

České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů



Diplomová práce
Doménově specifický jazyk pro MigDb

Bc. Martin Mazanec

Vedoucí práce: Ing. Ondřej Macek

Studijní program: Otevřená informatika

Obor: Softwarové inženýrství

2. května 2014

Poděkování

Chtěl bych poděkovat všem, kteří mi jakýmkoliv způsobem pomáhali s přípravou diplomové práce. Jedná se zejména o vedoucího práce, Ing. Ondřeje Macka, za jeho cenné rady a veškerou pomoc. Dále svojí rodině za podporu a trpělivost.

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 2. května 2014

.....

Abstract

The evolution of the software is the problem that software developers have to face. One of the solutions of this problem is the school project called MigDb, which deals with the automatization of the evolution process. The project aims to solve the problem of object-relational mapping without losing the data stored in a relational database. Changes in persistent classes are described by the set of predefined operations that are used to generate a SQL script containing commands for a database schema and data migration.

The main purpose of this diploma thesis is to create the domain-specific language called Ops for easier handling of operation sequences in project MigDb. The language is formally specified and based on its specification the Eclipse IDE plug-in is implemented.

Abstrakt

Evoluce softwaru je palčivým problémem kterému čelí softwarový vývojáři. Jedním z možných řešení je použití studentského projektu MigDb, jež se zabývá automatizací evolučního procesu. Projekt si klade za cíl vyřešit problematiku objektově-relačního mapování bez ztráty dat uložených v relační databázi. Změny v persistentních třídách jsou popsány sadou předdefinovaných operací na jejichž základě dojde k vygenerování SQL skriptu obsahující příkazy pro migraci databázových schémat a dat.

Podstatou této diplomové práce je vytvoření doménově specifického jazyku Ops pro snadnou manipulaci se sekvencemi operací v projektu MigDb. Vzniklý jazyk je formálně specifikován a na základě specifikace je provedena jeho implementace jako jazykového plug-inu do vývojového prostředí Eclipse IDE.

Obsah

1	Úvod	1
2	Framework MigDb	3
2.1	Metamodely	4
2.1.1	Aplikační metamodel	4
2.1.2	Databázový metamodel	4
2.2	Struktura frameworku	5
2.2.1	Aplikační vrstva (APP)	5
2.2.2	Objektově-relační mapování operací (ORMo)	5
2.2.3	Databázová vrstva (RDB)	6
2.2.4	Generátor	6
2.2.5	Spouštěcí vrstva (nepovinná)	6
2.3	Současné použití	7
2.3.1	Nevýhody	7
2.4	Navrhované použití	8
3	Příbuzné projekty	9
3.1	ORM frameworky	9
3.1.1	Hibernate	9
3.1.2	Active Record	9
3.2	Evoluce databázové vrstvy	9
3.2.1	MeDEA	9
3.2.2	PRISM	10
3.2.3	DB-MAIN	10
3.2.4	Liquibase	10
3.3	Evoluce metamodelů	10
3.4	Shrnutí	10
4	Operace nad aplikačním modelem	13
4.1	Add Class	13
4.2	Remove Class	14
4.3	Add Property	14
4.4	Remove Property	15
4.5	Rename Property	15
4.6	Move Property	16

4.7	Pull Up Property	17
4.8	Push Down Property	18
4.9	Set Parent	19
4.10	Remove Parent	19
4.11	Extract Class	20
4.12	Extract SubClass	21
4.13	Extract SuperClass	22
5	Syntaxe jazyka Ops	23
5.1	Vývoj syntaxe	23
5.2	Finální verze syntaxe	24
5.3	Rozbor finální syntaxe	26
6	Operační sémantika	27
6.1	Značení a zápis	28
6.2	Operační sémantika	28
6.3	Struktura \mathcal{S}	28
6.4	Přepisovací pravidla	29
6.5	Vstupní a výstupní funkce	32
7	Typový systém	33
7.1	Typový systém	33
7.2	Typová pravidla	34
8	Vlastnosti jazyka Ops	39
8.1	Terminace	40
8.2	Soundness = Progress + Preservation	43
9	Implementace	47
9.1	Xtext gramatika	47
9.2	Typový systém	49
9.3	Rozšíření grafického rozhraní	51
9.3.1	Formátování	51
9.3.2	Průvodce novým Ops programem	51
9.4	Operační sémantika	52
9.5	Integrace jazyku Ops s MigDb	52
9.5.1	MWE2 pracovní tok	52
9.5.2	Spuštění Ops programu	53
10	Testování	55
10.1	Testovací příklad	55
10.1.1	Sestavení výchozího modelu	55
10.1.2	Vznik hierarchie	56
10.1.3	Extrahování atributů	57
10.2	Jednotkové testy	57
11	Závěr	59

Literatura	61
A Jazyk Ops	63
A.1 Gramatika	63
A.2 Operační sémantika	64
A.3 Typový systém	65
B Metamodely MigDb	67
B.1 Aplikační metamodel	67
B.2 Databázový metamodel	69
C Instalační a uživatelská příručka	71
C.1 Vytvoření Hello World	71
D Seznam použitých zkratk	73
E Obsah příloženého CD	75

Seznam obrázků

2.1	Rozdělení vrstev softwaru	3
2.2	Struktura frameworku MigDb, obrázek převzat z [22, str. 5]	5
2.3	Objektově-relační mapování operací, obrázek převzat z [22, str. 18]	6
4.1	Ukázka aplikace operace ADD CLASS	13
4.2	Ukázka aplikace operace REMOVE CLASS	14
4.3	Ukázka aplikace operace MOVE PROPERTY	16
4.4	Ukázka aplikace operace PULL UP PROPERTY	17
4.5	Ukázka aplikace operace PUSH DOWN PROPERTY	18
4.6	Ukázka aplikace operace EXTRACT CLASS	20
4.7	Ukázka aplikace operace EXTRACT SUBCLASS	21
4.8	Ukázka aplikace operace EXTRACT SUPERCLASS	22
6.1	Abstraktní stroj, obrázek převzat z [28, str. 48]	27
8.1	Možnosti vyhodnocení programu abstraktním strojem	39
9.1	Převod vstupního výrazu na abstraktní syntaktický strom (MigDb model)	48
9.2	UML diagram tříd implementace typového systému	49
9.3	Ukázka editoru Ops programů	51
9.4	Pracovní tok frameworku MigDb, obrázek převzat z [12, str. 13]	52
9.5	Spuštění Ops programu v Eclipse IDE	54
10.1	Výchozí stav testovacího modelu	55
10.2	Upravený testovací model - vznik hierarchie	56
10.3	Upravený testovací model - extrakce atributů	57
B.1	Aplikační metamodel frameworku MigDb - operace	67
B.2	Aplikační metamodel frameworku MigDb - struktura	68
B.3	Databázový metamodel frameworku MigDb - operace	69
B.4	Databázový metamodel frameworku MigDb - struktura	70
C.1	Vytvoření Hello World programu v jazyce Ops	72

Seznam tabulek

3.1 Přehled projektů zabývajících se evolučním procesem	11
---	----

Seznam teorémů

8.2	Tvrzení (Terminace)	41
8.3	Tvrzení (Progress)	43
8.4	Tvrzení (Preservation)	45

Seznam ukázek kódu

2.1	Současné použití operací v QVTO transformaci	7
2.2	Použití operací v jazyku Ops	8
5.1	Původní verze syntaxe jazyka Ops	24
5.2	Finální verze syntaxe jazyka Ops	24
9.1	Ukázka gramatických pravidel jazyka Ops v Xtext	48
9.2	Implementace typového pravidla T-REMCLS	50
9.3	Implementace úpravy typového kontextu pravidlem T-REMCLS	50
9.4	Sestavení aplikačního modelu z Ops programu	53
10.1	Sestavení testovacího modelu	56
10.2	Vyextrahování společného předka	56
10.3	Přesun adresy	57
10.4	Testovací program v Ops	58
10.5	Implementace metody pro jednotkové testování	58

Kapitola 1

Úvod

Mezi hlavní problémy, se kterými se vývojáři softwaru každodenně potýkají, patří evoluce softwarových systémů. Změny ve funkčních a nefunkčních požadavcích mají dopad na různé části systému, především na databázi a persistentní třídy (entity) [11], které je nutné modifikovat za účelem naplnění těchto požadavků. Evoluční proces nemá dopad pouze na strukturu daných částí systému, ale také na data uložená v databázi, jež je nutné migrovat s respektováním nové podoby databázových schémat.

Na trhu existuje nepřehledné množství nástrojů zabývajících se evolučním procesem na různých úrovních software. Jedním z nich je také studentský projekt MigDb [12, 14, 22] vzniklý na ČVUT. Tento projekt je dostupný jako plug-in pro Eclipse IDE [23] a klade si za cíl vyřešit objektově-relační mapování bez ztráty dat uložených v relační databázi. Myšlenka projektu spočívá v zavedení tzv. operací, kterými jsou popsány změny struktury entit softwaru. Na základě popsaných úprav dojde k vygenerování SQL skriptu obsahující příkazy pro změnu odpovídajících databázových schémat včetně migrace dat.

Nevýhody projektu MigDb spočívají v jeho nepříliš praktickém použití. Od uživatele tohoto projektu je vyžadováno definování transformačních souborů v jazyku QVTO [19], které obsahují sekvenci operací. Zápis v QVTO je značně upovídaný a vyžaduje zbytečné psaní hlaviček. Nepraktické je také spouštění transformačních souborů za účelem vygenerování SQL skriptu. Uživatel je přinucen pro každý transformační soubor sestavit pracovní tok MWE2 [26] nebo minimálně provést jeho úpravu. Dalším negativem projektu je jeho neformální a především neúplný popis dostupných operací.

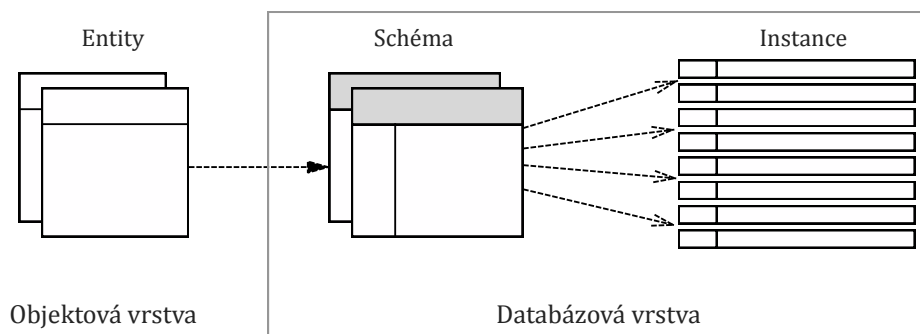
Předmětem této práce je vytvoření doménově specifického jazyku Ops zjednodušující sestavování sekvencí operací v projektu MigDb. Jazyk Ops bude formálně specifikován a na základě znalostí získaných z předmětu Teorie programovacích jazyků [29] budou ověřeny jeho vybrané vlastnosti. Podle vzniklé specifikace bude jazyk Ops implementován jako jazykový plug-in, využitím nástroje Xtext [25], do prostředí Eclipse IDE. Dále bude ověřena správnost jeho implementace na vybraných testovacích příkladech.

Členění textu je následovné. V kapitole 2 detailněji popíšeme princip projektu MigDb, jeho strukturu a použití. Neformálním vysvětlením dostupných operací MigDb se zabývá kapitola 4, která navíc popisuje jejich nutné předpoklady a aplikaci. Kapitoly 5, 6 a 7 definují syntaxi, operační sémantiku a typový systém jazyku Ops. Vybrané vlastnosti vyvíjeného jazyku jsou následně ověřeny kapitolou 8. Implementaci a testování jazykového plug-inu do Eclipse IDE jsou věnovány závěrečné kapitoly 9 a 10.

Kapitola 2

Framework MigDb

Tato kapitola je věnována hlubšímu představení frameworku MigDb [12, 14, 22], zabývající se evolučním procesem při vývoji software. Požadavky na software jsou průběžně měněny během jeho životního cyklu. Vzniklé změny (evoluce) mají dopad na různé části softwaru, které je nutné přetransformovat z počátečního stavu do požadovaného výsledku. MigDb se nezabývá evolucí softwaru jako celku, ale pouze jeho objektovou vrstvou (entity) a databázovou vrstvou (schéma a instance), viz Obr. 2.1, kdy vzniklé změny struktury na objektové úrovni je nutné propagovat do úrovně databázové, čímž dojde k úpravě odpovídajících databázových schémat. Evoluce nemusí způsobit pouze změnu struktury jednotlivých vrstev, ale může mít dopad na uložené instance dat v relační databázi, které je nutné upravit (migrovat) s respektováním nové formy databázových schémat.



Obrázek 2.1: Rozdělení vrstev softwaru

Proces evoluce změn v životním cyklu softwaru lze, podle [12, str. 4], shrnout do několika hlavních kroků, které zůstávají během vývoje softwaru obvykle neměnné:

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

Většina existujících ORM nástrojů pokrývá pouze kroky 1 a 2. Tyto nástroje jsou schopny automatizovaně reflektovat změny objektové vrstvy do databázových schémat, ale samotná migrace dat, popsána v krocích 3 až 5, musí být provedena manuálně. Hlavní nevýhodou manuální migrace je její náročnost a především velká náchylnost ke vzniku chyb, které mohou uvést databázi, posléze celý software, do nekonzistentního stavu. Cílem frameworku MigDb je co nejvíce evoluční proces zautomatizovat, zamezit nekonzistentním stavům v databázi a předejít zbytečné ztrátě v ní uložených dat. Více se o možných problémech rozepisuje Petr Tarant [22]. Bakalářská práce [12, str. 5] navrhuje, na místo výše zmíněných kroků, provést při evolučním procesu následující:

1. sběr potřebných informací pro migraci dat a příprava sady operací,
2. ověření proveditelnosti sady operací,
3. vykonání sady operací.

Navrhované řešení definuje sadu **operací** za pomoci které jsou popsány požadované úpravy objektové vrstvy. Sada operací obsahuje prostředky pro přidávání nových tříd a atributů až po pokročilejší úpravy jako jsou různé formy extrakce atributů. Dostupným operacím je věnována kapitola 4. Na základě vzniklé sady operací dojde k automatizované propagaci změn z objektové vrstvy do databázové, včetně samotné migrace instancí dat uložených v relační databázi. Navíc uvedené řešení umožňuje předejít možným problémům hned na začátku evolučního procesu, protože každá operace je, před vykonáním, ověřena na proveditelnost a v případě selhání nedojde ani ke změně objektové vrstvy narozdíl od původního postupu.

2.1 Metamodely

Frameworku MigDb je postaven na konceptu MDA [18] a pro popsání jednotlivých vrstev software zavádí metamodely, které definují strukturu modelů z nich vycházející. Instance metamodelů nám umožní snadno reprezentovat stav objektové a databázové vrstvy daného softwaru nad nímž provádíme evoluční proces. Navíc metamodely definují podobu jednotlivých operací, které lze nad oběma vrstvami provádět.

2.1.1 Aplikační metamodel

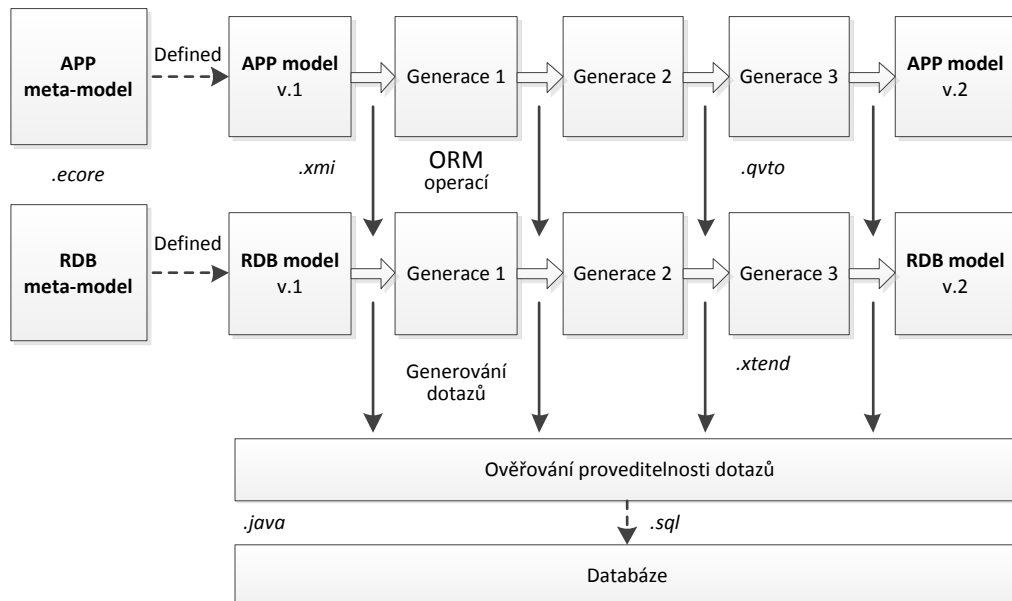
Jiří Ježek [12, str. 8] popisuje význam tohoto metamodelu následovně: „*Aplikační metamodel slouží jako formální přepis pro tvorbu konkrétních aplikačních modelů. Definuje dílčí komponenty, ze kterých se tyto konkrétní modely mohou skládat, a také zahrnuje sadu operací, které posléze mohou být nad těmito modely vykonávány.*“. Struktura metamodelu viz příloha B.1.

2.1.2 Databázový metamodel

„*Databázový metamodel obsahuje šablonu entit vhodných pro obecný modelový popis aktuálního stavu databáze. Každý databázový prvek či omezení je v modelu reprezentován vlastním objektem.*“ viz bakalářská práce [22, str. 11]. Kromě entit pro popis stavu databáze, obsahuje databázový metamodel sadu předdefinovaných operací, které mohou být aplikovány nad instancí metamodelu. Příloha B.2 obsahuje podobu daného metamodelu.

2.2 Struktura frameworku

Framework vznikl jako plug-in pro Eclipse IDE [23] a je rozdělen do čtyř základních vrstev (aplikační, ORM operací, databázové, generátor) a jedné doplňkové (spouštěcí) viz obrázek 2.2. Z pohledu uživatele frameworku MigDb je nejdůležitější aplikační vrstva umožňující popsat strukturu objektové vrstvy softwaru a definovat sadu operací, jež má být vykonána nad touto vrstvou. Jednotlivé vrstvy mezi sebou spolupracují za účelem vytvoření SQL skriptu, který je následně aplikován na danou relační databázi.



Obrázek 2.2: Struktura frameworku MigDb, obrázek převzat z [22, str. 5]

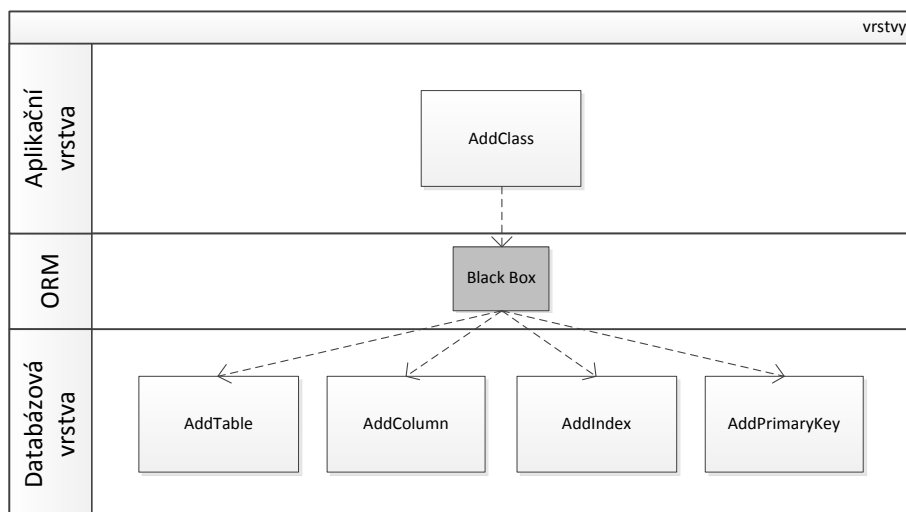
2.2.1 Aplikační vrstva (APP)

Součástí aplikační vrstvy je aplikační model vycházející z prvků aplikačního meta-modelu. Tento model obsahuje *strukturu* a sekvenci *operací*. Struktura popisuje jednotlivé entity objektové vrstvy softwaru, naopak operace zachycují požadované úpravy počátečního stavu struktury za účelem její přeměny do výsledné podoby. Nad aplikační vrstvou je spuštěna evoluční transformace napsaná v jazyce QVTO [19], která sekvenčně aplikuje jednotlivé operace na počáteční strukturu, čímž vznikají tzv. generace obsahující její aktualizovaný stav. Vykonáním evoluční transformace ověříme proveditelnost jednotlivých operací na aplikační vrstvě (objektové).

