto zápisu však ihned začně ohlašovat chybu. Není totiž možný. V samotné operaci *getFirstInSortList*() není problém. Vrací typ *Node*, tedy stejný typ, který vrací metoda *getFirst*() a vyžaduje původní metoda *getBiggestNode*() a obsahuje správnou sekvenci výrazů. Problém je však v jejím umístění. První výraz v nové metodě je volání operace *sortList*(). Kdybychom si do zápisu přidali hodnotu *this*, mohli bychom psát *this.sortList*(). Zde nastává problém. Objekt typu *CleverClass* ani třída *CleverClass* neposkytují metodu

sortList(). Tu ve svém interface vůči okolí poskytuje třída *List*. Řešením tohoto problému je vytvořit novou metodu ve třídě *List*. Výsledek extrahování pak bude vypadat následovně:

```
class CleverClass{
    Node getBiggestNode(){
        return getAllNodes().getFirstInSortList();
    }

    List getAllNodes(){ /* Expression */ }
}

class List{
    SortList sortList(){ /* Expression */ }

    Node getFirstInSortList(){
        return sortList().getFirst();
    }
}

class SortList{
    Node getFirst(){ /* Expression */ }
}

class Node{}
```

Pokud se na problém podíváme optikou výrazů a jejich parametrů, stále jde pouze o dodržování posloupnosti $out - caller$. Pro metodu *sortList*() je *caller* typu *List*, proto pokud před ním není jiný výraz, který má jako parametr *out* typ *List*, musí být metoda volána pouze v těle třídy, kde je uložena.

Extract Method je z hlediska meta-modelu v podstatě posloupnost několika atomických operací. Pokud dochází k extrahování, musí se dodržet následující scénář transformace:

- Vytvoření nové metody v cílové třídě (Operace *addMethod*). Tento krok není zcela triviální. Pokud máme v plánu přesunutí výrazu, který využívá některý z parametrů metody, nebo nějaký z atributů třídy, musíme novou metodu také parametrizovat.
- Vytvoření sekvence výrazů (Operace *addExpressionSequence*). Sekvenci přiřadíme do těla nové metody a z původní sekvence ji odstraníme.
- Vytvoření nového výrazu volání metody (*addExpression*). Do tohoto výrazu je vložena nová extrahovaná metoda a elementární výraz je přiřazen do původního těla metody na konec sekvence.

Po vykonání této sekvence operací musí platit všechna omezení konzistence modelu. Nesmí dojít ke kolizi jmen, musí existovat všechny v těle metody použité proměnné, atributy, konstruktory a metody. Nesmí dojít k přetěžování ani k přepisování metod. Musí platit všechna omezení týkající se typové konzistence, a to jak návratové hodnoty, tak typů parametrů a typů v těle metody.

Extrahování metody funguje pouze v rámci oněch výše popsaných dvou tříd. Pokud je v refaktoringu vyžadováno například extrahování do třídy předka, je nutná kombinace operace Extract method s dalšími operacemi meta-modelu.

Další podmínkou je, že metody, vyskytující se v extrahované sekvenci výrazů, nesmí ve svých parametrech používat žádné jiné metody. Z principu meta-modelu by se totiž muselo jednat o metody stejné třídy, jako je původní metoda. Ve výsledku by tak muselo dojít k přesunu všech těchto používaných metod. Takový krok však v rámci objektového modelování nedává nejmenší smysl.

Omezení by se dalo vyhnout jedním jednoduchým rozšířením meta-modelu, které by ale ve výsledku způsobilo skoro exponenciální zesložitění transformačních a validačních pravidel. Stačilo by definovat, že v *in* sekvenci každého výrazu může být znovu celá sekvence výrazů, tedy *sec* (*sec EXPRESSION*). Díky tomu by mohlo ve výrazech, které jsou použity v parametrech, docházet i k volání metod nad objekty, nejen k přímému volání metod, jak je tomu v meta-modelu nyní.