2.2.2 Objektově-relační mapování operací (ORMo)

Po úspěšném provedení evoluční transformace na aplikační úrovni dojde k přemapování jednotlivých aplikačních operací na operace databázové. Mapování nemusí být nutně v poměru 1:1, viz obrázek 2.3, protože aplikační operace jsou obvykle namapovány na více dílčích databázových operacích. Během mapování vznikne nový model - databázový model,

kteřý obsahuje sekvenci databázových operací, vzniklých přemapováním z aplikačních, ale také databázovou strukturu vycházející z počátečního stavu aplikační struktury. Databázová struktura obsahuje jednotlivé entity aplikační vrstvy ve formě databázových schémat a omezení, jež jsou reprezentovány prvky z databázového metamodelu.



Obrázek 2.3: Objektově-relační mapování operací, obrázek převzat z [22, str. 18]

2.2.3 Databázová vrstva (RDB)

Tato vrstva, obdobně jako aplikační, je složena z databázového modelu definovaného metamodelem z sekce 2.1.2. Databázový model obsahuje *strukturu* a *operace* na jejichž základě dojde ke spuštění evoluční transformace, která postupně aplikuje jednotlivé databázové operace na počáteční strukturu, tím ověří jejich proveditelnost na databázové vrstvě. Každou aplikací operace vznikají nové generace počátečního stavu struktury.

Uvědomme si, že narozdíl od aplikační vrstvy, jsou v této vrstvě brány v úvahu potenciační instance dat uložené v databázi. Například operace pro přesun atributů (**MOVE PROPERTY**) nemá žádný významný dopad na objektové vrstvě software, ale na databázové vrstvě může její vykonání způsobit ztrátu dat. Z tohoto důvodu musí být při ORM operacích zajištěno vytvoření kontrolních mechanismů zabranujících zbytečné ztrátě informací.

2.2.4 Generátor

Poslední ze čtyř základních vrstev je generátor, jehož účelem je vytvoření textového skriptu obsahující SQL příkazy k jednotlivým databázovým operacím, kterým předcházelo ověření jejich proveditelnosti na databázové vrstvě pomocí evoluční transformace.

2.2.5 Spouštěcí vrstva (nepovinná)

Nepovinná vrstva slouží k vykonání vygenerovaného SQL skriptu nad konkrétní instancí relační databáze. Navíc tato vrstva informuje uživatele frameworku o případných selháních vzniklých během vykonávání daného skriptu.

2.3 Současné použití

Nainstalováním všech součástí frameworku do Eclipse IDE je toto vývojové prostředí připraveno pro zahájení evolučního procesu software. Použití MigDb lze rozdělit do čtyř základních kroků:

1. vytvoření XMI [20] souboru zachycující strukturu objektové vrstvy software s respektováním prvků aplikačního metamodelu z sekce 2.1.1,
2. definování QVTO transformace obsahující sekvenci operací, jež má být vykonána nad danou aplikační strukturou a uvést jí do požadované podoby,
3. sestavení MWE2 [26] pracovního toku na základě komponent MigDb a cest k XMI souboru a transformaci,
4. zahájení evolučního procesu spuštěním pracovního toku.

Výstupem spuštění pracovního toku je, v případě úspěšného evolučního procesu, skript obsahující SQL příkazy vycházející z operací definovaných v transformaci. Vykonání vzniklého skriptu nad konkrétní relační databází má za následek přeměnu odpovídajících databázových schémat a migraci dat.

2.3.1 Nevýhody

Nejprve sestavme příklad QVTO transformace, viz ukázka kódu 2.1, obsahující sekvenci operací pro vytvoření tříd *Country* a *Person* včetně jejich atributů.

```

1 import builder_app;
2 import queries_app;
3 modeltype APP uses "http://www.collectionspro.eu/jam/mm/app";
4
5 transformation example(out inoutModel : APP);
6
7 main(){
8     var ops : OrderedSet(APP::ops::ModelOperation) := OrderedSet{};
9
10    ops += _addStandardClass("Country", false, APP::InheritanceType::joined);
11    ops += _addProperty("Country", "name", "String");
12
13    ops += _addStandardClass("Person", false, APP::InheritanceType::joined);
14    ops += _addProperty("Person", "fullname", "String");
15    ops += _addProperty("Person", "country", "Country");
16
17    _appOperations(ops);
18 }

```

Ukázka kódu 2.1: Současné použití operací v QVTO transformaci

Zápis použitý ve zmíněné ukázce má několik negativ mezi něž patří: jeho **upovídanost**, uvádění **hlavičky** transformace, viz řádky 1 až 7, ale také nutnost vytvářet **uspořádanou množinu**, v ukázce označenou jako *ops*, do které přidáváme jednotlivé operace.

Dalším negativem současného použití frameworku spočívá v **sestavování MWE2 pracovního toku** či v jeho úpravě pro každou novou QVTO transformaci.

2.4 Navrhované použití

Na základě nevýhod uvedených v sekci 2.3.1 navrhuji vytvořit doménově specifický jazyk Ops. Podle [6] by tak vznikl *programovací jazyk s omezenou vyjadřovací schopností zaměřující se na specifickou doménu*, jehož účelem by bylo sestavování sekvencí operací.

Dále navrhuji, aby namísto nepraktického vytváření MWE2 pracovního toku pro každou QVTO transformaci vzniklo rozšíření do vývojového prostředí Eclipse IDE, které by umožňovalo automatické sestavení pracovního toku a jeho následné spuštění.

```
1 add Country //addStandardClass
2 add Country.name : String //addProperty
3
4 add Person //addStandardClass
5 add Person.fullname : String //addProperty
6 add Person.country : Country //addProperty
```

Ukázka kódu 2.2: Použití operací v jazyku Ops

Ukázka 2.2 obsahuje identickou sekvenci operací z příkladu 2.1 sestavenou v navrhovaném doménově specifickém jazyce Ops, který bude popsán v následujících kapitolách. Na první pohled je zřejmé, že použitý zápis je kompaktnější a odstraňuje uvedené neduhy QVTO transformací.

Kapitola 3

Příbuzné projekty

V následující části textu představíme výčet vybraných projektů zabývajících se evolučním procesem při vývoji software na jeho různých úrovních.

3.1 ORM frameworky

3.1.1 Hibernate

Mezi nejvýznamější nástroj na poli objektově-relačního mapování na platformě Java patří bezpochyby Hibernate [10]. Podpora evoluce v tomto nástroji je značně omezená. Umožňuje provádět pouze základní operace: vytváření databázových schémat a sloupců na základě vzniku entit (tříd) či jejich rozšíření o nové atributy. Na složitější operace, jako jsou různé typy extrakce atributů, je Hibernate nedostatečný a tyto změny musejí být realizovány manuálně.

3.1.2 Active Record

Dalším nástrojem z rodiny ORM je Active Record [3], který je součástí webového frameworku Ruby on Rails [4]. Evoluční proces je v tomto nástroji podporován pouze na úrovni vytváření, odstraňování a úpravy databázových schémat na základě změny odpovídajících entit nebo jejich atributů z objektové vrstvy software. Active Record navíc disponuje DSL pro vytváření migračních skriptů. Těmito skripty je možné popsat jednoduché úpravy databázových schémat včetně samotné migrace dat.

3.2 Evoluce databázové vrstvy

3.2.1 MeDEA

Architektura MeDEA [5] je založena na konceptu modelem řízené evoluci databázové vrstvy. Nejedná se o konkrétní nástroj, ale o návrh jak postupovat během evoluce. Doporučuje evoluční proces rozdělit do více úrovní (konceptuální, logická, fyzická) s různým stupněm abstrakce (např. diagramy tříd, entity, databázová schémata) a dále definovat tzv. propagační pravidla, která jsou použita při propagaci změn z jedné úrovně do ostatních.

3.2.2 PRISM

Nástroj PRISM [2] slouží k evoluci databázových schémat a migraci dat využitím předdefinované sady tzv. schema modification operators (SMO), což jsou funkce, které změni schémata databáze podle specifikace a provedou migraci. Tento nástroj podporuje komplexní operace včetně spojování databázových schémat či jejich rozdělování na více částí.

PRISM je především určen pro databázové administrátory, kteří mohou skrze webové rozhraní vykonat úpravy nad danou instancí relační databáze.

3.2.3 DB-MAIN

Evoluční proces lze v grafickém nástroji [9] provádět napříč všemi vrstvami software, avšak nejprve je započat na databázové vrstvě. Vzniklé změny jsou následně propagovány do objektové vrstvy, což má za následek vytvoření nových entity či jejich úpravu.

3.2.4 Liquibase

Velmi rozpracovaný nástroj Liquibase [17] podporuje evoluční proces na databázové vrstvě. Definuje rozsáhlou množinu refaktoringů pomocí níž je možné provést migraci databázových schémat a dat nad různými typy databází. Použití jednotlivých refaktoringů probíhá pomocí XML souborů v nichž je přesně definována požadovaná úprava.

3.3 Evoluce metamodelů

Projekt COPE [8] se zabývá problematikou evoluce metamodelů jejichž změna způsobuje zneplatnění všech instancí modelů z nich vycházející. COPE definuje sadu operací, kterou lze aplikovat na počáteční stav metamodelu a uvést ho do požadované podoby. Na základě sestavené sekvence operací dojde k migraci jednotlivých modelů, čímž je změněna jejich struktura s respektováním nové verze metamodelu. Navíc tento nástroj ukládá historii vzniklých změn ke kterým je možné se zpětně vrátet.

3.4 Shrnutí

Evoluční proces může být proveden na různých úrovních software. V textu byly představeny frameworky zabývající se evolucí software na úrovni objektů, databázové vrstvy, ale také na úrovni metamodelů.

Frameworky z rodiny ORM podporují evoluční proces pouze omezenou mírou. Zvládají operace pro vytváření a úpravu databázových schémat na základě změn v objektové vrstvě, ale na komplexní operace jako jsou různé extrakce atributů nestačí. Samotná migrace dat musí být prováděna manuálně. Framework Active Record navíc podporuje operaci mazání a disponuje jazykem pro vytváření migračních skriptů.

Evoluci databázové vrstvy pokrývají projekty PRISM, DB-MAIN, Liquibase a MeDEA. První tři zmíněné projekty poskytují rozumné nástroje pro evoluci databázových schémat včetně migrace dat. Akademický projekt MeDEA neposkytuje reálnou implementaci, ale

doporučuje evoluční proces rozdělit do více úrovní abstrakce mezi nimiž jsou vzniklé evolute propagovány. Za vyzdvižení stojí projekt Liquibase od Nathana Voxlanda. Tento projekt poskytuje rozsáhlou sadu refaktoringů, kterými lze provést téměř jakoukoliv změnu databázové vrstvy.

Projekt COPE nabízí jiný pohled na evoluční proces - z pohledu metamodelů, kdy na základě vzniklých změn v metamodelu je nutné provést přeměnění struktury modelů z nich vycházející.

	Komplexní úpravy	Migrace dat	Způsob použití
Hibernate	✗	✗	Java
Active Record	✗	✓	Ruby+DSL
MeDEA	✓	✓	různé
PRISM	✓	✓	GUI+SMO
DB-MAIN	✓	✓	GUI
Liquibase	✓	✓	XML
COPE	✓	✗	Java+DSL
MigDb	✓	✓	QVTO/DSL

Tabulka 3.1: Přehled projektů zabývajících se evolučním procesem

Závěrem lze říci, že vyjmenované ORM frameworky nejsou příliš vhodné pro evoluci softwaru z důvodu jejich omezené podpory úprav objektové vrstvy, viz tabulka 3.1. Uvedené nástroje pro evoluci databázové vrstvy jsou konkurencí projektu MigDB a v některých faktorech jej i překonají. Nicméně jsou především orientované na databázové administrátory narozdíl od MigDb, které se snaží softwarové vývojáře co nejvíce odstínit od hlubších znalostí databázové vrstvy a evoluci provádět pouze na základě změn v objektové vrstvě.

Kapitola 4

Operace nad aplikačním modelem

V této části neformálně popíšeme význam, nutné předpoklady a kroky použití operací nad aplikačním modelem, jenž popisuje strukturu systému či jeho dílčí části. Jednotlivé operace vycházejí z refaktoringu software z literatury [7], posléze [15].

Každá z níže uvedených operací nějakým způsobem manipuluje s obsahem modelu. Operace nám umožňují vytvářet či upravovat entity systému, tvořit mezi nimi různé typy vazeb či je sdružovat do hierarchického seskupení. Navíc některé operace (EXTRACT CLASS a jiné) umožňují provádět komplexnější změny modelu v jediném kroku.

4.1 Add Class

Význam operace:

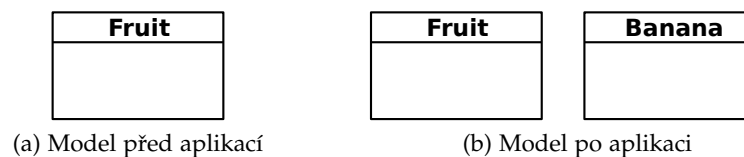
Operace slouží k přidání nové neexistující třídy do modelu.

Předpoklady:

1. Model neobsahuje třídu s názvem stejným jako je název přidávané třídy.

Aplikace:

1. Přidání pojmenované třídy do modelu.



Obrázek 4.1: Ukázka aplikace operace ADD CLASS

4.2 Remove Class

Význam operace:

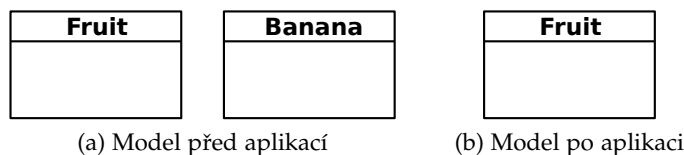
Odstranění existující třídy z modelu.

Předpoklady:

1. Existence třídy.
2. Odstraňovaná třída nemá žádné přímé potomky a atributy.
3. V modelu neexistuje atribut s typem odstraňované třídy.

Aplikace:

1. Odstranění třídy z modelu.



Obrázek 4.2: Ukázka aplikace operace REMOVE CLASS

4.3 Add Property

Význam operace:

Operaci ADD PROPERTY používáme k přidání nového neexistujícího atributu dané třídě v modelu. Každý atribut má svoje pojmenování, typ, dolní a horní kardinalitu omezující počet instancí zdola a shora.

Předpoklady:

1. Existence třídy, které je přidáván atribut.
2. Jméno atributu neexistuje v dané třídě, v jejích předcích ani potomcích.
3. Zvolený typ atributu existuje v modelu jako třída nebo se jedná o primitivní typ.
4. Horní kardinalita je větší než 0 a zároveň rovna či větší dolní kardinalitě, která může být nulová.

Aplikace:

1. Přidání pojmenovaného atributu, s typem, dolní a horní kardinalitou, dané třídě.

4.4 Remove Property

Význam operace:

Využitím této operace lze odstranit existující atributy třídy.

Předpoklady:

1. Existence třídy jejíž atribut bude odebrán.
2. Existence odstraňovaného atributu ve třídě.

Aplikace:

1. Odstranění atributu z dané třídy.

4.5 Rename Property

Význam operace:

RENAME PROPERTY slouží k přejmenování již existujícího atributu dané třídy.

Předpoklady:

1. Existence třídy jejíž atribut bude přejmenován.
2. Nové pojmenování atributu neexistuje v dané třídě, v jejích předcích ani potomcích.

Aplikace:

1. Odstranění původního atributu z třídy využitím operace [REMOVE PROPERTY](#).
2. Přidání nového atributu, se jménem odpovídající přejmenování, s typem, dolní a horní kardinalitou identickou původnímu atributu, dané třídě aplikací operace [ADD PROPERTY](#).

4.6 Move Property

Význam operace:

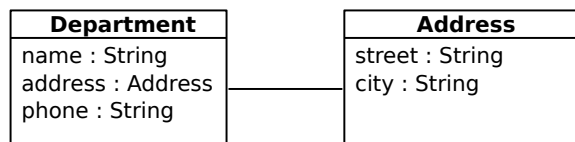
Použitím operace `MOVE PROPERTY` lze přesouvat atributy z jedné třídy do druhé využitím existujícího spojení (asociace) mezi oběma třídami.

Předpoklady:

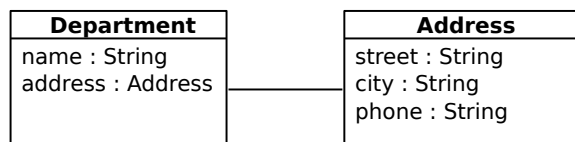
1. Existence zdrojové třídy ze které bude atribut přesunut.
2. Existence cílové třídy do které bude atribut přesunut.
3. Existence přesouvaného atributu ve zdrojové třídě.
4. Existence asociace ve zdrojové třídě, jejíž typ odpovídá cílové třídě s kardinalitou 1:1.
5. Jméno přesouvaného atributu neexistuje v cílové třídě, v jejích předcích ani potomcích.

Aplikace:

1. Použitím operace `REMOVE PROPERTY` odebereme atribut ze zdrojové třídy.
2. Přidání atributu, s identickými údaji jako měl právě odstraněný atribut ze zdrojové třídy, do cílové třídy využitím operace `ADD PROPERTY`.



(a) Model před aplikací



(b) Model po aplikaci

Obrázek 4.3: Ukázka aplikace operace `MOVE PROPERTY`

4.7 Pull Up Property

Význam operace:

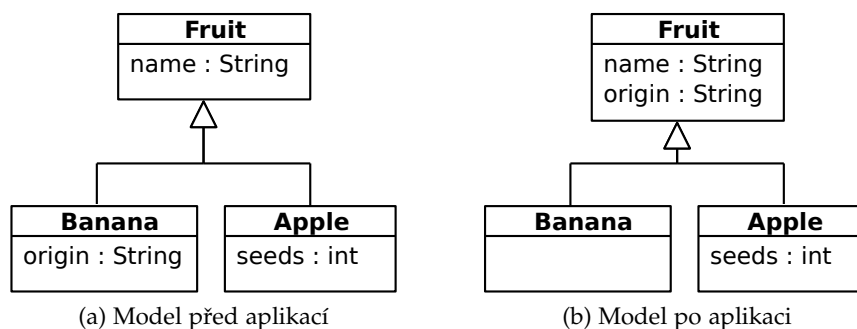
Operace `PULL UP PROPERTY` slouží k přesunu atributů v rámci hierarchie, kdy přesouváme atributy zdrojové třídy směrem nahoru do jejího rodiče.

Předpoklady:

1. Existence zdrojové třídy ze které bude atribut přesunut.
2. Existence rodiče zdrojové třídy.
3. Existence přesouvaného atributu ve zdrojové třídě.
4. Jméno přesouvaného atributu neexistuje v sourozencích ani předcích zdrojové třídy.

Aplikace:

1. Odstranění atributu ze zdrojové třídy aplikací `REMOVE PROPERTY`.
2. Přidání atributu, s identickými údaji jako měl právě odstraněný atribut ze zdrojové třídy, do rodičovské třídy operací `ADD PROPERTY`.