extractMethodSucc

$$
\begin{array}{l}
\Delta APPLICATION \\
m? : METHOD \\
newM? : METHOD \\
e? : \mathrm{seq}\, EXPRESSION \\
\hline
\exists\, bom : BODYOFMETHOD;\ mEx : EXPRESSION; \\
\qquad\qquad esEx : INSEQOFEXPR;\ tarC : CLASS \bullet \\
bom.method = m? \land e? \,\mathrm{suffix}\, bom.body \land esEx.e = mEx \Rightarrow \\
\quad tarC = (head\ (e?)).caller \land addMethod[newM?/m?, tarC/c?] \land \\
\quad bom.body = bom.body \setminus e? \land \\
\quad addExpressionSequence[newM?/m?] \land \\
\qquad mEx.out = newM?.return \land \\
\qquad mEx.caller = tarC \land \\
\qquad mEx.type = newM? \land \\
\qquad (\forall\, ex : EXPRESSION;\ es : INSEQOFEXPR \bullet \\
\qquad ex \in \mathrm{ran}(e?) \land es.e = ex \Rightarrow \\
\qquad\quad ((ex.type = m \lor ex.type = c) \land es.in \neq \langle\rangle) \Rightarrow \\
\qquad\qquad esEx.in = esEx.in \frown es.in \\
\qquad\quad \lor\ ((ex.type = p) \land es.in \neq \langle\rangle) \Rightarrow \\
\qquad\qquad esEx.in = \langle\rangle) \land \\
\quad addExpression[mEx/e?]
\end{array}
$$

8.1.9.1 Chování operace v rozšířeném meta-modelu

Operace Extract Method se od svého oficiální refaktoringového protějšku liší umístěním nové metody. Pokud by však meta-model obsahoval jen drobnou úpravu, operace by se dala udělat i standardním způsobem.

Stačilo by definovat, že schéma pro tělo metody *BODYOFMETHOD* má v *body* místo sekvence výrazů sekvenci sekvencí výrazů seq(seq *EXPRESSION*). Tím by byla povolena

práce s řádky na meta-modelové úrovni. V operaci Extract Method by pak místo přesunu v rámci sekvence šlo o přesun několika řádků do nového těla metody.

Tato změna sice vypadá velice snadně, ale její dopad na validátory a typový systém je obrovský. V prvé řadě pokud chceme přemýšlet nad řádky v metodách, musíme mít nadefinovaný mnohem větší počet *EPRESSION*, nebo povolit návratovou hodnotu *VOID*. Obě tyto možnosti však přináší do typového systému až exponenciální zesložitění všech validačních operací.

Jelikož by zajištění typové konzistence a validity modelu bylo výrazně náročnější a nepřineslo by mnoho přidané hodnoty, řádky nebyly do meta-modelu implementovány.

8.1.10 Inline Method

Operace Inline Method je v podstatě opakem Extract Method. Pokud se v kódu nachází nějaká metoda, jejíž funkcionalita je mizivá a navíc není nikde znovupoužita, je dobré tělo takové metody vložit tam, kde je metoda volána.

Situace se dá ilustrovat na stejném příkladu jako u operace Extract Method, pouze s tím rozdílem, že počáteční stav je 8.1.9 a finální stav po transformaci je 8.1.9.

Z tohoto příkladu také vyplývá jedna z vlastností operace Inline Method, která je rozdílná oproti operaci Extract Method. Při operaci Inline Method dochází ke kompletnímu smazání vkládané metody.

Stejně jako se u operace Extract Method v rámci zpřehlednění transformačních pravidel smí odebírat sekvence jen odzadu, tak se v Inline Method dá vkládat pouze v případě, že je volání vkládané metody v nové metodě na konci sekvence.

Operace Inline Method se dá rozdělit do sekvence elementárních operací:

- Smazání výrazu reprezentujícího vkládanou metodu (*removeExpression*).
- Vyjmutí celé sekvence výrazů a vložení na konec nové metody (*addExpressionSekvence*).
- Smazání sekvence a těla vkládané metody.
- Smazání vkládané metody (*removeMethod*).

inlineMethodSucc

$$
\begin{array}{l}
\Delta APPLICATION \\
c? : CLASS \\
m? : METHOD \\
inlineM? : METHOD \\
\hline
\exists\, bom_1, bom_2 : BODYOFMETHOD;\ seqE : \mathrm{seq}\, EXPRESSION \bullet \\
bom_1.method = m? \land bom_2.method = inlineM? \land seqE = bom_2.body \Rightarrow \\
\quad removeExpression \land addExpressionSequence[seqE/e?] \land \\
\quad removeExpressionSequence[seqE/e?, inlineM?/m?] \land \\
\quad bodyOfMethods' = bodyOfMethods \setminus \{bom_2\} \land \\
\quad removeMethod[inlineM?/m?]
\end{array}
$$

8.1.11 Move Method

Přesouvání metod mezi třídami je jedním z nejčastějších refaktoringů vůbec. Často je důvodem například přílišná provázanost třídy (coupling), vysoká funkční vybavenost třídy, nebo nelogičnost umístění metody.

Samotná meta-modelová operace přesunu je velmi zatížena omezeními. Aby byl model po aplikování operace v konzistentním stavu, musí platit následující vstupní požadavky:

- Metoda nesmí využívat žádných dalších prvků třídy (atributy, jiné metody).
- V nové třídě nemí existovat žádná metoda se stejným jménem.
- Nikde v modelu nesmí být v žádném výrazu metoda použita.

Jedná se sice o přísná omezení. Pokud má ale k přesunu metody dojít, meta-model poskytuje operace, kterými se tyto vstupní požadavky dají zajistit (např. smázání metody ze sekvence výrazů).

Operace Move Method je stejně jako Extract Method nebo Inline Method operací komplexní, tedy složenou z elementárních operací meta-modelu a rozšířenou o potřebnou dodatečnou logiku.

- Vytvoření nové metody se stejným názvem v cílové třídě (*addMethod*).
- Vytvoření těla metody a importování celé sekvence výrazů do nového těla metody.
- Smazání původního těla metody (*removeExpressionSequence*).
- Smazání původní metody (*removeMethod*).

moveMethodSucc

$$
\begin{array}{l}
\Delta APPLICATION \\
m? : METHOD \\
targetC? : CLASS \\
\hline
\exists\, newM : METHOD;\ bom : BODYOFMETHOD;\ seqE : \mathrm{seq}\, EXPRESSION \bullet \\
newM.label = m?.label \land newM.params = m?.params \land \\
newM.return = m?.return \land bom.method = m? \land seqE = bom.body \Rightarrow \\
\quad addMethod[newM/m?, targetC?/c?] \land \\
\quad addExpressionSequence[newM/m?, seqE/e?] \land \\
\quad removeExpressionSequence[seqE/e?] \land \\
\quad bodyOfMethods' = bodyOfMethods \setminus \{bom\} \land \\
\quad removeMethod[targetC?/c?]
\end{array}
$$

Kapitola 9

Ukázka chodu transformačního pravidla

Představme si v aplikaci situaci jako z příkladu v kapitole Extract Method 8.1.9. Třída *CleverClass* poskytuje metodu *getBiggerstNode*(), která z listu všech *Node* prvků vrátí ten největší. Výraz v metodě postupuje následovně. Nejprve si vezme seznam všech *Node* prvků. Ten je uložený v objektu typu *List*. Nad tímto objektem všechny prvky srovná od největšího po nejmenší metodou *sortList*(), která seřazený seznam vrátí v podobě objektu *SortList*. Z takto seřazeného seznamu objektů už pak jen stačí vzít první prvek metodou *getFirst*(), kterou poskytuje třída *SortList*.

Jelikož výrazy nemají v meta-modelu nadefinovaný název, pro přehlednost v této ukázce je každý výraz pojmenován. Koncovka ${}_{c}on$ značí použití konstruktorů a ${}_{m}$ použití metod. U metod tříd, těl metod a konstruktorů je situace stejně vizualizačně nevhodná jako u výrazů, proto mají také přiřazená jména.