Obrázek 4.4: Ukázka aplikace operace `PULL UP PROPERTY`

4.8 Push Down Property

Význam operace:

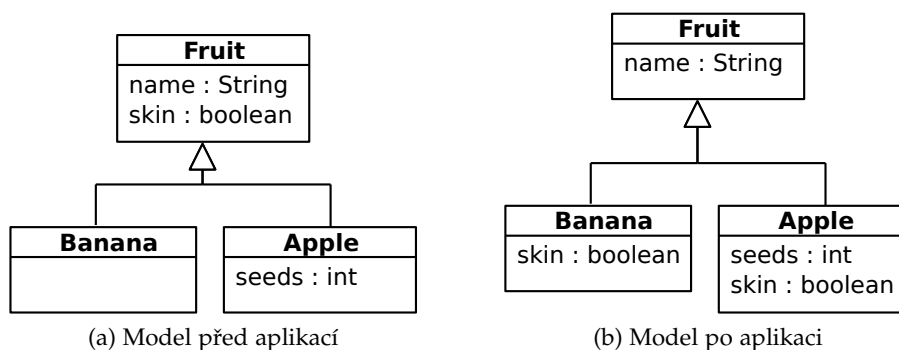
PUSH DOWN PROPERTY, obdobně jako operace PULL UP PROPERTY, slouží k přesunu atributů v rámci hierarchie s tím rozdílem, že nám umožňuje přesun atributů opačným směrem, tedy ze zdrojové třídy směrem dolů do všech přímých potomků.

Předpoklady:

1. Existence zdrojové třídy ze které bude atribut přesunut.
2. Existence alespoň jednoho přímého potomka zdrojové třídy.
3. Existence přesouvaného atributu ve zdrojové třídě.
4. Jméno přesouvaného atributu neexistuje v potomcích zdrojové třídy.

Aplikace:

1. Odstranění atributu ze zdrojové třídy aplikací [REMOVE PROPERTY](#).
2. Přidání atributu, s identickými údaji jako měl právě odstraněný atribut ze zdrojové třídy, do všech přímých potomků zdrojové třídy několika násobným použitím operace [ADD PROPERTY](#).



Obrázek 4.5: Ukázka aplikace operace PUSH DOWN PROPERTY

4.9 Set Parent

Význam operace:

Operace SET PARENT slouží k nastavení rodiče dané třídy.

Předpoklady:

1. Existence zdrojové třídy, které bude nastaven rodič.
2. Existence rodičovské třídy.
3. Nastavením rodičovské třídy jako rodiče zdrojové třídy nevznikne v hierarchii tříd cyklus.
4. Jména všech atributů zdrojové třídy a jejích potomků se nevyskytují v rodičovské třídě ani v jejích předcích.

Aplikace:

1. Nastavení rodiče dané třídy.

4.10 Remove Parent

Význam operace:

Aplikací operace REMOVE PARENT lze zrušit vztah potomek-rodič mezi dvěma třídami v hierarchickém uspořádání.

Předpoklady:

1. Existence zdrojové třídy.
2. Existence rodiče zdrojové třídy.

Aplikace:

1. Odstranění vztahu potomek-rodič mezi zdrojovou třídou a jejím rodičem.

4.11 Extract Class

Význam operace:

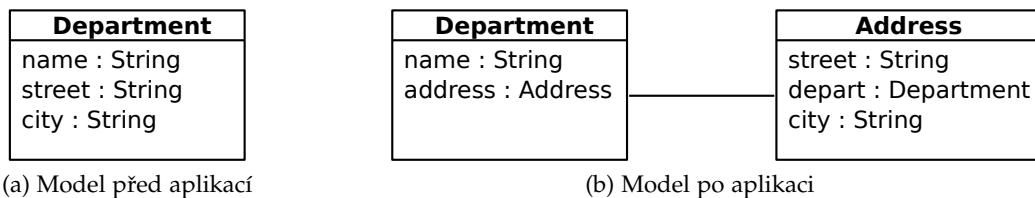
EXTRACT CLASS slouží k delegování zodpovědností (atributů) jedné třídy na novou třídu využitím tzv. extrahování. Extrahování vytvoří novou třídu do níž jsou přesunuty delegované zodpovědnosti, a následně mezi zdrojovou třídou a právě vyextrahovanou třídou vznikne obousměrné spojení (asociace).

Předpoklady:

1. Existence zdrojové třídy.
2. Existence extrahovaného atributu ve zdrojové třídě.
3. Model neobsahuje třídu s názvem odpovídající extrahované třídě.
4. Zdrojová třída včetně jejích potomků a předků neobsahuje atribut se stejným pojmenováním jako vytvářená asociace.

Aplikace:

1. Přidání extrahované třídy do modelu využitím [ADD CLASS](#).
2. Vytvoření obousměrné asociace s kardinalitou 1:1 mezi zdrojovou třídou a právě vyextrahovanou třídou operací [ADD PROPERTY](#).
3. Přesunutí extrahovaného atributu pomocí [MOVE PROPERTY](#) ze zdrojové třídy do vyextrahované třídy.



Obrázek 4.6: Ukázka aplikace operace EXTRACT CLASS

4.12 Extract SubClass

Význam operace:

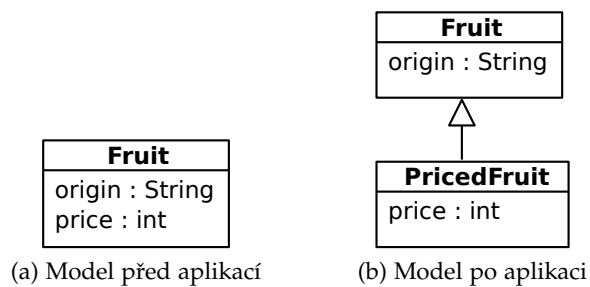
Operace EXTRACT SUBCLASS se využívá v situacích, kdy některé vlastnosti (atributy) třídy nejsou využívány všemi jejími instancemi. Řešení spočívá ve vyextrahování nevyužitých atributů do speciálních potomků zdrojové třídy.

Předpoklady:

1. Existence zdrojové třídy.
2. Existence extrahovaného atributu ve zdrojové třídě.
3. Extrahovaná třída neexistuje v modelu.

Aplikace:

1. Přidání extrahované třídy do modelu využitím [ADD CLASS](#).
2. Nastavení vztahu potomek-rodíč mezi extrahovanou třídou a zdrojovou třídou pomocí [SET PARENT](#).
3. Přesunutí extrahovaného atributu ze zdrojové třídy do všech jejích přímých potomků využitím operace [PUSH DOWN PROPERTY](#).



Obrázek 4.7: Ukázka aplikace operace EXTRACT SUBCLASS

4.13 Extract SuperClass

Význam operace:

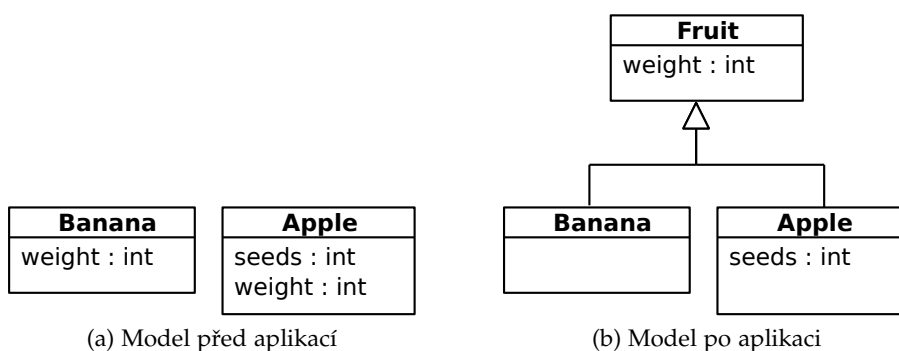
Tato operace řeší situace, kdy model obsahuje několik různých tříd mající stejnou vlastnost (atribut), ale žádného rodiče. Operace vyextrahuje společnou vlastnost z jednotlivých zdrojových tříd do nové třídy, která je posléze nastavena jako rodič všech zdrojových tříd.

Předpoklady:

1. Existence zdrojových tříd.
2. Žádná zdrojová třída nemá rodiče.
3. Extrahovaný atribut existuje ve všech zdrojových třídách s identickým typem a kardinalitou.
4. Model neobsahuje třídu s názvem odpovídající extrahované třídě.

Aplikace:

1. Vytvoření extrahované třídy pomocí operace [ADD CLASS](#).
2. Nastavení vztahu potomek-rodič mezi jednotlivými zdrojovými třídami a extrahovanou třídou využitím [SET PARENT](#).
3. Odstranění extrahovaného atributu ze všech zdrojových tříd kromě první z nich operací [REMOVE PROPERTY](#).
4. Přesunutí extrahovaného atributu do rodiče z první zdrojové třídy aplikací operace [PULL UP PROPERTY](#).



Obrázek 4.8: Ukázka aplikace operace EXTRACT SUPERCLASS

Kapitola 5

Syntaxe jazyka Ops

Syntaxe programovacího jazyka je formálně popsána gramatikou, která obsahuje gramatická pravidla definující přípustné jazykové konstrukce. Těmito pravidly jsme schopni přesně určit povolené tvary a sled po sobě jdoucích symbolů či skupin symbolů.

Definice gramatiky jazyka je obvykle zachycena nějakou z variant Backus-Naurovy formy, v našem případě použijeme rozšíření této formy, hojně využívané v literatuře [21], jejíž značení je následující:

- *neterminální* symboly jsou označeny kurzívou,
- pro označení terminálních symbolů je použito neproporcionální písmo,
- gramatická pravidla mají zápis: *Pozdrav* ::= ahoj *Jméno*, obsahující na levé straně, tzn. před symbolem ::=, vždy neterminál, na pravé straně se může vyskytovat kombinace terminálů a neterminálů,
- oddělení variant je vyjádřeno znakem svislé čáry |,
- symbol + vyjadřuje opakování nějakého neterminálu, např. výraz *Number*⁺ znamená 1 až *n* po sobě jdoucích výskytů neterminálu *Number*,
- term? značí nepovinný výskyt terminálu term,
- není-li řečeno jinak, pak znaky: {}-><. /: [] jsou terminálními symboly.

5.1 Vývoj syntaxe

Syntaxe jazyku Ops, pro popis sekvence operací, prošla během svého vývoje několika verzemi. Cílem vývoje bylo nalézt, co nejjednodušší zápis, který by byl zároveň dostatečně srozumitelný. Splněním těchto požadavků docílíme snadné orientace, v programu napsaném v jazyce Ops, a možnosti se, co v nejkratším čase naučit syntaxi jednotlivých operací.

První způsob zápisu, zachycený v ukázce kódu 5.1, je značně jednoduchý a kompaktní. Avšak jeho hlavní nevýhoda spočívá právě v jeho až příliš velké kompaktnosti, kdy není na první pohled zřejmé, jaký zápis reprezentuje kterou operaci. Typ použité operace je skryt pouze za speciální symboly (=, +=, !). Do jisté míry můžeme za výhody této syntaxe považovat použití symbolů <:, ->. Symbol dědičnosti <: našel základy v diagramech tříd standardu UML. Tento standard používá pro vyjádření vztahu potomek-rodíč

obdobný typ šipky \leftarrow . Právě podobnost symbolů může snadno v uživateli evokovat vztah dědičnosti. Naopak symbol šipky \rightarrow může v uživateli vyvolat pocit přesunu popisované věci, či její změnu. Z tohoto důvodu, je tento symbol využíván při přejmenování atributů nebo v zápisu operací sloužící k jejich přesun mezi různými třídami.

```

1 Record = class //Add Class
2 List = class //Add Class
3
4 Record += value : Integer //Add Property
5 Record += next : Record //Add Property
6 Record.value -> cargo //Rename Property
7
8 Record <: List //Set Parent
9 Record !<: //Remove Parent
10
11 !List //Remove Class

```

Ukázka kódu 5.1: Původní verze syntaxe jazyka Ops

Finální podoba syntaxe, viz ukázka 5.2, vychází z původní verze zápisu, ale snaží se eliminovat její nedostatky. Namísto použití speciálních symbolů, jsou v této verzi použita klíčová slova `add`, `rename`, `set`, `remove`, `...`, která snadněji vystihují typ použité operace.

```

1 add Record //Add Class
2 add List //Add Class
3
4 add Record.value : Integer //Add Property
5 add Record.next : Record //Add Property
6 rename Record.value -> cargo //Rename Property
7
8 set Record <: List //Set Parent
9 remove Record <: List //Remove Parent
10
11 remove List //Remove Class

```

Ukázka kódu 5.2: Finální verze syntaxe jazyka Ops

5.2 Finální verze syntaxe

Abychom se vyhnuli složitému definování neterminálů zachycující pojmenování tříd, atributů a dalších struktur vyskytujících se v modelu, definujeme je následovně:

- Primitivní typy jsou značeny *Primitive*, jedná se o množinu obsahující názvy primitivních typů: $\{Integer, Bool, Char, String\}$.
- Pojmenování tříd značíme *Class*, které je definováno na množině neprázdných řetězců, a navíc platí $Class \cap Primitive = \emptyset$ a $Top \notin Class$.
- Atributy tříd označíme jako *Property*. Jedná se o množinu neprázdných řetězců pro které platí $Property \cap Primitive = \emptyset$.
- Číselná konstanta *Number* je množina přirozených čísel sloužící k zadání hodnoty kardinalit daného atributu.

<i>Operation</i>	<pre> ::= AddClass RemoveClass AddProperty RemoveProperty RenameProperty MoveProperty PullUpProperty PushDownProperty SetParent RemoveParent ExtractClass ExtractSubClass ExtractSuperClass unit error Operation ; ? Operation </pre>
<i>AddClass</i>	<pre>::= add Class</pre>
<i>RemoveClass</i>	<pre>::= remove Class</pre>
<i>AddProperty</i>	<pre> ::= add Class.Property : TypeName add Class.Property : TypeName [Number] add Class.Property : TypeName [Number . . Number] </pre>
<i>RemoveProperty</i>	<pre>::= remove Class.Property</pre>
<i>RenameProperty</i>	<pre>::= rename Class.Property -> Property</pre>
<i>MoveProperty</i>	<pre>::= move Class.Property / Property -> Class</pre>
<i>PullUpProperty</i>	<pre>::= pullUp Class.Property</pre>
<i>PushDownProperty</i>	<pre>::= pushDown Class.Property</pre>
<i>SetParent</i>	<pre>::= set Class <: Class</pre>
<i>RemoveParent</i>	<pre>::= remove Class <: Class</pre>
<i>ExtractClass</i>	<pre>::= extract Class.Property / Property -> Class.Property</pre>
<i>ExtractSubClass</i>	<pre>::= extractSub Class.Property -> Class</pre>
<i>ExtractSuperClass</i>	<pre>::= extractSuper {Class⁺}.Property -> Class</pre>
<i>TypeName</i>	<pre> ::= Primitive Class </pre>

Gramatika 5.1: Finální podoba syntaxe jazyku Ops

5.3 Rozbor finální syntaxe

Program napsaný v jazyce Ops je tvořen sekvencí po sobě jdoucích operací, kdy jednotlivé operace mohou být odděleny středníkem. Zápis většiny operací má následující podobu:

`<operace> <Třída>.<atribut>`

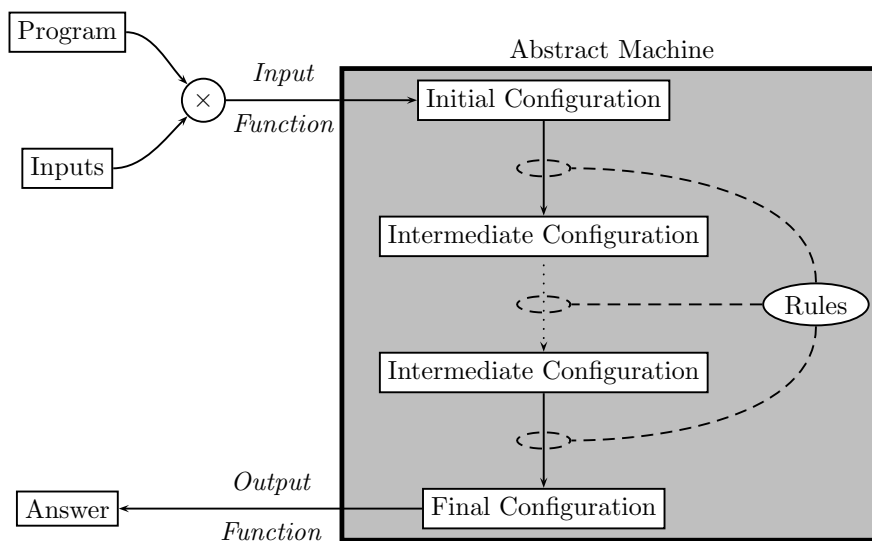
výraz na prvním místě vždy obsahuje zvolený typ operace `<operace>`, po něm následuje název zdrojové třídy `<Třída>` vlastní atribut pojmenovaný `<atribut>` na němž provedeme aplikaci dané operace. Pokud bychom chtěli výraz převést na větu, pak použití operace `remove Pozdrav.text` odpovídá větě *odstraň třídu Pozdrav atribut text*. Pro ostatní případy, kdy je zápis odlišný, je zde krátký přehled:

- `add Fruit`
Přidání nové třídy pojmenované *Fruit*.
- `add Fruit.weight : Integer[0..1]`
Přidání atributu *weight*, který je typu *Integer* s dolní kardinalitou 0 a horní kardinalitou 1, třídě *Fruit*.
- `rename Fruit.money -> price`
Přejmenování atributu *money*, patřící třídě *Fruit*, na *price*.
- `move Department.phone / address -> Address`
Přesunutí atributu *phone*, patřící zdrojové třídě *Department*, přes asociaci pojmenovanou *address* vedoucí ze zdrojové třídy do cílové třídy *Address*.
- `set Banana <: Fruit`
Nastavení vztahu potomek-rodíč mezi třídou *Banana* a třídou *Fruit*.
- `extract Department.city / addr -> Address.depart`
Vyextrahování atributu *city*, patřící zdrojové třídě *Department*, do cílové třídy *Address*. Mezi zdrojovou a cílovou třídou vznikne obousměrná asociace, která má na straně zdrojové třídy *Department* pojmenování *addr*, naopak na straně cílové třídy *Address* má pojmenování *depart*.
- `extractSub Fruit.price -> PricedFruit`
Vyextrahování atributu *price*, patřící zdrojové třídě *Fruit*, do všech přímých potomků zdrojové třídy včetně nové vzniklého potomka *PricedFruit*.
- `extractSuper {Apple, Banana}.weight -> Fruit`
Vyextrahování atributu *weight* ze tříd *Apple* a *Banana* do nové třídy *Fruit*, která je následně nastavena jako rodič obou zdrojových tříd.

Kapitola 6

Operační sémantika

Sémantika slouží k formalizování významu jednotlivých syntaktických konstruktů daného programovacího jazyka. Popisuje krok za krokem proces vyhodnocení syntakticky správných řetězců nad vybraným výpočetním modelem. V našem případě se budeme zabývat Operační sémantikou malého kroku využívající jako výpočetní model Abstraktní stroj [28].



Obrázek 6.1: Abstraktní stroj, obrázek převzat z [28, str. 48]

Abstraktní stroj si lze zjednodušeně představit jako vylepšený stavový automat, který během vyhodnocování syntaktického výrazu může nabývat různých stavů (konfigurací). Na počátku vyhodnocování výrazu je abstraktnímu stroji předána nejvhodnější **počáteční konfigurace**, jež byla vybrána **vstupní funkcí** na základě vstupního **programu** (výrazu) a ostatních **vstupů**. Poté, co byla zvolena počáteční konfigurace, dojde k samotnému „spuštění“ abstraktního stroje. Ten podle dostupné sady **přepisovacích pravidel** přechází (přepisuje) z jedné konfigurace do druhé. Vyhodnocení syntaktického výrazu abstraktním strojem končí, když je současná konfigurace označena jako finální nebo neexistuje vhodné přepisovací pravidlo, které by jí přepsalo na jinou konfiguraci. Posledním krokem

vyhodnocení syntaktického výrazu je převedení **finální konfigurace**, v níž se abstraktní stroj nachází, využitím **výstupní funkce** na **odpověď**. Forma odpovědi závisí na definici výstupní funkce, může se jednat o cokoliv od čísel až po složité algebraické struktury.

Sada přepisovacích pravidel definuje binární přepisovací relaci \Rightarrow nad konfiguracemi. Každé z pravidel má předpoklady (nad čarou), které musí být splněny, aby platil jejich závěr (pod čarou).

$$\frac{\text{předpoklady}}{\text{závěr}} \quad (6.1)$$

Uvědomme si, že přepisovací pravidla vlastně určují za jakých předpokladů dochází k přepsání jedné konfigurace na druhou konfiguraci.

6.1 Značení a zápis

V následujícím textu budeme uvažovat, že pro jednotlivé metaproměnné platí:

$$\begin{aligned} o &\in \text{Operation} \\ c, d, e &\in \text{Class} \\ p, q, r &\in \text{Property} \\ k, l, m, n &\in \text{Number} \\ s, t &\in \text{TypeName} \end{aligned} \quad (6.2)$$

6.2 Operační sémantika

Definice 6.1: *Operační sémantika malého kroku*, je uspořádaná pětice $(CF, \Rightarrow, FC, IF, OF)$, kde:

- CF je množina konfigurací,
- $\Rightarrow \subseteq CF \times CF$ je binární relace nad konfiguracemi,
- $FC \subseteq CF$ je množina finálních konfigurací,
- $IF : \text{Program} \times \text{Input} \rightarrow CF$ je vstupní funkce,
- $OF : FC \rightarrow \text{Output}$ je výstupní funkce.

Následující podkapitoly formálně specifikují jednotlivé prvky pětice operační sémantiky malého kroku pro potřeby našeho jazyku.

6.3 Struktura \mathcal{S}

Struktura \mathcal{S} slouží k zachycení stavu výpočtu operační sémantiky definované níže. Strukturu si lze představit jako tabulku, kde každý její řádek odpovídá jedné třídě a informacím jí se týkající (kdo je její rodič, jaké má atributy).

Před samotným formálním definováním struktury, nejprve zavedeme pomocnou množinu atributů *PropertySet*:

Definice 6.2: *PropertySet* je množina všech podmnožin čtveřic (p, t, n, m) reprezentujících atribut pojmenovaný $p \in Property$, který je typu $t \in TypeName$ s dolní n a horní m kardinalitou omezující počty instancí, kde $n, m \in Number$:

$$PropertySet = \mathcal{P}(Property \times TypeName \times Number \times Number) \quad (6.3)$$

Definice 6.3: *Strukturu* \mathcal{S} definujeme jako mapování z *Class* do $(Class \cup \{Top\}) \times PropertySet$:

$$\mathcal{S}(c) = (d, \{p_1 : t_1[n_1..m_1], \dots, p_j : t_j[n_j..m_j]\}), \quad \text{kde } j \geq 0 \quad (6.4)$$

kde $d \in Class \cup \{Top\}$ je rodič třídy $c \in Class$. Zápis $p_i : t_i[n_i..m_i]$ odpovídá čtveřici (p_i, t_i, n_i, m_i) obsahující informace o atributu p_i .

Definice 6.4: Mějme $\mathcal{S}(c) = (d, \{p_1 : t_1[n_1..m_1], \dots, p_j : t_j[n_j..m_j]\})$ pro nějaké přirozené číslo $j \geq 0$, zkráceně psáno $\mathcal{S}(c) = (d, ps)$, kde $ps \in PropertySet$. Pak definujeme následující pomocné funkce/zápisy:

- $\mathcal{S}(c.p_i)$ slouží k přístupu k definovanému atributu p_i existující třídy c , výsledkem je čtveřice $p_i : t_i[n_i..m_i]$,
- $prop_{\mathcal{S}}(c) = \{p_1, \dots, p_j\}$, vrací názvy atributů třídy c ve struktuře \mathcal{S} ,
- $par_{\mathcal{S}}(c) = d$, vrací přímého rodiče třídy c ve struktuře \mathcal{S} ,
- $sub_{\mathcal{S}}(c) = \{d \in Class \mid par_{\mathcal{S}}(d) = c\}$, vrací množinu přímých potomků třídy c .

6.4 Přepisovací pravidla

Definice 6.5: *Konfigurace* cf operační sémantiky malého kroku je definovaná jako dvojice $(\mathcal{S}, o) = cf \in Structure \times Operation$.

Definice 6.6: *Přepisovací relaci* \Rightarrow nad konfiguracemi cf definovanou níže uvedenými pravidly zapisujeme jako $(\mathcal{S}, o_1) \Rightarrow (\mathcal{S}', o_2)$, kde o_i reprezentuje operaci a \mathcal{S} , resp. \mathcal{S}' , odpovídá struktuře před, resp. po, vykonání operace o_1 .

Definice 6.7: *Počáteční konfigurace* odpovídá konfiguraci ve tvaru: (\emptyset, o_1) , kde $o_1 \in Operation$, a *finální konfigurace* odpovídají tvaru: (\mathcal{S}, o_2) , kde \mathcal{S} je struktura po vykonání všech operací z počáteční konfigurace a o_2 je jednotková operace `unit` nebo chybová operace `error`.

V následující části textu věnujeme prostor pro definování jednotlivých přepisovacích pravidel. Jelikož budeme v kapitole 7 definovat typová pravidla, která do značné míry odpovídají přepisovacím pravidlům ve smyslu popisované operace (*Operation*), zavedeme pro pojmenování jednotlivých přepisovacích pravidel prefix E, a pro typová pravidla prefix T. Pojmenování pravidla `E-AddCls`, pak odpovídá přepisovacímu pravidlu operace pro přidání nové třídy.

Dále poznamenejme, že kvůli úspoře místa předpokládáme následující: každou konfiguraci cf , kterou nelze přepsat na jinou konfiguraci cf' uvedenou v níže definovaných přepisovacích pravidlech, protože jejich předpoklady nejsou splněny, automaticky přepíšeme na konfiguraci $(\mathcal{S}, error)$ s chybovou operací `error`. Ve výsledku to znamená, že implicitně předpokládáme existenci chybových přepisovacích pravidel obsahující všechny možné kombinace předpokladů níže uvedených pravidel.

$$\frac{}{(\mathcal{S}, \text{add } c) \Rightarrow (\mathcal{S}[c \mapsto (\text{Top}, \emptyset)], \text{unit})} \quad (\text{E-ADDCLS})$$

Přepisovací pravidlo **E-ADDCLS** reprezentuje aplikaci operace **ADD CLASS**. Pravidlo říká, že se výraz $\text{add } c$ nad nějakou strukturou \mathcal{S} přepíše na výraz unit , a původní struktura \mathcal{S} je zápisem $\mathcal{S}[c \mapsto (\text{Top}, \emptyset)]$ rozšířena o novou prázdnou třídu c , která má rodiče nastaveného na pomocnou třídu Top a množina atributů třídy c je prázdná.

$$\frac{}{(\mathcal{S}, \text{remove } c) \Rightarrow (\mathcal{S} \setminus (c, \mathcal{S}(c)), \text{unit})} \quad (\text{E-REMCLS})$$

Aplikace operace **REMOVE CLASS** je vyjádřena přepisovacím pravidlem **E-REMCLS**. Toto pravidlo vyhodnotí výraz $\text{remove } c$ nad strukturou \mathcal{S} tím, že ze struktury odstraní třídu c včetně informací jí se týkajících, a přepíše ho na jednotkovou operaci unit .

$$\frac{}{(\mathcal{S}, \text{add } c.p : t) \Rightarrow (\mathcal{S}, \text{add } c.p : t[0..1])} \quad (\text{E-ADDPROP1})$$

$$\frac{}{(\mathcal{S}, \text{add } c.p : t[n]) \Rightarrow (\mathcal{S}, \text{add } c.p : t[0..n])} \quad (\text{E-ADDPROP2})$$

$$\frac{\mathcal{S}(c) = (d, ps)}{(\mathcal{S}, \text{add } c.p : t[n..m]) \Rightarrow (\mathcal{S}[c \mapsto (d, ps \cup \{p : t[n..m]\})], \text{unit})} \quad (\text{E-ADDPROP3})$$

Pravidla **E-ADDPROP1** a **E-ADDPROP2** jsou pouze zjednodušením přepisovacího pravidla **E-ADDPROP3**, kdy dolní a horní kardinality jsou implicitně nastaveny. Předpoklad **E-ADDPROP3** obsahuje zápis pro získání rodiče d třídy c a její množiny atributů ps . Vyhodnocením pravidel **E-ADDPROP1** až **E-ADDPROP3** dojde k rozšíření původní množiny atributů ps třídy c o nový atribut $p : t[n..m]$ a k následnému přepsání na jednotkovou operaci.

$$\frac{\mathcal{S}(c) = (d, ps)}{(\mathcal{S}, \text{remove } c.p) \Rightarrow (\mathcal{S}[c \mapsto (d, ps \setminus \{\mathcal{S}(c.p)\})], \text{unit})} \quad (\text{E-REMPROP})$$

Konfigurace $(\mathcal{S}, \text{remove } c.p)$ je v přepisovacím pravidlu **E-REMPROP** vyhodnocena na dvojici obsahující jednotkovou operaci a strukturu \mathcal{S} obsahující aktualizovanou informaci o třídě c , z jejíž množiny atributů ps byl odstraněn atribut p .

$$\frac{\mathcal{S}(c.p) = p : t[n..m]}{(\mathcal{S}, \text{rename } c.p \rightarrow q) \Rightarrow (\mathcal{S}, \text{remove } c.p; \text{add } c.q : t[n..m])} \quad (\text{E-NAMPROP})$$

Výraz $\text{rename } c.p \rightarrow q$ nad strukturou \mathcal{S} se pravidlem **E-NAMPROP** přepíše na sekveci operací **REMOVE PROPERTY** a **ADD PROPERTY**, čímž dojde k odstranění původního atributu p ze třídy c a k jeho následnému znovupřidání s novým pojmenováním q .

$$\frac{\mathcal{S}(c.p) = p : t[n..m]}{(\mathcal{S}, \text{move } c.p / q \rightarrow d) \Rightarrow (\mathcal{S}, \text{remove } c.p; \text{add } d.p : t[n..m])} \quad (\text{E-MOVPROP})$$

Na základě aplikace operace **MOVE PROPERTY** dojde vyhodnocením přepisovacího pravidla **E-MOVPROP** k přesunutí atributu p ze třídy c do třídy d tím, že je konfigurace $(\mathcal{S}, \text{move } c.p / q \rightarrow d)$ přepsána na novou konfiguraci, obsahující původní strukturu \mathcal{S} a sekvenci operací **REMOVE PROPERTY** a **ADD PROPERTY**, která je následně vyhodnocena.

$$\frac{\text{par}_{\mathcal{S}}(c) = d \quad \mathcal{S}(c.p) = p : t[n..m]}{(\mathcal{S}, \text{pullUp } c.p) \Rightarrow (\mathcal{S}, \text{remove } c.p; \text{add } d.p : t[n..m])} \quad (\text{E-PULLPROP})$$

$$\frac{\mathcal{S}(c.p) = p : t[n..m]}{(\mathcal{S}, \text{pushDown } c.p) \Rightarrow (\mathcal{S}, \text{remove } c.p; \Delta)} \quad (\text{E-PUSHPROP})$$

kde $\Delta = \text{add } c_1.p : t[n..m]; \dots; \text{add } c_j.p : t[n..m]$ pro $c_i \in \text{sub}_{\mathcal{S}}(c)$

Přepisovací pravidla **E-PULLPROP** a **E-PUSHPROP** zachycují použití operací **PULL UP PROPERTY** a **PUSH DOWN PROPERTY** sloužící k přesunu atributů v rámci hierarchie, kdy přesouváme atributy zdrojové třídy c směrem nahoru, resp. dolů, do jejího rodiče d , resp. do všech jejích potomků c_i z množiny $\text{sub}_{\mathcal{S}}(c)$. V obou případech dojde k přepsání prvotní konfigurace na novou konfiguraci cf obsahující sekvenci operací a strukturu, která zůstane neměnná, dokud není sekvence operací vyhodnocena.

$$\frac{\mathcal{S}(c) = (e, ps)}{(\mathcal{S}, \text{set } c <: d) \Rightarrow (\mathcal{S}[c \mapsto (d, ps)], \text{unit})} \quad (\text{E-SETPAR})$$

E-SETPAR říká, že se výraz $\text{set } c <: d$ nad strukturou \mathcal{S} přepíše na jednotkovou operaci, a ve struktuře \mathcal{S} je zápisem $\mathcal{S}[c \mapsto (d, ps)]$ změněn původní rodič e třídy c na třídu d . Množina atributů ps třídy c zůstane nezměněna.

$$\frac{\mathcal{S}(c) = (d, ps)}{(\mathcal{S}, \text{remove } c <: d) \Rightarrow (\mathcal{S}[c \mapsto (Top, ps)], \text{unit})} \quad (\text{E-REMPAR})$$

Přepisovací pravidlo **E-REMPAR**, obdobně jako **E-SETPAR**, slouží ke změně rodiče třídy c s tím rozdílem, že je třídě nastaven rodič na pomocnou třídu Top . Uvědomme si, že pomocnou třídu Top nelze ručně vytvořit ani jí použít pomocí syntaxe jazyka, protože neleží v množině všech pojmenování tříd $Class$. Top je rezervované slovo jazyka a slouží jako záložka, obdobně jako třída $Object$ v programovacím jazyku Java, která je implicitně rodičem všech tříd, jež nemají explicitně nastaveného jiného rodiče.

$$\frac{}{(\mathcal{S}, \text{extract } c.p / q \rightarrow d.r) \Rightarrow (\mathcal{S}, \text{add } d; \text{add } d.r : c[1..1]; \text{add } c.q : d[1..1]; \text{move } c.p / q \rightarrow d)} \quad (\text{E-EXTCLS})$$

$$\frac{}{(\mathcal{S}, \text{extractSub } c.p \rightarrow d) \Rightarrow (\mathcal{S}, \text{add } d; \text{set } d <: c; \text{pushDown } c.p)} \quad (\text{E-EXTSUB})$$

Pravidla **E-EXTCLS**, **E-EXTSUB** zachycují aplikaci operací **EXTRACT CLASS**, **EXTRACT SUBCLASS** sloužící k vyextrahování atributu p ze zdrojové třídy c do nové speciální třídy d , která je v případě operace **EXTRACT SUBCLASS** potomkem zdrojové třídy. Naopak operací

EXTRACT CLASS vznikne mezi speciální třídou a zdrojovou třídou pouze obousměrná asociace. Prvotní konfigurace ve všech zmíněných pravidlech je přepsána na novou konfiguraci obsahující sekvenci dílčích operací, které po svém vykonání změní odpovídajícím způsobem strukturu \mathcal{S} .

$$\frac{}{(\mathcal{S}, \text{extractSuper } \{c_1 \dots c_j\}.p \rightarrow d \Rightarrow (\mathcal{S}, \text{add } d; \Delta; \text{pullUp } c_1.p))} \quad (\text{E-EXTSUPER})$$

kde $\Delta = \text{set } c_1 <: d; \dots; \text{set } c_i <: d; \text{remove } c_2.p; \dots; \text{remove } c_j.p$

Aplikace operace **EXTRACT SUPERCLASS** je vyjádřena pravidlem **E-EXTSUPER**, které vyextrahuje atribut p ze zdrojových tříd c_i do nové třídy d . Nejprve je vytvořena třída d , jež je nastavena jako rodič všech zdrojových tříd. Následně využitím operací **REMOVE PROPERTY** a **PULL UP PROPERTY** dojde k přesunutí samotného atributu p do třídy d .

$$\frac{(\mathcal{S}, o_1) \Rightarrow (\mathcal{S}', o'_1)}{(\mathcal{S}, o_1; o_2) \Rightarrow (\mathcal{S}', o'_1; o_2)} \quad (\text{E-SEQ})$$

Přepisovací pravidlo **E-SEQ** slouží k vyhodnocení sekvence operací. **E-SEQ** říká, že vyhodnocení sekvence operací probíhá zleva, kdy vyhodnocujeme vždy nejlevější operaci dokud je to možné.

$$\frac{}{(\mathcal{S}, \text{unit}; o_2) \Rightarrow (\mathcal{S}, o_2)} \quad (\text{E-SEQUNIT})$$

$$\frac{}{(\mathcal{S}, \text{error}; o_2) \Rightarrow (\mathcal{S}, \text{error})} \quad (\text{E-SEQERR})$$

E-SEQUNIT využijeme, vyskytne-li se v sekvenci operací jednotková operace **unit**, pak jí můžeme přeskočit a pokračovat vyhodnocováním operace o_2 po ní následující. Naopak význam pravidla **E-SEQERR** spočívá v přepsání celé sekvence operací na konfiguraci $(\mathcal{S}, \text{error})$. Jelikož je tato konfigurace finální, abstraktní stroj zastaví proces vyhodnocování. Pravidlo **E-SEQERR** najde uplatnění v případech, kdy sekvence obsahuje chybovou operaci **error**, pak nemá smysl pokračovat ve vyhodnocování po ní následujících operací, ale rovnou celou sekvenci přepsat na finální chybovou konfiguraci $(\mathcal{S}, \text{error})$ a zastavit tak vyhodnocování abstraktním strojem.

6.5 Vstupní a výstupní funkce

Definice 6.8: *Vstupní funkci* IF definujeme jako mapování z *Operation* do množiny všech konfigurací $CF = \text{Structure} \times \text{Operation}$:

$$IF(o) = (\emptyset, o) \quad \text{kde } o \in \text{Operation} \quad (6.5)$$

Definice 6.9: *Výstupní funkci* OF definujeme jako mapování z množiny všech finálních konfigurací $FC = \text{Structure} \times \text{Operation}$ do *Structure*:

$$OF(\mathcal{S}, o) = \begin{cases} \mathcal{S} & \text{když } o = \text{unit} \\ \emptyset & \text{jinak} \end{cases} \quad (6.6)$$

Kapitola 7

Typový systém

Účelem typového systému je předcházení vzniku chyb během vyhodnocení programu abstraktním strojem [1]. Různé části programu vyžadují různé druhy hodnot či vlastností, které musejí být vhodně zvoleny, aby byl celý program správně vyhodnocen a nedošlo k zaseknutí abstraktního stroje. Součástí typového systému jsou:

- **Typová pravidla** slouží k ověření požadovaných vlastností a přiřazení příslušných typů programu včetně jeho dílčím částem. Za pomoci typů jsme schopni typově zkontrolovat správnost uživatelského programu bez nutnosti jej vyhodnocovat, což je zvlášť výhodné, je-li program velmi složitý a jeho vyhodnocení by bylo příliš výpočetně náročné, a navíc můžeme předejít samotnému vyhodnocování programu, který nesplňuje požadované vlastnosti.
- **Typový kontext (prostředí)** reprezentuje strukturu pro uložení pomocných informací typovými pravidly. Kontext využijeme během procesu otypování programu, např. pro uložení informace jaký typ náleží proměnné.
- **Typová relace** je definovaná typovými pravidly a její prvky přiřazují syntakticky správným výrazům typ s ohledem na typové prostředí.

7.1 Typový systém

Definice 7.1: *Typovou relaci* $\Gamma \vdash o : T$, vyjádřenou níže uvedenými pravidly, definujeme jako trojici mezi kontextem, operacemi a typy. Význam zápisu $\Gamma \vdash o : T$ odpovídá, že v typovém prostředí Γ má operace o typ T .

Typový kontext Γ bude v našem případě sloužit k uložení pozorovatelných změn (akcí) dané operace o . Budeme tak schopni zachytit, že vykonáním operace `add c` dojde k rozšíření typového kontextu o přidání nové třídy, a nebo operací `remove c.p` k odstranění nějakého atributu z kontextu Γ . Akce nám umožňují zkontrolovat správnost programu (sekvence operací) bez nutnosti jej spustit, protože ze zadaného programu jsme schopni zrekonstruovat kontext Γ a ověřit jeho validnost. Pomocí akcí můžeme například odhalit, že odstranění neexistující třídy způsobí chybu, protože v typovém kontextu není uložena informace o existenci dané neexistující třídy.