Třídy:
$\{label : Node\}$; $\{label : List\}$; $\{label : SortList\}$; $\{label : CleverClass\}$

Metody:
$\{label : sortList, params : \varnothing, return : SortList\}$
$\{label : getFirst, params : \varnothing, return : Node\}$
$\{label : getBiggestNode, params : \varnothing, return : Node\}$
$\{label : getAllNodes, params : \varnothing, return : List\}$

Konstruktory:
$\{label : Node, params : \varnothing, type : Node\}$
$\{label : List, params : \varnothing, type : List\}$
$\{label : SortList, params : \varnothing, type : SortList\}$
$\{label : CleverClass, params : \varnothing, type : CleverClass\}$

Výrazy:
sortList_con: $\{in : \langle\rangle, out : SortList, caller : NULLCLASS, type : SortList\}$
node_con: $\{in : \langle\rangle, out : Node, caller : NULLCLASS, type : Node\}$
list_con: $\{in : \langle\rangle, out : List, caller : NULLCLASS, type : List\}$
sortList_m: $\{in : \langle\rangle, out : SortList, caller : List, type : sortList\}$
getFirst_m: $\{in : \langle\rangle, out : Node, caller : SortList, type : getFirst\}$
getAllNodes_m: $\{in : \langle\rangle, out : List, caller : CleverClass, type : getAllNodes\}$

Metody tříd:
sortList_L: $\{class : List, method : sortList\}$
getFirst_SL: $\{class : SortList, method : getFirst\}$
getAllNodes_CC: $\{class : CleverClass, method : getAllNodes\}$
getBiggestNode_CC: $\{class : CleverClass, method : getBiggestNode\}$

Těla metod:
sortList_sortList_con: $\{method : sortList, body :< sortList\_con >\}$
getFirst_node_con: $\{method : getFirst, body :< node\_con >\}$
getAN_list_con: $\{method : getAllNodes, body :< list\_con >\}$
getBN_seq: $\{method : getBiggestNode, body :< getAllNodes\_m, sortList\_m, getFirst\_m >\}$

Konstruktory tříd
N_N: $\{constructor : Node, class : Node\}$
L_L: $\{constructor : List, class : List\}$
SL_SL: $\{constructor : SortList, class : SortList\}$
CC_CC: $\{constructor : CleverClass, class : CleverClass\}$

Aplikace
$\{$
$classes : \{Node, List, SortList, CleverClass\},$
$properties : \varnothing,$
$propertiesOfClasses : \varnothing,$
$methods : \{sortList, getFirst, getAllNodes, getBiggestNode\},$
$methodsOfClasses : \{sortList\_L, getFirst\_SL, getAllNodes\_CC, getBiggestNode\_CC\},$
$constructors : \{Node, List, SortList, CleverClass\},$
$constructorsOfClasses : \{N\_N, L\_L, SL\_SL, CC\_CC\},$
$constructorsBodies : \varnothing,$
$inheritances : \varnothing,$
$bodyOfMethods : \{sortList\_sortList\_con, getFirst\_node\_con, getAN\_list\_con, getBN\_seq\},$
$expressions : \{sortList\_con, node\_con, list\_con, sortList\_m, getFirst\_m, getAllNodes\_m\},$
$assignedExpressions : \varnothing$
$\}$

V takto nadefinované aplikaci se rozhodneme pro extrahování sekvence výrazů *sortList*().*getFirst*() do nové metody s názvem *getFirstInSortList*(). Jde tedy o aplikaci operace *extractMethod*.

9.1 Aplikace operace Extract Method

Vstupem operaci Extract method tedy jsou:

- Původní metoda $m?$.
- Nová, extrahovaná metoda $newM?$.
- Sekvence výrazů $e?$, které mají být z původní metody $m?$ extrahovány.

V modelu musí existovat tělo metody ($BODYOFMETHOD$) takové, že je jeho vlastnící metodou $m?$ a obsahuje potřebnou sekvenci výrazů $e?$. Hledaná sekvence $e?$ musí být navíc suffixem celého těla metody. Tedy nacházet se na konci původního těla metody. Takovým tělem je *getBiggestNode_seq* Jelikož je operace Extract Method skrze více tříd, musí se nadefinovat cílová třída $tarC$. Tato třída odpovídá *out* třídě prvního výrazu v extrahovanému výrazu. Tedy *SortList*.

Dále se v modelu vytvoří nová metoda. Jinak řečeno se metoda $newM?$ uloží do struktury aplikace. Z původního těla metody se odstraní hledaná sekvence výrazů, tedy $< sortList\_m, getList\_m >$. Do nově vzniklé metody se uloží vstupní sekvence $e?$. Spolu s ní se vytvoří i nové tělo metody new_body.

Jako poslední krok je třeba vytvořit nový výraz new_expr, který bude reprezentovat volání nové metody a přiřadí se na konec sekvence v původní metodě $m?$. Při vytváření nového výrazu je jako *out* nastavena hodnota *Node*, jako *caller* je nastavena $tarC$, tedy *SortList* a sekvence vstupních parametrů *in* zůstane prázdná.

Po vykonání všech těchto operací vypadá model následovně (v rámci přehlednosti jsou zobrazenou pouze místa, kde došlo ke změně):

Metody:
$\{label : getFirstInSortList, params : \varnothing, return : Node\}$

Výrazy:
new_expr: $\{in : \langle\rangle, out : Node, caller : List, type : getFirstInSortList\}$

Metody tříd:
new_m: $\{class : List, method : getFirstInSortList\}$

Těla metod:
getBN_seq: $\{method : getBiggestNode, body :< getAllNodes\_m, new\_expr >\}$
new_body: $\{method : getBiggestNode, body :< sortList\_m, getFirst\_m >\}$

Aplikace
{
$methods : methods \cup \{getFirstInSortList\}$,
$methodsOfClasses : methodsOfClasses \cup \{new\_m\}$,
$bodyOfMethods : bodyOfMethods \cup \{new\_body\}$,
$expressions : expressions \cup \{new\_expr\}$,
}

Kapitola 10

Type soundness neboli typová solidnost

V kapitole 5 byl vytvořen formální meta-model aplikační vrstvy s ohledem na vysokou typovou ochranu. To znamená, že pokud je program dobře otypovaný, neměl by se nikdy "zaseknout". Této vlastnosti se říká *soundness* neboli volně do češtiny přeloženo *solidnost*. Kapitola 6 pak všechna matematicky formálně nadefinovaná pravidla převádí do reality jazyka Z a vytváří k nim i požadované invarianty. V kapitole devět je pak možnost vidět, jak fungují tranformace nad ukázkovým modelem.

To vše však nijak nedokazuje, že je tento vytvořený model opravdu typově zapouzdřený, že opravdu nemůže nastat situace, kdy by se model mohl "zaseknou", i když bude zcela dobře otypovaný.

Důkaz solidnosti modelu si ale žádá definování několika dalších vlastností, které v modelu ne zcela mohou být vidět. Jedná se o *progress*, neboli česky *vývoj*. Nejprve je totiž třeba dokázat, že pokud máme složený výraz, můžeme ho zpracovávat, vyvíjet do té doby, než dojdeme do jeho normální formy, tedy do zcela elementárního výrazu.

Vše bude pro svou kompaktnost znovu napsáno v matematickém zápise, který využívá FJ. Proto jen pro připomenutí zápis $\Gamma \vdash e : C$ znamená, že v prostředí Γ je výraz e, který má typ C. Zápis $C' <: C$ definuje dědičnost, kde C' je potomkem C. Nutno ještě dodat, že jakýkoliv element, který nad sebou má linku $\overline{e}$, je množinou.

Redukce

$$Kdyz\ \Gamma \vdash e : C\ a\ e \to e',\ pak\ \Gamma \vdash e' : C',\ pro\ ktere\ plati\ C' <: C \tag{10.1}$$

Nyní je třeba si projít jeden případ po druhém. Pokud se ve výrazu objeví atribut třídy, jedná se o tento případ:

$$e = (new\ C_0(\overline{e})).f_i \qquad e' = e_i cProperties(C_0) = \overline{D}\ \overline{f} \tag{10.2}$$

Pro konstruktor musí mít správně zadané parametry:

$$\Gamma \vdash \overline{e} : \overline{C} \qquad C <: C_0 \tag{10.3}$$

Podle vlastností modelu platí, že konstruktor vrací typ třídy, ve které je vytvořen, tedy $new\ C_0(\overline{e})$ vrací hodnotu C_0. Jak již bylo napsáno, třída C_0 má atributy typu $\overline{D}\ \overline{f}$. Pak pro každé i platí, že $f_i : D_i$. e' má pak typ podle volaného atributu.
Pro metody je případ následující:

$$e = (new\ C_0(\overline{e})).m(\overline{d}) \qquad mBody(m, C_0) = e_0 \tag{10.4}$$