Definice 7.2: Kontext Γ a typy T definujeme induktivně gramatikami:

$$\begin{aligned} T & ::= \text{Operation} \\ \Gamma & ::= \emptyset \mid \Gamma, c \mid \Gamma, c.p : t[n..m] \mid \Gamma, c <: d \end{aligned}$$

Definice 7.3: Mějme kontext Γ . Pak definujeme pomocné funkce s následujícím významem:

- $prop_{\Gamma}(c) = \{p \mid c.p : t[n..m] \in \Gamma\}$, vrací množinu názvů atributů třídy c ,
- $parprop_{\Gamma}(c) = \begin{cases} \emptyset & \text{pokud } c \notin \Gamma \\ prop_{\Gamma}(c) \cup parprop_{\Gamma}(par_{\Gamma}(c)) \end{cases}$
vrací množinu všech atributů třídy c a její rodičů,
- $allprop_{\Gamma}(c) = parprop_{\Gamma}(c) \cup (\bigcup \{allprop_{\Gamma}(d) \mid par_{\Gamma}(d) = c\})$, vrací množinu všech atributů třídy c , její potomků a rodičů,
- $par_{\Gamma}(c) = \begin{cases} d & \text{pokud } \exists d : c <: d \in \Gamma \\ Top \end{cases}$ vrátí jednoho rodiče třídy c ,
- $sub_{\Gamma}(c) = \{d \in Class \mid d <: c \in \Gamma\}$, vrací přímé potomky třídy c ,
- $preds_{\Gamma}(c) = \begin{cases} \emptyset & \text{pokud } c = Top \\ \{c\} \cup preds_{\Gamma}(par_{\Gamma}(c)) \end{cases}$
vrací množinu všech předků třídy c včetně samotné třídy.

7.2 Typová pravidla

Abychom se vyhnuli duplicitnímu definování typových pravidel pro jednotlivé operace (o) a pro jejich sekvence ($o_1; o_2$), předpokládáme, že zápis $\Gamma \vdash o_1; o_2 : T$, sloužící k otypování sekvence operací lze použít i pro jednotlivé operace, nahrazením druhé operace o_2 speciálním symbolem ϵ . Uvědomme si, že symbol ϵ nepatří do gramatiky jazyka ani není operací, slouží pouze pro zkrácení zápisu. Pomocné typovací pravidlo T-EPSILON pro ϵ je definováno následovně:

$$\frac{}{\Gamma \vdash \epsilon : \text{Operation}} \quad (\text{T-EPSILON})$$

Dále poznamenejme, že hodnoty metaproměnných *volně vyskytujících se* v předpokladech typových pravidlech mohou být libovolné, avšak v rámci množiny na níž jsou definovány. Například předpoklady typového pravidla **T-REMCls** obsahují volný výskyt metaproměnných d, q, n, m , které nejsou vázány na výraz obsažený v závěru tohoto pravidla. Předpokládáme, že hodnota volné metaproměnné d obsahuje libovolné pojmenování třídy z množiny *Class*, q obsahuje libovolné pojmenování atributu z množiny *Property*. Analogicky i pro metaproměnné n, m , které jsou definovány na množině *Number*.

$$\frac{c \notin \Gamma \quad \Gamma, c \vdash o : \text{Operation}}{\Gamma \vdash \text{add } c; o : \text{Operation}} \quad (\text{T-ADDCLS})$$

Typové pravidlo **T-ADDCLS** odpovídá předpokladům operace **ADD CLASS**. Podle těchto předpokladů, se nově přidávaná třída c nesmí vyskytovat v prostředí Γ . Operace o následující po $\text{add } c$ je otypována nad rozšířeným prostředím Γ do něhož byla zápisem Γ, c přidána třída c . Jsou-li splněny oba předpoklady **T-ADDCLS**, má uvedená sekvence operací typ *Operation* nad typovým prostředím Γ .

$$\frac{c \in \Gamma \quad \text{prop}_{\Gamma}(c) = \text{sub}_{\Gamma}(c) = \emptyset \quad d.q : c[n..m] \notin \Gamma \quad \Gamma \setminus \{c\} \vdash o : \text{Operation}}{\Gamma \vdash \text{remove } c; o : \text{Operation}} \quad (\text{T-REMCls})$$

T-REMCls reprezentuje ověření předpokladů operace pro odstranění třídy z modelu. Na základě předpokladů **REMOVE CLASS** musí třída c existovat v typovém prostředí Γ a její množina atributů a množina potomků musí být prázdná. Navíc musí být splněno, že typ (*TypeName*) libovolného atributu v prostředí Γ neodpovídá právě odstraňované třídě. Operace o je otypována nad upraveným prostředím z něhož byla, zápisem $\Gamma \setminus \{c\}$, odstraněna třída c .

$$\frac{\Gamma \vdash \text{add } c.p : t[0..1]; o : \text{Operation}}{\Gamma \vdash \text{add } c.p : t; o : \text{Operation}} \quad (\text{T-ADDPROP1})$$

$$\frac{\Gamma \vdash \text{add } c.p : t[0..n]; o : \text{Operation}}{\Gamma \vdash \text{add } c.p : t[n]; o : \text{Operation}} \quad (\text{T-ADDPROP2})$$

$$\frac{c \in \Gamma \quad (t \in \Gamma \vee t \in \text{Primitive}) \quad m \geq n \geq 0 \quad m > 0 \quad p \notin \text{allprop}_{\Gamma}(c) \quad \Gamma, c.p : t[n..m] \vdash o : \text{Operation}}{\Gamma \vdash \text{add } c.p : t[n..m]; o : \text{Operation}} \quad (\text{T-ADDPROP3})$$

Typová pravidla **T-ADDPROP1** až **T-ADDPROP3** slouží k otypování operace **ADD PROPERTY** pro přidání atributu p dané třídě c . Z předpokladů pravidel **T-ADDPROP1** a **T-ADDPROP2** vyplývá, že jsou pouze zjednodušením pravidla **T-ADDPROP3**, kdy dolní a horní kardinality jsou implicitně nastaveny. Na základě předpokladů operace **ADD PROPERTY**, musí existovat třída jíž je přidáván nový atribut p , typ atributu p musí existovat v kontextu Γ jako třída nebo se jedná primitivní typ *Primitive*. Dále musí být splněno, že se pojmenování atributu p nenachází v dané třídě, jejích potomcích ani předcích, jinak by vznikl duplicitní atribut. Tuto skutečnost zajistíme využitím pomocné funkce $\text{allprop}_{\Gamma}(c)$ sloužící k získání množiny všech atributů třídy c , jejích potomků a předků.

$$\frac{c, c.p : t[n..m] \in \Gamma \quad \Gamma \setminus \{c.p : t[n..m]\} \vdash o : \text{Operation}}{\Gamma \vdash \text{remove } c.p; o : \text{Operation}} \quad (\text{T-REMPROP})$$

Předpoklady operace **REMOVE PROPERTY** jsou zajištěny typovým pravidlem **T-REMPROP**, které zkontroluje existenci třídy z níž je odebírán atribut p a také existenci samotného atributu. Všimněme si, že typ a kardinality atributu mohou nabývat libovolné hodnoty.

$$\frac{c, c.p : t[n..m] \in \Gamma \quad q \notin \text{allprop}_{\Gamma}(c) \quad \Gamma \setminus \{c.p : t[n..m]\}, c.q : t[n..m] \vdash o : \text{Operation}}{\Gamma \vdash \text{rename } c.p \rightarrow q; o : \text{Operation}} \quad (\text{T-NAMPROP})$$

T-NAMPROP slouží ke kontrole předpokladů operace **RENAME PROPERTY**. Dle předpokladů je nutné ověřit, zda přejmenováváný atribut a samotná třída existují v prostředí Γ . Dále je nutné zajistit, že se nový název atribut nenachází ve třídě c ani v jejích předcích a rodičích využitím funkce $allprop_{\Gamma}(c)$, abychom zabránili duplicitě.

$$\frac{c, d, c.p : t[n..m], c.q : d[1..1] \in \Gamma \quad p \notin allprop_{\Gamma}(d) \quad \Gamma \setminus \{c.p : t[n..m]\}, d.p : t[n..m] \vdash o : Operation}{\Gamma \vdash move\ c.p / q \rightarrow d; o : Operation} \quad (\text{T-MOVPROP})$$

Předpoklady operace pro přesun atributů mezi třídami jsou ověřeny typovým pravidlem **T-MOVPROP**. Toto pravidlo kontroluje existenci cílové třídy d , atributu p a samotné zdrojové třídy c , ale také existenci asociace $c.q : d[1..1]$ mezi třídou c a cílovou třídou d . Aby bylo možné atribut přesunout, je nutné zajistit, že se jeho pojmenování nenachází v množině všech atributů $allprop_{\Gamma}(d)$ cílové třídy d .

$$\frac{c, c.p : t[n..m], c <: d \in \Gamma \quad p \notin allprop_{\Gamma \setminus \{c <: d\}}(d) \quad \Gamma \setminus \{c.p : t[n..m]\}, d.p : t[n..m] \vdash o : Operation}{\Gamma \vdash pullUp\ c.p; o : Operation} \quad (\text{T-PULLPROP})$$

$$\frac{c, c.p : t[n..m] \in \Gamma \quad sub_{\Gamma}(c) \neq \emptyset \quad \Gamma \setminus \{c.p : t[n..m]\}, d_1.p : t[n..m], \dots, d_j.p : t[n..m] \vdash o : Operation}{\Gamma \vdash pushDown\ c.p; o : Operation} \quad (\text{T-PUSHPROP})$$

kde $d_i \in sub_{\Gamma}(c)$

Typová pravidla **T-PULLPROP** a **T-PUSHPROP** zachycují kontrolu předpokladů operací **PULL UP PROPERTY** a **PUSH DOWN PROPERTY**. Tyto operace najdou uplatnění při přesunu atributů v rámci hierarchie. Obě pravidla ověřují existenci přesouvaného atributu a třídy v níž se atribut nachází.

Pravidlo **T-PULLPROP** navíc kontroluje existenci rodiče d zdrojové třídy c a dále zápisem $allprop_{\Gamma \setminus \{c <: d\}}(d)$ ověřuje, že sourozenci třídy c neobsahují atribut p , který bude přesunut do jejich společného rodiče. Kdyby atribut v sourozencích existoval, pak by došlo při přesunu směrem nahoru do rodiče k duplicitě.

Přesun směrem dolů do všech potomků d_i zdrojové třídy, operací **PUSH DOWN PROPERTY**, vyžaduje včetně existence zdrojové třídy a přesouvaného atributu i existenci samotných potomků d_i . Není nutné ověřovat, zdali libovolný z potomků neobsahuje přesouvaný atribut, protože ostatní typová pravidla nám zaručují, že prostředí Γ neobsahuje duplicitu.

$$\frac{c, d \in \Gamma \quad c \notin preds_{\Gamma}(d) \quad allprop_{\Gamma \setminus \{c <: d\}}(c) \cap parprop_{\Gamma}(d) = \emptyset \quad \Gamma, c <: d \vdash o : Operation}{\Gamma \vdash set\ c <: d; o : Operation} \quad (\text{T-SETPAR})$$

T-SETPAR ověřuje předpoklady **SET PARENT** sloužící k nastavení rodiče třídě. Aby bylo možné třídě c nastavit jako rodiče třídu d , musí obě třídy existovat v prostředí Γ a nesmí nastavením vzniknout cyklus v hierarchii tříd. Vzniku cyklu zamezíme použitím pomocné

funkce $preds_{\Gamma}(d)$, která vrátí množinu obsahující všechny předky třídy d včetně samotné třídy d . Pokud by třída c byla obsažena v této množině, pak by došlo ke vzniku cyklu. Dále je nutné zamezit, aby nastavením rodiče nevznikly duplicitní atributy. Použitím pomocné funkce $allprop$ nad upraveným typovým prostředím $\Gamma \setminus \{c <: d\}$ získáme množinu názvů všech atributů třídy c a jejích potomků, a funkcí $parprop_{\Gamma}(d)$ dostaneme množinu obsahující pojmenování atributů třídy d a jejích předků. Obě množiny musí být disjunktní, tedy nesmí mít žádné společné prvky, jinak by vznikly duplicity.

$$\frac{c, d, c <: d \in \Gamma \quad \Gamma \setminus \{c <: d\} \vdash o : Operation}{\Gamma \vdash \text{remove } c <: d; o : Operation} \quad (\text{T-REMPAR})$$

Předpoklady pro odstranění rodiče d třídy c jsou zkontrolovány pravidlem [T-REMPAR](#), které ověří existenci obou tříd a vztahu potomek-rodič mezi danými třídami v prostředí Γ . Operace o , následující po výrazu $\text{remove } c <: d$, je poté otypována nad upraveným prostředím z něhož byl zápisem $\Gamma \setminus \{c <: d\}$ odstraněn vztah potomek-rodič mezi třídami c a d .

$$\frac{\begin{array}{l} c, c.p : t[n..m] \in \Gamma \quad d \notin \Gamma \quad q \notin allprop_{\Gamma}(c) \\ \Gamma' = \Gamma \setminus \{c.p : t[n..m]\}, d, c.q : d[1..1], d.r : c[1..1] \\ \Gamma', d.p : t[n..m] \vdash o : Operation \end{array}}{\Gamma \vdash \text{extract } c.p / q \rightarrow d.r; o : Operation} \quad (\text{T-EXTCLS})$$

Typové pravidlo [T-EXTCLS](#) odpovídá předpokladům operace [EXTRACT CLASS](#). Na základě těchto předpokladů, musí existovat atribut p a třída c ze které provádíme extrahování atributu. Dále je nutné ověřit, že prostředí Γ neobsahuje třídu d do níž bude provedena extrakce. Jelikož po vyextrahování vznikne obousměrná asociace mezi oběma třídami, musíme zajistit, aby zdrojová třída c a ani její potomci či předci neobsahovali atribut pojmenovaný q , který bude reprezentovat směr asociace z třídy c do třídy d .

$$\frac{\begin{array}{l} c, c.p : t[n..m] \in \Gamma \quad d \notin \Gamma \quad \Gamma' = \Gamma \setminus \{c.p : t[n..m]\}, d, d <: c \\ \Gamma', d_1.p : t[n..m], \dots, d_j.p : t[n..m] \vdash o : Operation \end{array}}{\Gamma \vdash \text{extractSub } c.p \rightarrow d; o : Operation} \quad (\text{T-EXTSUB})$$

kde $d_i \in sub_{\Gamma'}(c)$

[T-EXTSUB](#) zachycuje nutné podmínky operace [EXTRACT SUBCLASS](#) pro extrakci atributu p do speciálního potomka d zdrojové třídy c . Extrahovaný atribut p a třída c , která jej obsahuje, musí být přítomny v kontextu Γ . Dále se v kontextu nesmí vyskytovat třída d do níž provádíme extrakci. Otypování operace následující po extractSub uděláme nad aktualizovaným typovým prostředím, kvůli přehlednosti provedeme jeho aktualizaci ve dvou fázích. V první fázi z kontextu Γ odstraníme extrahovaný atribut, přidáme do něho nově vzniklou třídu d a vztah potomek-rodič mezi třídami d a c . Výsledkem této fáze „uložíme“ do prostředí Γ' . Druhá fáze navazuje na první a obsahuje informace o přesunu extrahovaného atributu do všech potomků d_i třídy c , které jsme získali funkcí $sub_{\Gamma'}(c)$.

$$\begin{array}{c}
c_1, \dots, c_j \in \Gamma \quad d, c_1 <: e, \dots, c_j <: e \notin \Gamma \quad |\{c_1, \dots, c_j\}| = j \\
c_1.p : t_1[n_1..m_1], \dots, c_j.p : t_1[n_1..m_1] \in \Gamma \\
\Gamma' = \Gamma \setminus \{c_1.p : t_1[n_1..m_1], \dots, c_j.p : t_1[n_1..m_1]\} \\
\Gamma', d, c_1 <: d, \dots, c_j <: d, d.p : t_1[n_1..m_1] \vdash o : Operation \\
\hline
\Gamma \vdash \text{extractSuper } \{c_1 \dots c_j\}.p \rightarrow d; o : Operation
\end{array}
\quad (\text{T-EXTSUPER})$$

Předpoklady pro použití operace **EXTRACT SUPERCLASS** jsou vyjádřeny typovým pravidlem **T-EXTSUPER**. Dle definovaných předpokladů je nutné ověřit existenci všech zdrojových tříd c_i a zkontrolovat, že se extrahovaný atribut p vyskytuje ve všech zdrojových třídách s identickým typem a kardinalitu. Dále je nutné ověřit, že každá ze tříd c_i nemá nastaveného rodiče a také zajistit, aby neexistovala rodičovská třída d do níž bude extrahovaný atribut přesunut. Zápisem $|\{c_1, \dots, c_j\}| = j$ ověříme, že jsou všechny zdrojové třídy navzájem různé. Pokud by různé nebyly, pak počet prvků v množině tříd nebude odpovídat hodnotě proměnné j vyjadřující počet uvedených tříd v operaci.

$$\frac{\Gamma \vdash o : Operation}{\Gamma \vdash \text{unit}; o : Operation}
\quad (\text{T-UNIT})$$

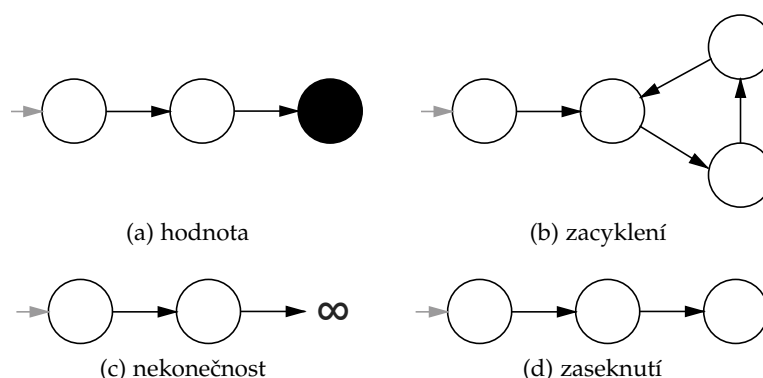
$$\frac{\Gamma \vdash o : Operation}{\Gamma \vdash \text{error}; o : Operation}
\quad (\text{T-ERR})$$

Otypování jednotkové operace **unit** a chybové operace **error** je zachyceno pravidly **T-UNIT** a **T-ERR**. Samotné operace nemají žádné předpoklady, které by bylo nutné ověřit před jejich aplikací. V případě sekvence operací závisí její správné otypování na operaci následující po **unit**, resp. **error**.

Kapitola 8

Vlastnosti jazyka Ops

Vraťme se k abstraktnímu stroji vysvětleném v kapitole 6. Podle uvedeného popisu vyhodnocení syntakticky správného výrazu (programu) abstraktním strojem, dochází na základě dostupných přepisovacích pravidel k přepisování počáteční konfigurace na jiné konfigurace. Vyhodnocení programu končí, pokud je aktuální konfigurace označena jako finální (**hodnota**) nebo neexistuje přepisovací pravidlo, které by současnou nefinální konfiguraci přepsalo na jinou (**zaseknutí**). Avšak samotné vyhodnocení abstraktním strojem nemusí v některých případech vůbec skončit, protože došlo k **zacyklení** nebo k **nekonečnému přepisování** na stále nové konfigurace.



Obrázek 8.1: Možnosti vyhodnocení programu abstraktním strojem

Na obrázku 8.1 jsou zachyceny jednotlivé možnosti, jak může dopadnout vyhodnocení programu abstraktním strojem. Kolečko značí konfiguraci, šipka přepsání jedné konfigurace na druhou, plné kolečko vyjadřuje finální konfiguraci.

Cílem této kapitoly je definování a ověření vybraných vlastností jazyka Ops, které omezují možnosti jakými může dopadnout vyhodnocení abstraktním strojem. Požadujeme, aby vyhodnocení libovolného syntakticky správného programu, napsaného v jazyku Ops, vždy vedlo k úspěšnému zastavení abstraktního stroje a navíc vždy skončilo finální konfigurací (hodnotou). Následující část textu obsahuje popis požadovaných vlastností omezující možnosti vyhodnocení abstraktním strojem:

Terminace: *Libovolný Ops program vždy skončí [28].*

Jinými slovy terminace zaručuje, že vyhodnocení libovolného syntakticky správného programu, napsaném v jazyku Ops, vždy vede k zastavení abstraktního stroje, ať už hodnotou (finální konfigurací) nebo stavem zaseknutí.

Progress: *Dobře otypovaný Ops program je buďto hodnotou nebo existuje přepisovací pravidlo, které jej přepíše na jiný program [21].*

Na první pohled se může zdát, že vlastnost Progress vylučuje stav zaseknutí abstraktního stroje, protože dobře otypovaný program je hodnotou nebo jej přepíšeme. Avšak vlastnost sama o sobě nezaručuje dobré otypování výsledku přepsání. V některých případech může přepsáním původního dobře otypovaného programu vzniknout program, který nelze otypovat a abstraktní stroj se při jeho vyhodnocování zasekne. Z tohoto důvodu potřebujeme další vlastnost o zachování typů:

Preservation: *Přepsáním dobře otypovaného Ops programu na jiný program, je zachován jeho typ [21].¹*

Tato vlastnost v kombinaci s Progress zaručuje tzn. Soundness jazyku. Abstraktní stroj se během vyhodnocování otypovaného programu nikdy nezasekne, vždy existuje vhodné přepisovací pravidlo, které program přepíše na jiný dobře otypovaný, nebo je program již hodnotou. Vlastnost Soundness nezaručuje nezacyklení či konečnost procesu vyhodnocení abstraktním strojem.

8.1 Terminace

Před samotnou formulací vlastnosti terminace, nejprve definujeme pomocnou funkci tzn. energii [28]. Tato funkce libovolné konfiguraci, z množiny všech konfigurací CF viz sekce 6.2, přiřadí přirozené číslo - energii. Energii následně využijeme k dokázání, že se každým přepsáním konfigurace na jinou konfiguraci ostře snižuje její hodnota energie. Jelikož energie počáteční konfigurace udává maximální množství přepisů, které je nutné provést abychom se dostali do finální konfigurace, a navíc tato hodnota je konečná, pak nemohou v programu existovat nekonečné cykly.

Definice 8.1: Energie je funkce $\mathbb{E} : Structure \times Operation \rightarrow \mathbb{N}$ přiřazující libovolné konfiguraci (S, o) její energii podle následujících definicí:

$$\mathbb{E}(S, \text{add } c.p : t) = 1 + \mathbb{E}(S, \text{add } c.p : t[0..1]) \quad (8.1)$$

$$\mathbb{E}(S, \text{add } c.p : t[n]) = 1 + \mathbb{E}(S, \text{add } c.p : t[0..n]) \quad (8.2)$$

$$\mathbb{E}(S, \text{rename } c.p \rightarrow q) = 1 + \mathbb{E}(S, \text{remove } c.p; \text{add } c.q : t[n..m]) \quad (8.3)$$

$$\mathbb{E}(S, \text{move } c.p / q \rightarrow d) = 1 + \mathbb{E}(S, \text{remove } c.p; \text{add } d.p : t[n..m]) \quad (8.4)$$

$$\mathbb{E}(S, \text{pullUp } c.p) = 1 + \mathbb{E}(S, \text{remove } c.p; \text{add } d.p : t[n..m]) \quad (8.5)$$

¹Formulace této vlastnosti se může lišit. V některých případech je pouze vyžadováno, aby přepsáním byl nově vzniklý program dobře otypovaný, ale jeho typ může být odlišný.

$$\mathbb{E}(\mathcal{S}, \text{pushDown } c.p) = 1 + \mathbb{E}(\mathcal{S}, \Delta) \quad (8.6)$$

$$\mathbb{E}(\mathcal{S}, \text{extract } c.p / q \rightarrow d.r) = 1 + \mathbb{E}(\mathcal{S}, \Phi) \quad (8.7)$$

$$\mathbb{E}(\mathcal{S}, \text{extractSub } c.p \rightarrow d) = 1 + \mathbb{E}(\mathcal{S}, \text{add } d; \text{set } d <: c; \text{pushDown } c.p) \quad (8.8)$$

$$\mathbb{E}(\mathcal{S}, \text{extractSuper } \{c_1 \dots c_j\}.p \rightarrow d) = 1 + \mathbb{E}(\mathcal{S}, \Lambda) \quad (8.9)$$

$$\mathbb{E}(\mathcal{S}, o_1; o_2) = 1 + \mathbb{E}(\mathcal{S}, o_1) + \mathbb{E}(\mathcal{S}, o_2) \quad (8.10)$$

$$\mathbb{E}(\mathcal{S}, v) = 0 \quad \text{kde } v \in \{\text{unit}, \text{error}\} \quad (8.11)$$

$$\mathbb{E}(\mathcal{S}, o) = 2 \quad \text{pro v\u0161e ostatn\u00ed} \quad (8.12)$$

Pro jednotliv\u00e9 substitu\u010dn\u00ed prom\u011bn\u011b plat\u00ed:

$$\Delta = \text{remove } c.p; \text{add } c_1.p : t[n..m]; \dots; \text{add } c_j.p : t[n..m], \text{ kde } c_i \in \text{sub}_{\mathcal{S}}(c)$$

$$\Phi = \text{add } d; \text{add } d.r : c[1..1]; \text{add } c.q : d[1..1]; \text{move } c.p / q \rightarrow d$$

$$\Lambda = \text{add } d; \text{set } c_1 <: d; \dots; \text{set } c_i <: d; \text{remove } c_2.p; \dots; \text{remove } c_j.p; \text{pullUp } c_1.p$$

Poznamenejme, \u017e *voln\u011b vyskytuj\u00edc\u00ed* se prom\u011bn\u011b (n, m, a jin\u00e9), ve v\u00fd\u0161e uveden\u00fdch definic\u00edch, mohou b\u00fdt libovoln\u011b zvolen\u00e9, av\u0161ak v r\u00e1mci mno\u017ein\u00fd na n\u00ed\u017e jsou definov\u00e1ny. D\u00e1le si uv\u011bdomme, \u017e nap\u00edr\u00edklad energie $\mathbb{E}(\mathcal{S}, \text{add } c.p : t)$ je stejn\u00e1 jako $\mathbb{E}(\mathcal{S}, \text{add } d.q : s)$ bez ohledu na zvolenou hodnotu prom\u011bn\u011bch.

Tvrz\u00e9n\u00ed 8.2 (Terminace): Jestli\u017ee $(\mathcal{S}, o) \Rightarrow (\mathcal{S}', o')$, pak $\mathbb{E}(\mathcal{S}, o) > \mathbb{E}(\mathcal{S}', o')$.

D\u00falaz. Pro ov\u011bren\u00ed vlastnosti terminace je nutn\u011b matematickou induk\u00c1 uk\u00e1zat, \u017e p\u0159eps\u00e1n\u00edm libovoln\u011b konfigurace (\mathcal{S}, o) na jinou doch\u00e1z\u00ed ke sn\u00ed\u017een\u00ed její energie definovan\u00e9 funkc\u00ed \mathbb{E} . Abychom dok\u00e1zali platnost Terminace pro jazyk Ops, sta\u010d\u00ed ov\u011b\u00edt dokazovan\u00e9 tvrz\u00e9n\u00ed pro ka\u017ed\u00e9 z p\u0159episovac\u00edch pravidel uveden\u00fdch v sekci 6.4. N\u00e1sleduj\u00edc\u00ed \u010d\u00e1st textu obsahuje rozbor jednotliv\u00fdch mo\u017enost\u00ed:

- **E-SEQUNIT:** P\u0159eps\u00e1n\u00edm jednotkov\u00e9 operace `unit` v sekvenci operac\u00ed dojde ke sn\u00ed\u017een\u00ed energie o jednu jednotku:

$$\begin{aligned} \mathbb{E}(\mathcal{S}, \text{unit}; o_2) &= 1 + \mathbb{E}(\mathcal{S}, \text{unit}) + \mathbb{E}(\mathcal{S}, o_2) && \text{dle (8.10)} \\ &= 1 + \mathbb{E}(\mathcal{S}, o_2) && \text{dle (8.11)} \end{aligned}$$

- **E-SEQERR:** Minim\u00e1ln\u011b o jednu jednotku se sn\u00ed\u017e\u00ed energie p\u0159eps\u00e1n\u00edm chybov\u00e9 operace `error` v sekvenci operac\u00ed:

$$\begin{aligned} \mathbb{E}(\mathcal{S}, \text{error}; o_2) &= 1 + \mathbb{E}(\mathcal{S}, \text{error}) + \mathbb{E}(\mathcal{S}, o_2) && \text{dle (8.10)} \\ &> \mathbb{E}(\mathcal{S}, \text{error}) \end{aligned}$$

- **E-ADDCLS:** P\u0159id\u00e1n\u00ed nov\u00e9 t\u0159\u00eddy spot\u0159ebeje dv\u011b jednotky energie:

$$\begin{aligned} \mathbb{E}(\mathcal{S}, \text{add } c) &= 2 + 0 && \text{dle (8.12)} \\ &= 2 + \mathbb{E}(\mathcal{S}[c \mapsto (\text{Top}, \emptyset)], \text{unit}) && \text{dle (8.11)} \end{aligned}$$

- **E-REMCls**: Odstranění třídy spotřebuje dvě jednotky energie analogicky jako právě ověřené prepisovací pravidlo **E-ADDCls**.
- **E-ADDPROP1** a **E-ADDPROP2**: Jedna jednotka energie je spotřebována přidáním nového atributu mající implicitně nastaveny kardinality.

$$\begin{aligned} \mathbb{E}(\mathcal{S}, \text{add } c.p : t) \\ = 1 + \mathbb{E}(\mathcal{S}, \text{add } c.p : t[0..1]) \end{aligned} \quad \text{dle (8.1)}$$

$$\begin{aligned} \mathbb{E}(\mathcal{S}, \text{add } c.p : t[n]) \\ = 1 + \mathbb{E}(\mathcal{S}, \text{add } c.p : t[0..n]) \end{aligned} \quad \text{dle (8.2)}$$

- **E-ADDPROP3**: Jelikož má toto prepisovací pravidlo předpoklad, který nemusí být splněn, vyšetříme obě varianty. První varianta je zachycena pravidlem **E-ADDPROP3**:

$$\begin{aligned} \mathbb{E}(\mathcal{S}, \text{add } c.p : t[n..m]) \\ = 2 + 0 \end{aligned} \quad \text{dle (8.12)}$$

$$= 2 + \mathbb{E}(\mathcal{S}[c \mapsto (d, ps \cup \{p : t[n..m]\})], \text{unit}) \quad \text{dle (8.11)}$$

Druhá možnost je zachycena prepisovacím pravidlem jehož existenci implicitně předpokládáme. Implicitní pravidlo říká, není-li splněn předpoklad, pak konfiguraci přepiš na $(\mathcal{S}, \text{error})$:

$$\begin{aligned} \mathbb{E}(\mathcal{S}, \text{add } c.p : t[n..m]) \\ = 2 + 0 \end{aligned} \quad \text{dle (8.12)}$$

$$= 2 + \mathbb{E}(\mathcal{S}, \text{error}) \quad \text{dle (8.11)}$$

V obou případech, přepsáním původní konfigurace na novou, dojde ke snížení energie o dvě jednotky.

- **E-SEQ**: Aplikujeme indukční předpoklad na předpoklady tohoto prepisovacího pravidla. Podle indukčního předpokladu platí pro první operaci o_1 , že jejím přepsáním dojde ke snížení energie, tzn. $\mathbb{E}(\mathcal{S}, o_1) > \mathbb{E}(\mathcal{S}', o'_1)$:

$$\begin{aligned} \mathbb{E}(\mathcal{S}, o_1 ; o_2) \\ = 1 + \mathbb{E}(\mathcal{S}, o_1) + \mathbb{E}(\mathcal{S}, o_2) \end{aligned} \quad \text{dle (8.10)}$$

$$> 1 + \mathbb{E}(\mathcal{S}', o'_1) + \mathbb{E}(\mathcal{S}, o_2) \quad \text{dle induk. předpokladu}$$

$$= \mathbb{E}(\mathcal{S}', o'_1 ; o_2) \quad \text{dle (8.10)}$$

Přepsáním první operace v sekvenci dojde ke snížení energie minimálně o jednu jednotku.

- **E-REMPROP**, **E-NAMPROP**, **E-MOVPROP**, **E-PULLPROP**, **E-PUSHPROP**, **E-SETPAR**, **E-REMPAR**, **E-EXTCls**, **E-EXTSUB**, **E-EXTSUPER**: Důkaz těchto případů je analogický s uvedeným postupem u výše ověřených pravidel.

■

Právě jsme dokázali, že jazyk Ops má vlastnost terminace, nebo-li vyhodnocení libovolného syntakticky správného výrazu, napsaném jazyce Ops, abstraktním strojem, vždy vede k jeho úspěšnému zastavení. Abstraktní stroj buď výraz vyhodnotí na finální konfiguraci (hodnotu) nebo neexistuje vhodné přepisovací pravidlo, které by aktuální konfiguraci přepsalo na jinou (zaseknutý stav). Vlastností terminace máme zaručeno, že nikdy nedojde k zacyklení abstraktního stroje či k nekonečnému vyhodnocování.

8.2 Soundness = Progress + Preservation

Tvrzení 8.3 (Progress): Jestliže operace o je *dobře otypovaná* (psáno $\emptyset \vdash o : T$), pak:

- (i) o je hodnota (unit, error), nebo
- (ii) pro nějakou strukturu S existují o' a S' takové, že $(S, o) \Rightarrow (S', o')$.

Důkaz. Strukturální indukcí podle derivace $o : T$ provedeme důkaz vlastnosti Progress. V každém kroku strukturální indukce budeme předpokládat, že dokazovaná vlastnost platí pro všechny dílčí derivace. Nejdříve ověříme platnost tvrzení pro axiomy, tzn. typová pravidla bez předpokladů, pak pro ostatní pravidla. Jelikož jsme typová pravidla pro každou operaci definovali jako sekvenci $\Gamma \vdash o_1; o_2 : Operation$, kde druhá operace může být nahrazena pomocným symbolem ϵ , provedeme rozbor jednotlivých pravidel ve dvou fázích. První fází ověříme typová pravidla pro samostatné operace:

- **T-UNIT:** $o = \text{unit}$

Splněno triviálně, o je hodnota unit.

- **T-ERR:** $o = \text{error}$

Analogicky jako případ **T-UNIT**.

- **T-ADDCLS:** $o = \text{add } c$

Předpoklady typového pravidla **T-ADDCLS** vyžadují, aby prázdné prostředí neobsahovalo přidávanou třídu c , což je splněno. Dále chceme ukázat, že o je buď hodnota nebo existuje přepisovací pravidlo, které by o přepsalo. Podle axiomu **E-ADDCLS** umíme přepsat o nad nějakým S na konfiguraci $(S[c \mapsto (Top, \emptyset)], \text{unit})$.

- **T-REMCLS:** $o = \text{remove } c$

Z předpokladů pravidla **T-REMCLS** vyplývá, že prostředí Γ musí obsahovat odstraněnou třídu c . Jelikož důkaz tvrzení provádíme nad prázdným prostředím, nemohou být splněny předpoklady tohoto pravidla. Protože neplatí předpoklad dokazované implikace, tvrzení automaticky platí pro **T-REMCLS**.

- **T-ADDPROP1:** $o = \text{add } c.p : t$

Na základě axiomu **E-ADDPROP1** umíme přepsat operaci o nad nějakým S na konfiguraci $(S, \text{add } c.p : t[0..1])$.

- **T-ADDPROP2:** $o = \text{add } c.p : t[n]$

Analogicky jako případ **T-ADDPROP1**.

- **T-ADDPROP3**, **T-REMPROP**, **T-NAMPROP**, **T-MOVPROP**, **T-PULLPROP**, **T-PUSHPROP**, **T-SETPAR**, **T-REMPAR**, **T-EXTCLS**, **T-EXTSUB**, **T-EXTSUPER**: Dokázání platnosti tvrzení pro tyto případy provedeme stejným způsobem jako v případě **T-REMCls**. Každé ze zmíněných typových pravidel nějakým způsobem ověřuje existenci atributů či tříd nad prázdným prostředím Γ . Obdobně jako v případě **T-REMCls**, platí dokazovaná implikace automaticky, protože není splněn její předpoklad.

V druhé fázi ověříme platnost tvrzení pro zřetězení operací:

- **T-UNIT**: $o = \text{unit}; o_1$
 $o_1 : \text{Operation}$

Na základě prepisovacího pravidla **E-SEQUNIT** umíme přepsat sekvenci operací o nad nějakou strukturou \mathcal{S} na konfiguraci (\mathcal{S}, o_1) .

- **T-ERR**: $o = \text{error}; o_1$
 $o_1 : \text{Operation}$

Obdobně jako v případě **T-UNIT**, umíme využitím prepisovacího pravidla **E-SEQERR** přepsat sekvenci operací o nad nějakou strukturou \mathcal{S} na konfiguraci $(\mathcal{S}, \text{error})$.

- **T-ADDCLS**: $o = \text{add } c; o_1$
 $o_1 : \text{Operation}$

Abychom ověřili platnost tvrzení pro typové pravidlo **T-ADDCLS**, musí sekvence o být hodnota nebo musí existovat prepisovací pravidlo, které by jí přepsalo. Jelikož o není hodnota, pak musíme najít vhodné prepisovací pravidlo. Podle pravidla **E-SEQ** umíme přepsat sekvence operací, jestliže umíme přepsat první operaci z dané sekvence. Zkombinováním pravidla **E-SEQ** a axiomu **E-ADDCLS** přepíšeme sekvenci $(\mathcal{S}, \text{add } c; o_1)$ na novou konfiguraci $(\mathcal{S}[c \mapsto (\text{Top}, \emptyset)], \text{unit}; o_1)$.

- **T-REMCls**: $o = \text{remove } c; o_1$

Předpoklad typového pravidla **T-REMCls** obsahuje ověření existence odstraňované třídy c v prostředí Γ , které je ovšem prázdné. Dokazované tvrzení je automaticky splněno pro tento případ.

- **T-ADDPROP1**: $o = \text{add } c.p : t; o_1$
 $\text{add } c.p : t[0..1]; o_1 : \text{Operation}$

Stejným postupem jako v případě **T-ADDCLS** provedeme zkombinování prepisovacího pravidla **E-SEQ** a axiomu **E-ADDPROP1**, čímž se sekvence operací o přepíše nad nějakou strukturou \mathcal{S} na $(\mathcal{S}, \text{add } c.p : t[0..1]; o_1)$.

- **T-ADDPROP2**: $o = \text{add } c.p : t[n]; o_1$
 $\text{add } c.p : t[n]; o_1 : \text{Operation}$

Analogicky jako v případě **T-ADDPROP1**.

- **T-ADDPROP3**, **T-REMPROP**, **T-NAMPROP**, **T-MOVPROP**, **T-PULLPROP**, **T-PUSHPROP**, **T-SETPAR**, **T-REMPAR**, **T-EXTCLS**, **T-EXTSUB**, **T-EXTSUPER**: Dokázání platnosti tvrzení těchto případů provedeme analogicky jako u **T-REMCls**. Předpoklady každého ze zmíněných typových pravidel nemohou být splněny nad prázdným prostředím Γ . Z tohoto důvodu platí dokazovaná implikace automaticky, protože není splněn její předpoklad.

■

Tvrzení 8.4 (Preservation): Jestliže $\emptyset \vdash o : T$, $(\mathcal{S}, o) \Rightarrow (\mathcal{S}', o')$, pak $\emptyset \vdash o' : T$.