Pokud nyní celý výraz nahradíme tělem metody, musíme udělat nasledující substituci:

$$e' = [\overline{d}/\overline{x},\ new\ C_0(\overline{e})/this]e_0 \qquad kde\ \overline{x}\ jsou\ parametry\ pouzite\ v\ E_0 \tag{10.5}$$

V těle metody tedy všechny výrazy *this* nahradíme konstruktory a všechny parametry za reálné hodnoty do metody vložené. Pro shrnutí platí:

$$\Gamma \vdash new\ C_0(\overline{e}) : C_0 \qquad mType(m,\ C_0) = \overline{D} \to C \tag{10.6}$$

$$\Gamma \vdash \overline{d} : \overline{C} \qquad \overline{C} <: \overline{D} \tag{10.7}$$

To ve finále znamená, že pokud nově rozebraný výraz e_0 má typ E, pak musí platit, že $E <: C$, a e' je tedy typu E.

Progress
Nyní dokážeme redukovat výrazy. Dále je třeba dokázat, že výrazy umíme "vyvíjet", tedy že je umíme redukovat až do jejich normální formy. Předpokládejme, že e je dobře otypované.

- Když e obsahuje jako jeden ze svých subvýrazů $(new\ C_0(\overline{e})).f_i$ pak podle redukce a v okamžiku, kdy jsou všechny členy subvýrazu dobře otypované, platí, že $cProperties(C_0) = \overline{C}\ \overline{f}$ a $f \in \overline{f}$ pro nějaká $\overline{C}$ a $\overline{f}$.
- Když e obsahuje jako jeden ze svých subvýrazů $(new\ C_0(\overline{e})).m(\overline{d})$, pak podle redukce a v okamžiku, kdy jsou všechny členy subvýrazu dobře otypované, platí, že $mType(m,\ C_0) = \overline{D} \to C$ a $\#(\overline{x}) = \#(\overline{d})$ pro nějaká $\overline{x}$ a e_0.

V tuto chvíli už dokážeme redukovat výrazy a už můžeme i říci, že výraz zůstane dobře otypovaný, i pokud zredukujeme nějaký jeho subvýraz. Nyní se dostáváme k definici solidnosti.

Soundness
Cílem našeho dokazování je snaha o poznání tvaru výrazu v jeho normální formě. Ve zjednodušeném meta-modelu si můžeme dovolit říci, že normální formou každého výrazu je výraz, který obsahuje pouze konstruktor (v jehož parametrech mohou být znovu jen konstruktory). Jinak řečeno že $k = new\ C(\overline{k})$.

Definice solidnosti pak vypadá následovně:

Pokud je $\varnothing \vdash e : C$ a $e \rightsquigarrow e$ s tím, že e' je normální forma, pak e' může nabývat jedině podoby $\varnothing \vdash k : D$, kde $D <: C$.

Toto tvrzení je důsledkem redukčních pravidel. Pokud redukujeme nějaký atribut třídy ve výrazu typu $new\ Pair(new\ A(),\ new\ B()).second$, vrátí nám hodnotu $new\ B()$. V metodách se postupnými redukcemi dostaneme až do podobného stavu jako u volání atributu a vše vždy skončí až u konstruktoru.

Díky prokázání typové solitnosti lze konstatovat, že pokud je model vytvořený z rozšířeného meta-modelu dobře otypovaný, nikdy se ve svém chodu nezasekne.

Model pouze reprezentuje stav nějakého syntakticky omezeného JAVA programu. Díky type soundness je prokázáno, že tato omezená podmnožina jazyka JAVA je tak bezpečná, že pokud je program vždy typově správný, nikdy neskončí s chybovou hláškou.

Je třeba zmínit, že typová solidnost nezaručuje správnost výsledku. Sémantiku, tedy smysl zapsaných algoritmů, si musí programátor ohlídat sám.

Kapitola 11

Shrnutí

11.1 Zhodnocení nástrojů

Jelikož je celý meta-model s transformacemi psaný v jazyce Z, pro vývoj existuje jediné uspokojivé vývojové prostředí - CZT: Community Z Tools [1]. Jelikož je toto prostředí postavené nad Eclipse IDE, je jeho ovládání vžité a intuitivní.

Problém prostředí je v jeho funkcionalitě a podpoře. Komunita kolem jazyka Z je velice malá a rozvíjí se opravdu pomalu. Je to znát jak na poskytovaných funkcích prostředí, tak na kompatibilitě kontroloru syntaxe a oficiální dokumentaci k notaci. Existuje celá řada zápisů, které vývojové IDE považuje za chybný zápis.

Posledním závažným omezením vývoje v jazyce Z je jeho obrovská nekompatibilita s moderními technologiemi. Vývoj se oficiálně zastavil již před mnoha lety, a tak se není čemu divit, že pro spuštění validátoru je třeba Python verze 1.2 nebo 8bit jádro systému. Tyto požadavky už v dnešní době nejsou splnitelné, proto se jazyk Z používá čistě jako notace pro formální zápis modelů.

11.2 Zhodnocení rozšířeného meta-modelu

Meta-model byl vytvořen s důrazem na maximální typovou čistotu. To má za následek jeho velikou funkční omezenost a špatnou praktickou nasaditelnost. Na druhou stranu se ale podařilo vytvořit správný předobraz pro budoucí vývoj v modelovacích nástrojích jako je Ecore. Přesně k tomu má formální metodologie sloužit, k důkladnému promyšlení problému a vytvoření sady teorémů, které pro problém musí platit. Meta-model je sestavený tak, že by do budoucna neměl být žádný problém ho rozšířit o požadované vlastnosti. Jde jak o rozšíření statických entit, tak i o rozšíření vlastností nových dynamických entit. Například více řádků v metodách, více možností ve výrazech, minimální omezení pro konstruktory.

Vylepšení se ale nemusí týkat jen prvků v meta-modelu. Může jít i o přidávání vlastností, tedy o změny invariantů nad modelem. Celý koncept jde rozšířit o podporu kovariance, kontravariance, primitivních typů, vazeb M:N, přetypování atd.

[1] CZT: Community Z Tools: http://czt.sourceforge.net

Kapitola 12

Závěr

Mým cílem bylo vytvoření rozšíření pro stávající formální meta-model o metody, konstruktory a výrazy. Základním kamenem takového rozšíření je předefinování meta-modelu. Ten je formálně popsaný v páté kapitole a do podrobností rozebraný v kapitole 6 a 7. Aby celé rozšíření dávalo smysl, bylo nutné vytvořit i transformace, které budou schopné nově vytvořené modely rozhýbat. Tyto transformace jsou popsané v kapitole 8. Příklad použití jedné z transformací a ukázka průchodu transformačním pravidlem je v kapitole 9. Důkaz type soundness nově vytvořeného modelu je v kapitole 10.

Rozšíření o metody by v budoucnousti přineslo obrovský benefit pro možné uživatele. Budou moci skrze jednotné rozhraní vytvářet kompletní sadu změn nad jejich aplikací. Framework totiž už nebude umět pouze změny v rámci ORM s ohledem na instance, bude zvládat i změny dynamického rázu, které jsou jedny z nejčastějších, jaké programátor ve svém systému provádí.

Literatura

[1] ABIAN, A. *The theory of sets and transfinite arithmetic.* Saunders mathematics books. Saunders, 1965.

[2] BOWEN, J. – STAVRIDOU, V. Safety-critical systems, formal methods and standards. *Software Engineering Journal.* 1993, 8, 4, s. 189–209.

[3] COLLINS, M. *Formal Methods* [online]. 2008. [cit. 7. 4. 2014]. Dostupné z: <https://www.ece.cmu.edu/~koopman/des_s99/formal_methods/>.

[4] FOWLER, M. *Refactoring: improving the design of existing code.* Č. 0201485672. 1999.

[5] IGARASHI, A. – PIERCE, B. C. – AVAYA, P. W. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems.* 2001, 23, 3, s. 189–209.

[6] KATEřINA, H. *Hodnocení indexace, kvalita a konzistence indexace* [online]. 2006. [cit. 15. 4. 2014]. Dostupné z: <http://www.phil.muni.cz/kivi/clanky.php?cl=67>.