Důkaz. Pomocí strukturální indukce podle derivace $o : T$ dokážeme vlastnost o zachování typů. V každém kroku indukce se budeme snažit najít vhodné přepisovací pravidlo, které by přepsalo o na o' nad nějakou strukturou \mathcal{S} . Následně ověříme, zdali operaci o' byl zachován stejný typ T . Jelikož jsme typová pravidla pro každou operaci definovali jako sekvenci, v níž může být druhá operace nahrazena speciálním symbolem, provedeme rozbor jednotlivých typových pravidel ve dvou fázích. V první fázi provedeme rozbor typových pravidel pro samostatné operace:

- **T-UNIT:** $o = \text{unit}$ $T = \text{Operation}$

Protože operace $o = \text{unit}$ je hodnota, pak neexistuje vhodné přepisovací pravidlo, které by přepsalo $(\mathcal{S}, o) \Rightarrow (\mathcal{S}', o')$. Předpoklad dokazovaného tvrzení není splněn, implikace platí automaticky.

- **T-ERR:** $o = \text{error}$ $T = \text{Operation}$

Analogicky jako případ **T-UNIT**.

- **T-ADDCLS:** $o = \text{add } c$ $T = \text{Operation}$

Musíme nalézt takové přepisovací pravidlo mající ve svém závěru na levé straně operaci o . Tomuto požadavku vyhovuje axiom **E-ADDCLS**, podle něhož se o nad nějakou strukturou \mathcal{S} přepíše na operaci $o' = \text{unit}$. Na základě typového pravidla **T-UNIT**, pro otypování samotné operace unit , víme, že přepsáním operace o na operaci o' nedošlo ke změně typu (Operation).

- **T-REMCLS:** $o = \text{remove } c$ $T = \text{Operation}$

Podle předpokladů typového pravidla musí prostředí Γ obsahovat odstraňovanou třídu c . Protože provádíme důkaz nad prázdným prostředím, pak není možné splnit předpoklady pravidla **T-REMCLS** a dokazovaná implikace platí.

- **T-ADDPROP1:** $o = \text{add } c.p : t$ $T = \text{Operation}$
 $\text{add } c.p : t[0..1] : \text{Operation}$

Prohledáním přepisovacích pravidel, můžeme operaci o přepsat axiomem **E-ADDPROP1** na $o' = \text{add } c.p : t[0..1]$. Podle předpokladů právě ověřovaného typového pravidla **T-ADDPROP1** je výraz $\text{add } c.p : t[0..1]$ typu Operation . Přepsáním nedojde ke změně typů.

- **T-ADDPROP2:** $o = \text{add } c.p : t[n]$ $T = \text{Operation}$
 $\text{add } c.p : t[0..n] : \text{Operation}$

Analogicky jako případ **T-ADDPROP1**.

- **T-ADDPROP3, T-REMPROP, T-NAMPROP, T-MOVPROP, T-PULLPROP, T-PUSHPROP, T-SETPAR, T-REMPAR, T-EXTCLS, T-EXTSUB, T-EXTSUPER:** Dokazovaná implikace pro tyto případy platí automaticky, protože není splněn jeden z jejích předpokladů - $\emptyset \vdash o \not\vdash T$.

V druhé fázi ověříme platnost tvrzení pro zřetězení operací:

- **T-UNIT:** $o = \text{unit}; o_1 \quad T = \text{Operation}$
 $o_1 : \text{Operation}$

Na základě přepisovacího pravidla **E-SEQUNIT** umíme přepsat sekvenci operací o na $o' = o_1$. Jelikož z předpokladu dokazovaného typového pravidla **T-UNIT** víme, že $o_1 : \text{Operation}$, pak má operace o' stejný typ jako původní sekvence o .

- **T-ERR:** $o = \text{error}; o_1 \quad T = \text{Operation}$
 $o_1 : \text{Operation}$

Obdobně jako v případě **T-UNIT** nalezneme vhodné přepisovací pravidlo, jež by přepsalo sekvenci o . Tomuto požadavku vyhovuje axiom **E-SEQERR**, který sekvenci přepíše na $o' = \text{error}$. Podle typového pravidla **T-ERR**, pro otypování samostatné operace error , víme, že přepsáním sekvence o nedošlo ke změně jejího typu, protože o' je typu Operation .

- **T-ADDCLS:** $o = \text{add } c; o_1 \quad T = \text{Operation}$
 $o_1 : \text{Operation}$

Podle přepisovacího pravidla **E-SEQ** lze přepsat sekvenci operací o na o' , umíme-li přepsat první operaci vyskytující se v této sekvenci. Na základě axiomu **E-ADDCLS** lze přepsat první operaci $\text{add } c$ nad nějakou strukturou \mathcal{S} na konfiguraci $(\mathcal{S}[c \mapsto (\text{Top}, \emptyset)], \text{unit})$. Tímto máme splněn předpoklad pravidla **E-SEQ**, kterým přepíšeme o na $o' = \text{unit}; o_1$. Dále zbývá ověřit, zdali přepsání sekvence zachovává typy. Využitím skutečnosti, že $o_1 : \text{Operation}$ a typového pravidla **T-UNIT**, pro otypování sekvence, snadno ukážeme, že přepsáním sekvence o na o' nebyl změněn její typ.

- **T-REMCLS:** $o = \text{remove } c; o_1$

Předpoklad typového pravidla **T-REMCLS** obsahuje ověření existence odstraňované třídy c v prostředí Γ , které je ovšem prázdné. Dokazované tvrzení je automaticky splněno pro tento případ.

- **T-ADDPROP1, T-ADDPROP2:** $o = \text{add } c.p : t; o_1 \quad T = \text{Operation}$
 $\text{add } c.p : t[0..1]; o_1 : \text{Operation}$

Analogicky jako v případě **T-ADDCLS**. Nejprve pomocí axiomů **E-ADDPROP1**, resp. **E-ADDPROP2**, přepíšeme první operaci sekvence a pak využitím přepisovacího pravidla **E-SEQ** přepíšeme o na o' . Jelikož z předpokladu právě ověřovaného typového pravidla **T-ADDPROP1**, resp. **T-ADDPROP2**, víme, že $\text{add } c.p : t[0..1]; o_1 : \text{Operation}$, pak má operace o' stejný typ jako původní sekvence o .

- **T-ADDPROP3, T-REMPROP, T-NAMPROP, T-MOVPROP, T-PULLPROP, T-PUSHPROP, T-SETPAR, T-REMPAR, T-EXTCLS, T-EXTSUB, T-EXTSUPER:** Ve všech zmíněných případech snadno ukážeme, že dokazované tvrzení platí automaticky kvůli nesplnění předpokladů. ■

Dokázali jsme, že správně otypovaný program, napsaný v jazyce Ops, zachovává typy. Společně s výše dokázanou vlastností Progress máme zaručeno, že se abstraktní stroj během vyhodnocování otypovaného programu nikdy nezasekne, tzn. vždy existuje vhodné přepisovací pravidlo, které aktuální konfiguraci přepíše na jinou, nebo je aktuální konfigurace označena jako finální (hodnota). Vlastnost Soundness nezaručuje nezacyklení či konečnost procesu vyhodnocení abstraktním strojem.

Kapitola 9

Implementace

Pro implementaci jazyku Ops, jako plug-inu do vývojového prostředí Eclipse IDE, jsme na základě [16] zvolili nástroj Xtext [25]. Tento nástroj umožňuje z gramatických pravidel automaticky vytvořit prostředky pro provedení lexikální analýzy (lexer), syntaktické analýzy (parser), tvorbu objektového abstraktního syntaktického stromu (AST) a uživatelské prostředí pro snazší manipulaci s programy napsané ve vyvíjeném jazyku.

9.1 Xtext gramatika

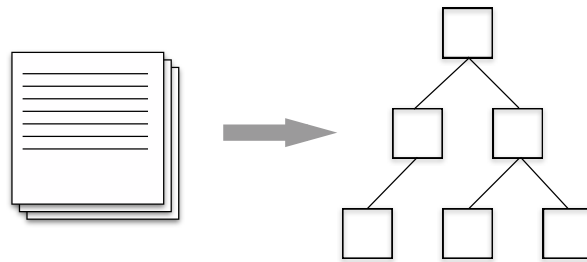
Xtext pro definici gramatických pravidel používá zápis podobný rozšířené Backus-Naurově formě použité v kapitole 5. Narozdíl od této formy rozlišuje dva typy gramatických pravidel: terminální a syntaktické.

Terminální pravidla popisují sekvenci znaků, která je během lexikální analýzy převedena na lexémy (tokeny), jež jsou použity v průběhu syntaktické analýzy pro ověření správného sledu jednotlivých po sobě jdoucích sekvencí. Příkladem terminálního pravidla je INT definující přirozená čísla. Terminální pravidla jsou v Xtextu označena klíčovým slovem `terminal` po němž následuje pojmenování velkými písmeny. Pro potřeby jazyku Ops si vystačíme z předdefinovanými terminálními pravidly, avšak kromě ID, které povoluje nepřípustné znaky pro pojmenování názvů atributů a tříd. Z tohoto důvodu předdefinujeme terminální pravidlo ID následovně:

```
1 terminal ID:  
2     ('a'..'z'|'A'..'Z') ('a'..'z'|'A'..'Z'|'0'..'9')*  
3 ;
```

Syntaktické pravidla odpovídají gramatickým pravidlům a slouží během procesu syntaktické analýzy k ověření, zdali pořadí jednotlivých terminálů (lexémů) a neterminálů koresponduje s definicemi těchto pravidel. Na základě definice syntaktických pravidel je sestaven abstraktní syntaktický strom ze vstupního výrazu, viz obrázek 9.1, který najde uplatnění v dalších fázích práce se vstupním výrazem (např. validace).

Jelikož Xtext ke každému syntaktickému pravidlu automaticky vygeneruje stejně pojmenovanou třídu včetně její atributů, čímž lze vytvořit objektový abstraktní strom, je možné využitím klíčového slova `returns` specifikovat jaká třída má být použita. Podle



Obrázek 9.1: Převod vstupního výrazu na abstraktní syntaktický strom (MigDb model)

této skutečnosti, použijeme třídy z již existujícího aplikačního metamodelu frameworku MigDb, namísto zbytečného vytváření nového modelu odpovídajícího struktuře syntaxe a jeho následné konverzi. Neboť struktura aplikačního metamodelu je pevně definovaná a nelze jí měnit bez větších dopadů na ostatní části frameworku MigDb, nebude naimplementovaná syntaxe jazyku Ops zcela korespondovat se syntaxí z kapitoly 5.

Gramatická pravidla jazyku Ops, s využitím nástroje Xtext, mají následující podobu:

```

1 ModelRoot returns app::ModelRoot:
2   Operations
3 ;
4
5 Operations returns app::Operations:
6   operations+=ModelOperation
7   (';'? operations+=ModelOperation)*
8 ;
9
10 ModelOperation returns ops::ModelOperation:
11   AddClass
12   | RemoveEntity
13   | AddProperty
14   | RenameProperty
15   | RemoveProperty
16   | MoveProperty
17   | PullUpProperty
18   | PushDownProperty
19   | AddParent
20   | RemoveParent
21   | ExtractClass
22   | ExtractSubClass
23   | ExtractSuperClass
24 ;
25
26 RemoveEntity returns ops::RemoveEntity:
27   'remove' name=ID
28 ;

```

Ukázka kódu 9.1: Ukázka gramatických pravidel jazyku Ops v Xtext

Všimněme si výskytu operátorů přiřazení (=, +=) v syntaktických pravidlech. Nástroj Xtext na základě pojmenování předcházející jednotlivým operátorům, vygeneruje odpovídající atributy tříd reprezentující dané syntaktické pravidlo. Navíc jsou tyto atributy naplněny požadovaným typem hodnot během tvorby abstraktního stromu.

Po úspěšném vytvoření gramatiky v nástroji Xtext, zbývá provést vygenerování samotných jazykových komponent. Spuštěním pracovního toku `GenerateOps.mwe2` dojde mimo jiné k vytvoření:

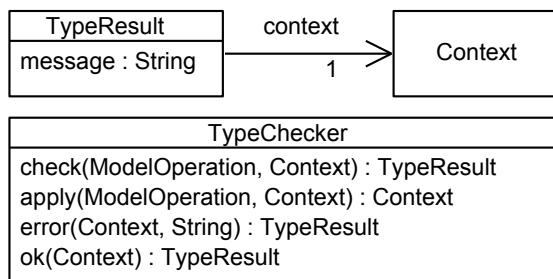
- editoru s funkcemi jako zvýraznění klíčových slov, doplňování kódu,
- formátovací komponenty pro automatizovanou organizaci textu,
- komponenty pro validování nad abstraktním stromem.

V následujících podkapitolách navážeme na jednotlivé vygenerované komponenty a popíšeme jejich rozšíření pro potřeby jazyku Ops.

9.2 Typový systém

Vznikem validační komponenty se nám naskytne vhodné prostředí, kde lze provádět typovou kontrolu vstupního programu. Validační komponenta přistupuje k jednotlivým uzlům abstraktního stromu a umožňuje nad nimi provádět různé kontroly požadovaných vlastností. Navíc máme zaručenu syntaktickou správnost programu, protože vzniku abstraktního stromu předchází lexikální a syntaktická analýza (parsování).

Implementace typového systému je provedena pomocí jazyku Xtend [24], který je nadstavbou jazyku Java a snaží se ho obohatit a odstínit od zbytečného psaní kódu. Hlavní výhodou Xtend oproti Javě je nativní podpora tzv. *multiple method dispatch*¹, umožňující vznik více metod se stejným pojmenováním, ale různými typy parametrů. Výběr vhodné metody je proveden na základě zadaných parametrů.



Obrázek 9.2: UML diagram tříd implementace typového systému

Struktura typového systému je zachycena UML diagramem tříd viz 9.2, kde uvedené třídy mají tento význam:

Context je pomocná struktura pro uložení stavu typového kontextu Γ během procesu typování. V instanci *Context* nalezneme veškeré informace o existenci tříd včetně jejich atributů, ale také informaci o vzniklých hierarchiích tříd.

TypeResult reprezentuje výsledek typové kontroly a kromě výsledného stavu kontextu může obsahovat chybové hlášení, uložené v atributu *message*, jež je následně použito pro informování uživatele o nalezených nedostatcích v Ops programu.

¹http://en.wikipedia.org/wiki/Multiple_dispatch

TypeChecker implementuje typová pravidla z kapitoly 7. Obsahuje multiple dispatch metodu *check()*, které je předán uzel abstraktního stromu (instance potomka třídy *ModelOperation*) a aktuální stav kontextu nad nímž dojde k ověření předpokladů typového pravidla odpovídající operace. Jsou-li tyto předpoklady splněny, pak voláním metody *apply()*, s příslušnými parametry, je aktuální stav kontextu pozměněn s respektováním úprav typového kontextu uvedených u každého typového pravidla v sekci 7.2.

Připomeňme si definici typového pravidla **T-REMCls** operace **REMOVE CLASS** sloužící k odstranění třídy z modelu:

$$\frac{c \in \Gamma \quad \text{prop}_{\Gamma}(c) = \text{sub}_{\Gamma}(c) = \emptyset \quad d.q : c[n..m] \notin \Gamma \quad \Gamma \setminus \{c\} \vdash o : \text{Operation}}{\Gamma \vdash \text{remove } c; o : \text{Operation}}$$

Podle jeho předpokladů musí odstraňovaná třída existovat v typovém prostředí, její množiny atributů a potomků musejí být prázdné, a navíc nesmí existovat atribut jehož typ odpovídá právě odstraňované třídě. Ověření těchto předpokladů je zahyceno následující implementací metody *check()*:

```

1 def dispatch TResult check(RemoveEntity op, Context g) {
2   if(!g.existsClass(op.name))
3     return error(g, Errors::classNotExists(op.name))
4   if(g.hasProp(op.name))
5     return error(g, Errors::classHasProperties(op.name))
6   if(g.hasSub(op.name))
7     return error(g, Errors::classHasSubclasses(op.name))
8   if(g.existsPropertyType(op.name))
9     return error(g, Errors::classTypeExists(op.name))
10  return ok(op.apply(g))
11 }

```

Ukázka kódu 9.2: Implementace typového pravidla **T-REMCls**

Je-li splněna libovolná z uvedených podmínek v metodě *check()*, pak dojde voláním *error()* k vytvoření instance třídy *TypeResult* obsahující původní stav typového kontextu společně s chybovým hlášením specifikující nesplněný předpoklad. Naopak v případě nesplnění ani jedné z podmínek dojde voláním *ok()* k vrácení nové instance *TypeResult* s pozměněným kontextem upraveným metodou *apply()*:

```

1 def dispatch Context apply(RemoveEntity op, Context g) {
2   return g.removeClass(op.name)
3 }

```

Ukázka kódu 9.3: Implementace úpravy typového kontextu pravidlem **T-REMCls**

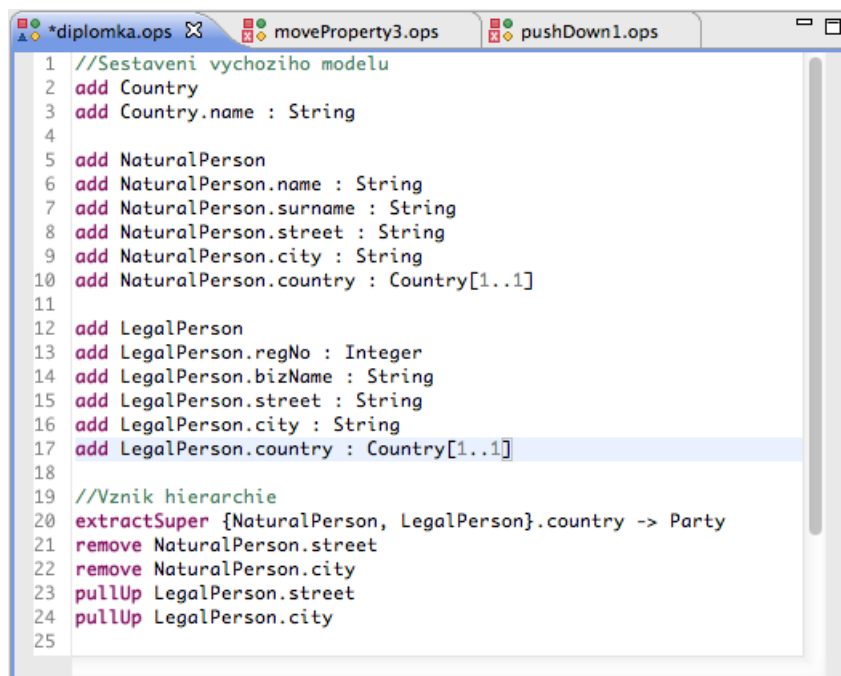
Implementace této metody provede odebrání odstraňované třídy z kontextu, čímž je umožněno provedení otypování operace následující po **REMOVE CLASS** nad aktualizovaným stavem kontextu. Můžeme tak detekovat chyby jako je přidávání atributu neexistující třídě, protože aktualizovaný kontext nebude obsahovat informaci o existenci dané třídy.

9.3 Rozšíření grafického rozhraní

Po úspěšném vytvoření gramatiky lze nástrojem Xtext automaticky vygenerovat uživatelské prostředí pro snadnou tvorbu a editaci programů napsaných ve vyvíjeném jazyce. Vygenerované prostředí v základu disponuje funkcemi pro zvýraznění klíčových slov či doplňování kódu. Navíc může být rozšířeno o další pokročilé funkce, které zpříjemňují uživateli práci během vývoje Ops programů.