[7] KOMUNITA. *Eclipse Modeling Framework Project (EMF)* [online]. 2012. [cit. 15. 4. 2012]. Dostupné z: <http://www.eclipse.org/modeling/emf/>.

[8] KOMUNITA. *MWE2 - The Modeling Workflow Engine 2* [online]. 2012. [cit. 15. 4. 2012]. Dostupné z: <http://www.eclipse.org/Xtext/documentation/2_0_0/118-mwe-in-depth.php>.

[9] KOMUNITA. *Xtend - Language made for JAVA developers* [online]. 2012. [cit. 15. 4. 2012]. Dostupné z: <http://www.eclipse.org/xtend/>.

[10] KOMUNITA. *Xtext - Language development made easy* [online]. 2012. [cit. 15. 4. 2012]. Dostupné z: <http://www.eclipse.org/Xtext/>.

[11] LUKEš, M. Dokončení projektu MigDB, 2014. Diplomová práce.

[12] MAZANEC, M. Doménově specifický jazyk pro MigDB, 2014. Diplomová práce.

[13] MERUNKA, V. *Objektové modelování.* Apla, 1th edition, 2008.

[14] OMG. *Meta Object Facility (MOF) 2.0 Query View Transformation Specification* [online]. 2011. [cit. 15. 4. 2012]. Dostupné z: <http://www.omg.org/spec/QVT/1.1/PDF/>.

[15] OMG. *MOF 2 XMI Mapping* [online]. 2011. [cit. 15. 4. 2012]. Dostupné z: <http://www.omg.org/spec/XMI/>.

[16] PAVEL MORAVEC, P. T. J. J. D. H. *A practical approach to dealing with evolving models and persisted data* [online]. 2012. [cit. 15. 4. 2012]. Dostupné z: <http://www.codegeneration.net/cg2012/sessioninfo.php?session=37>.

[17] RICHTA, K. *Jazyk OCL a modelem řízený vývoj* [online]. 2010. [cit. 15. 4. 2012]. Dostupné z: <https://www.ksi.mff.cuni.cz/~richta/publications/Richta-MD-2010.pdf>.

[18] SPIVEY, J. M. *The Z Notation: A Reference Manual*. Prentice Hall International, 2th edition, 1998.

[19] TARANT, P. et al. *Database Migration* [online]. 2011. [cit. 15. 4. 2014]. Dostupné z: <https://github.com/migdb/migdb>.

[20] web:minis. Ministerstvo dopravy. http://www.mdcr.cz/cs/default.htm.

[21] YOUNG, C. U. P. J. L. L. C. U. W. J. G. *Database migration*. Č. 20030028555. February 2003. Dostupné z: <http://www.freepatentsonline.com/y2003/0028555.html>.

Příloha A

Pomocné transformační metody

Tato kapitola obsahuje všechna pomocná pravidla předešlých transformací. Jde hlavně o inicializační metody vazebních schémat, které vytváří skutečné vazby mezi entitami v modelu.

initMethodOfClass

$c? : CLASS$
$m? : METHOD$
$moc! : METHODOFCLASS$

$moc!.class = c?$
$moc!.method = m?$

initBodyOfMethod

$e? : EXPRESSION$
$m? : METHOD$
$bom! : BODYOFMETHOD$

$bom!.method = m?$
$bom!.body = bom!.body \frown \langle e? \rangle$

initConstructorOfClass

$c? : CLASS$
$co? : CONSTRUCTOR$
$coc! : CONSTRUCTOROFCLASS$

$coc!.constructor = co?$
$coc!.class = c?$

Příloha B

Zkratky

CP CollectionsPro, s.r.o.

Ecore Core EMF

EMF Eclipse modeling framework

FJ Featherweight JAVA

IDE integrated development environment

JPA JAVA Persistence API

OCL Object Constraint Language

OMG Object management group

ORM Object-relational mapping

POJO Plain Old JAVA Object

QVT Query, View, Transformation

SQL structured query language

XMI XML metadata interchange

XML Extensible Markup Language

Příloha C

Obsah přiloženého CD

prostredi Nakonfigurované vývojové prostředí Eclipse - Community Z tool

migdb Rozšížení stávajícího meta-modelu a transformační pravidla

text Zdrojové kódy k textu diplomové práce