9.3.1 Formátování

Jednou z možností rozšíření uživatelského prostředí je formátování Ops programů vně editoru. Základní verze komponenty pro formátování pouze mezi každý terminál či skupinu terminálů vloží mezeru, tím vznikne velký shluk znaků výrazně snižující čitelnost celého programu.



```
1 //Sestaveni vychozihho modelu
2 add Country
3 add Country.name : String
4
5 add NaturalPerson
6 add NaturalPerson.name : String
7 add NaturalPerson.surname : String
8 add NaturalPerson.street : String
9 add NaturalPerson.city : String
10 add NaturalPerson.country : Country[1..1]
11
12 add LegalPerson
13 add LegalPerson.regNo : Integer
14 add LegalPerson.bizName : String
15 add LegalPerson.street : String
16 add LegalPerson.city : String
17 add LegalPerson.country : Country[1..1]
18
19 //Vznik hierarchie
20 extractSuper {NaturalPerson, LegalPerson}.country -> Party
21 remove NaturalPerson.street
22 remove NaturalPerson.city
23 pullUp LegalPerson.street
24 pullUp LegalPerson.city
25
```

Obrázek 9.3: Ukázka editoru Ops programů

Formátovací komponenta byla upravena, čímž je každá operace umístěna na svůj speciální řádek a navíc došlo k odstranění přebytečných mezer. Výsledek zformátování programu je zachycen obrázkem 9.3.

9.3.2 Průvodce novým Ops programem

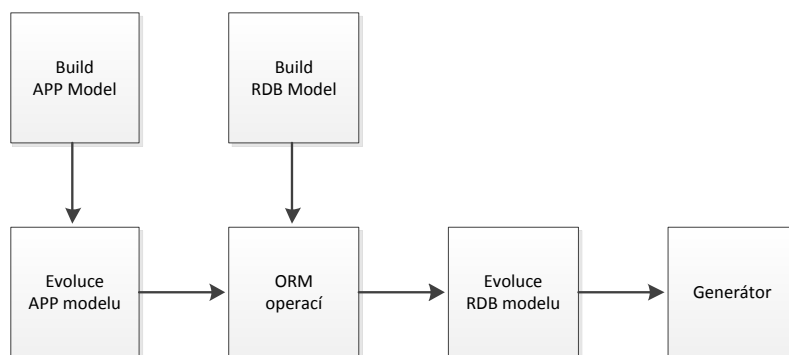
Dalším rozšířením uživatelského rozhraní bylo přidání průvodce pro vytvoření nových Ops programů do nabídky *File* → *New* v prostředí Eclipse IDE. Uživateli je tím pádem poskytnuta možnost snadné tvorby nových prázdných programů.

9.4 Operační sémantika

Implementace operační sémantiky je pokryta pracemi [13] a [14] od Martina Lukeše. Mnou psaná diplomová práce obsahuje pouze její formální specifikaci.

9.5 Integrace jazyku Ops s MigDb

Struktura frameworku MigDb, jak již bylo zmíněno v kapitole 2, je rozdělena do čtyř základních vrstev, které mezi sebou spolupracují za účelem vygenerování SQL skriptu na základě vstupního aplikačního modelu. Spolupráce jednotlivých vrstev je zachycena obrázkem 9.4. Pro integraci jazyku Ops s MigDb je nejpodstatnější první vrstva (Evoluce APP modelu) na níž navážeme.



Obrázek 9.4: Pracovní tok frameworku MigDb, obrázek převzat z [12, str. 13]

První vrstvě je předán sestavený aplikační model obsahující sadu operací a strukturu nad níž má dojít k vykonání operací. V našem případě budeme uvažovat, že je vstupní struktura vždy prázdná. Následně první vrstva provede evoluci vstupního aplikačního modelu, což znamená sekvenční aplikování jednotlivých operací na strukturu. Aplikaci každé z operací předchází ověření jejich předpokladů, viz kapitola 4. Jsou-li všechny operace úspěšně vykonány, dojde k jejich namapování na databázové operace (RDB), které jsou následně zpracovány ostatními vrstvami dokud nedojde k vygenerování SQL skriptu obsahující výsledné úpravy databázových tabulek.

Integraci jazykového plug-inu Ops s frameworkem MigDb rozdělíme do několika fází:

9.5.1 MWE2 pracovní tok

V první fázi integrace sestavíme pracovní tok (workflow) využitím jazyku MWE2 [26]. Tento jazyk umožňuje tvorbu různých pracovních postupů skládáním komponent, které se vzájemně ovlivňují. Spuštěním pracovního toku dojde k sekvenčnímu vyhodnocení jednotlivých komponent jejichž výstup může být použit jako vstup pro ostatní komponenty pomocí tzv. přepravek dat (slotů). V základu MWE2 disponuje předdefinovanými komponentami pro načítání souborů, mazání adresářů a jiné. Doplňující komponenty mohou být uživatelem libovolně naprogramovány.

Pomocí existujících komponent z frameworku MigDb docílíme vytvoření všech vrstev z obrázku 9.4. Zbývající sestavení aplikačního modelu (Build APP Model), obsahující operace a prázdnou strukturu, vytvoříme složením komponent *OpsReader* a *QVTOExecutor2*:

```

1  var OPS_PATH //cesta k Ops programu predana zvenci
2
3  Workflow {
4  //nastaveni pracovniho toku
5
6      component = OpsReader {
7          uri = "${OPS_PATH}"
8          outputSlot = "opsApp"
9      }
10     component = QVTOExecutor2 {
11         transformationFile = "${QVTO_DIR}/default_primitives.qvto"
12         outputSlot = "strApp"
13     }
14
15     //ostatni komponenty
16 }

```

Ukázka kódu 9.4: Sestavení aplikačního modelu z Ops programu

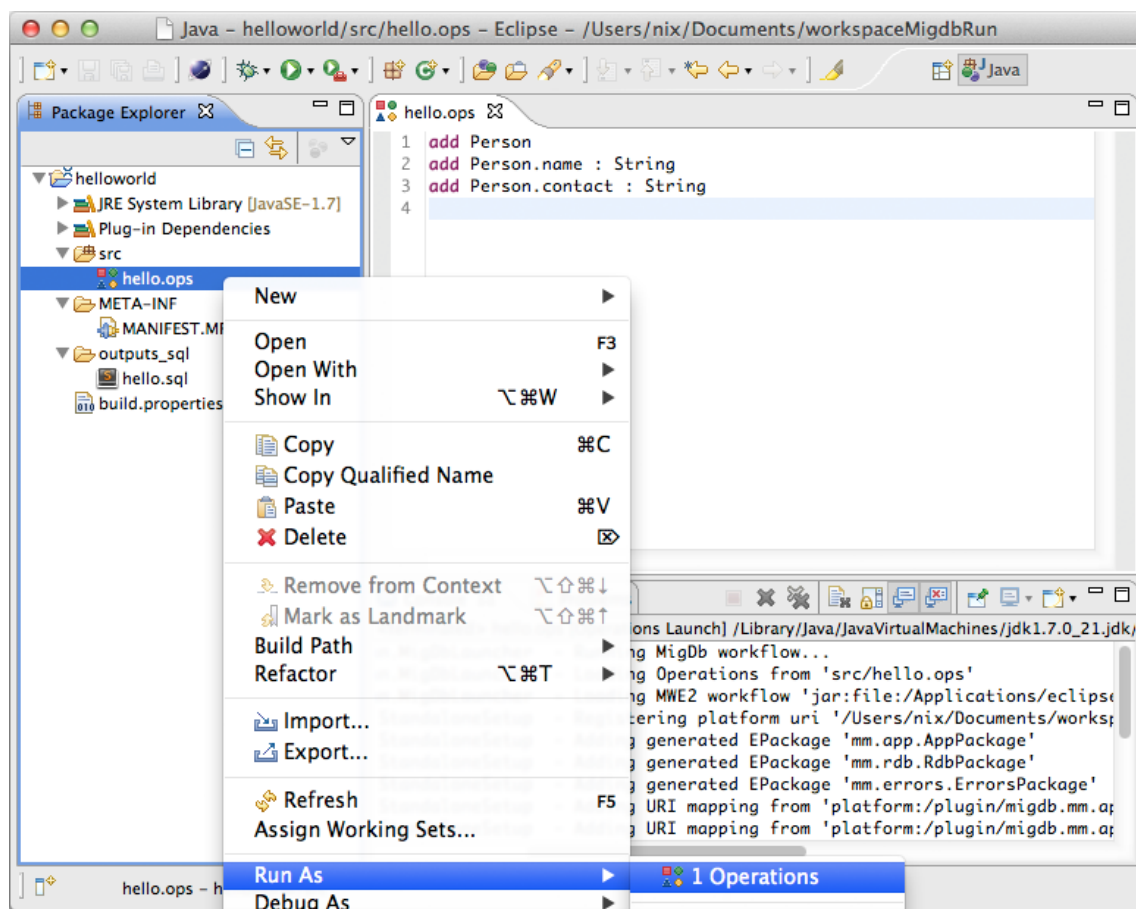
Komponenta *OpsReader* načte Ops program a vytvoří z něj objektový abstraktní strom. Tento abstraktní strom je následně uložen do přepravky pojmenované *opsApp* pro použití dalšími komponentami. Komponenta *QVTOExecutor2* provede spuštění QVTO transformace *default_primitives.qvto*, která sestaví strukturu obsahující pouze primitivní typy, a její výsledek uloží do přepravky *strApp*. Obě vzniklé přepravky jsou předány ostatním komponentám frameworku MigDb, které provedou evoluci aplikačního modelu, objektově-relační mapování operací, evoluci databázového modelu a vygenerování SQL skriptu.

9.5.2 Spuštění Ops programu

Druhá fáze integrace jazykového plug-inu Ops do MigDb spočívá v rozšíření vývojového prostředí Eclipse IDE o další typ spouštěcí konfigurace (Run Configuration), která bude umožňovat spouštění Ops programů v rámci Eclipse IDE. Typickým příkladem využívající spouštěcí konfigurace jsou Java projekty/soubory. Pokud chceme tyto projekty spustit, vytvoří se v Eclipse IDE příslušná konfigurace obsahující údaje o názvu projektu, názvu třídy obsahující *main()* metodu apod. Na základě těchto údajů spustí Eclipse IDE příslušný projekt nad Java Virtual Machine² a veškerý jeho výstup je tisknut do okna konzole vývojového prostředí.

Rozšířením tříd *JavaLaunchDelegate*, *ILaunchShortcut*, *ILaunchConfigurationTabGroup* a vytvořením třídy *MigDbLauncher* vznikne dostatečná infrastruktura pro spouštění Ops programů v rámci Eclipse IDE. Spuštěním Ops programu je předána jeho cesta třídě *MigDbLauncher*, která zajistí vykonávání pracovního toku z sekce 9.5.1 včetně doplnění nutných parametrů (cesta k Ops programu, název výstupního SQL skriptu). Ukázka spuštění Ops programu v Eclipse IDE je zachycena obrázkem 9.5.

²http://en.wikipedia.org/wiki/Java_virtual_machine



Obrázek 9.5: Spuštění Ops programu v Eclipse IDE

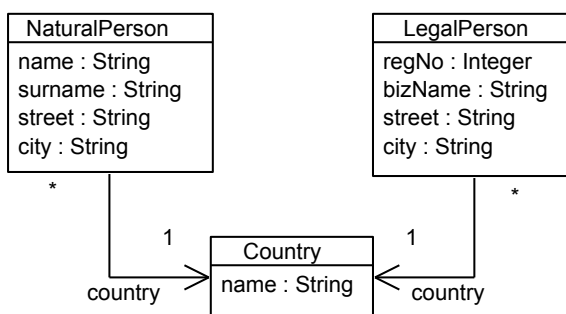
Kapitola 10

Testování

V této kapitole ukážeme použití jazyku Ops na jednoduchém aplikačním modelu převzatého z [12, 22]. Dále provedeme ověření funkčnosti a správné implementace typového systému, na vybraných testovacích příkladech, pomocí jednotkového testování (unit testing) využitím nástroje xtext-utils [27] umožňující automatické testování Xtext projektů.

10.1 Testovací příklad

Aplikační model testovacího příkladu zachycuje obyčejný registr fyzických a právnických osob. Součástí modelu jsou třídy: *LegalPerson* zastupující fyzické osoby, *NaturalPerson* reprezentující fyzické osoby a v poslední řadě třída *Country*, jenž představuje země ze kterých mohou osoby pocházet.



Obrázek 10.1: Výchozí stav testovacího modelu

Negativem tohoto modelu je duplikace kódu, kdy se stejné atributy vyskytují v obou třídách *LegalPerson* a *NaturalPerson*, ale také duplikace samotných adres, které v případě, že dvě osoby sídlí na stejné adrese, jsou zbytečně uloženy na dvě různá místa. Zmíněné neduhy se budeme snažit eliminovat v následujících sekcích:

10.1.1 Sestavení výchozího modelu

Před samotným rozšířením modelu, musíme nejdříve provést jeho sestavení. Využitím kombinace operací `ADD CLASS` a `ADD PROPERTY` vznikne model odpovídající obrázku 10.1:

```

1 add Country
2 add Country.name : String
3
4 add NaturalPerson
5 add NaturalPerson.name : String
6 add NaturalPerson.surname : String
7 add NaturalPerson.street : String
8 add NaturalPerson.city : String
9 add NaturalPerson.country : Country [1..1]
10
11 add LegalPerson
12 add LegalPerson.regNo : Integer
13 add LegalPerson.bizName : String
14 add LegalPerson.street : String
15 add LegalPerson.city : String
16 add LegalPerson.country : Country [1..1]

```

Ukázka kódu 10.1: Sestavení testovacího modelu

10.1.2 Vznik hierarchie

Jak již bylo zmíněno, nevýhodou původního modelu, z obrázku 10.1, je duplikace kódu, kdy atributy *street*, *city*, *country* jsou v obou typech osob. Tohoto nešvaru se zbavíme vytvořením speciální rodičovské třídy, která bude obsahovat dané atributy.

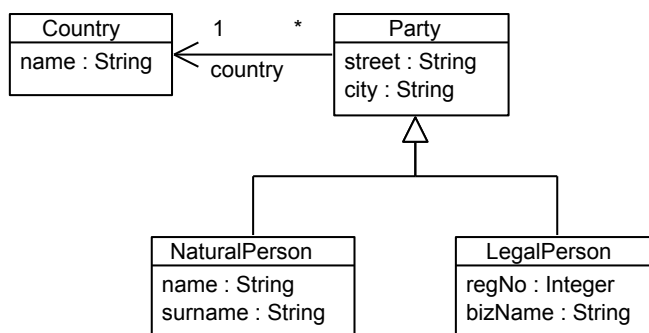
```

1 extractSuper {NaturalPerson, LegalPerson}.country -> Party
2 remove NaturalPerson.street
3 remove NaturalPerson.city
4 pullUp LegalPerson.street
5 pullUp LegalPerson.city

```

Ukázka kódu 10.2: Vyextrahování společného předka

Pomocí operace **EXTRACT SUPERCLASS** z obou tříd vyextrahujeme společný atribut *country* do nové třídy *Party*, která je následně nastavena jako rodič těchto tříd. Ostatní společné atributy jsou kombinací operací **REMOVE PROPERTY** a **PULL UP PROPERTY** přesunuty z obou potomků do rodičovské třídy *Party*. Požadovanou podobu modelu zachycuje obrázek 10.2.



Obrázek 10.2: Upravený testovací model - vznik hierarchie

10.1.3 Extrahování atributů

Mezi další nevýhody původního modelu, ale i upraveného, patří duplikace samotných adres osob. V případě, že více osob sdílí stejnou adresu, dochází k uložení identických údajů na různá místa. Navíc můžeme požadovat uložení více druhů adres - fakturační a kontaktní, čímž by, v případě použití současného způsobu uložení adres, docházelo k velkému vzniku duplicit.

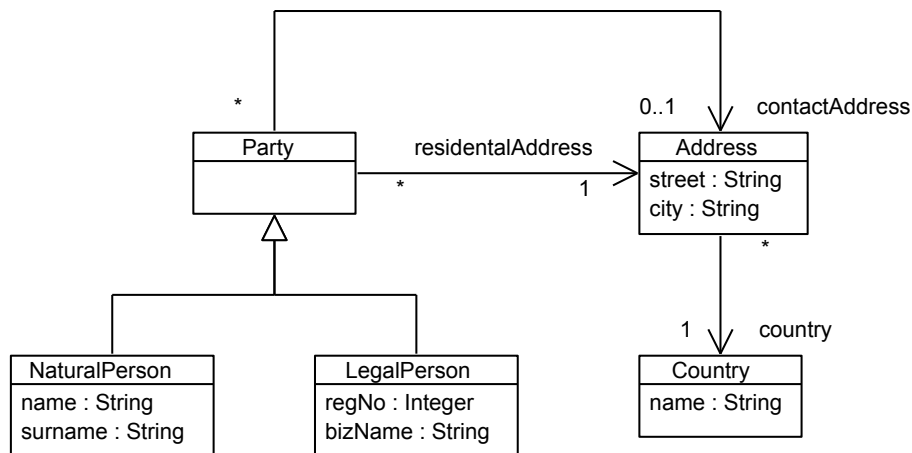
```

1 extract Party.country / residentialAddress -> Address.resident
2 remove Address.resident
3 move Party.street / residentialAddress -> Address
4 move Party.city / residentialAddress -> Address
5 add Party.contactAddress : Address

```

Ukázka kódu 10.3: Přesun adresy

Využitím operace **EXTRACT CLASS** provedeme delegování zodpovědnosti za udržování informací o adresách na novou třídu pojmenovanou *Address*. Jelikož podle specifikace **EXTRACT CLASS** dochází automaticky ke vzniku obousměrné asociace mezi zdrojovou a vyextrahovanou třídou, odstraníme tu část asociace vedoucí z *Address* reprezentovanou atributem *resident*. Ostatní atributy (*street*, *city*) přesuneme do vyextrahované třídy *Address* aplikací operace **MOVE PROPERTY** přes vzniklou asociaci *residentialAddress*. Abychom vyhověli požadavku pro podporu fakturační a kontaktní adresy, vytvoříme v *Party* nepovinný atribut *contactAddress* operací **ADD PROPERTY**.



Obrázek 10.3: Upravený testovací model - extrakce atributů

10.2 Jednotkové testy

Ověření správné implementace typového systému jazyku Ops je provedeno pomocí jednotkového testování. Vznikla sada testovacích příkladů obsahující problematické sekvence operací, na jejichž základě ověříme, že se implementace chová na vybraných příkladech dle formální specifikace typového systému z kapitoly 7.

Testovací příklad je zachycen na ukázce 10.4, která obsahuje ověření správné implementace typového pravidla **T-ExtCls** operace **EXTRACT CLASS**. Před samotným otestováním operace, dojde k nastavení vhodných podmínek pro samotný test. Řádky 1 až 7 testovacího příkladu vytvoří třídy: *Food* s atributem *name* a *Fruit* s atributem *origin*. Dále je mezi oběmi třídami nastaven vztah potomek-rodíč, kdy *Fruit* je potomkem třídy *Food*. Řádek 9 obsahuje aplikaci operace **EXTRACT CLASS** pro vyextrahování atributu *origin* ze třídy *Fruit* do nové třídy *Leaf*. Podle specifikace má mezi oběmi třídami vzniknout obousměrná asociace, která na straně *Fruit* ponese pojmenování *name*, naopak na straně nově vzniklé třídy *Leaf* ponese jméno *fruit*.

```

1 add Food
2 add Fruit
3
4 add Food.name : Integer
5 add Fruit.origin : Integer
6
7 set Fruit <: Food
8
9 extract Fruit.origin / name -> Leaf.fruit //chybna operace

```

Ukázka kódu 10.4: Testovací program v Ops

Dle předpokladů typového pravidla **T-ExtCls**, mimo jiné, nesmí existovat asociační atribut *name* ve třídě *Fruit* ani v její předcích či potomcích, jinak by vznikla duplicita. Tento předpoklad je porušen, protože předek třídy *Fruit* již obsahuje atribut *name*. Implementace typového systému upozorní uživatele chybovým hlášením specifikující nesplněný předpoklad na řádce 9. Této skutečnosti využijeme při vzniku jednotkového testu:

```

1 @Test
2 def extractClass1() {
3   testFile("extractClass1.ops")
4   assertConstraints(
5     inLine(9, "Property name already exists in class hierarchy of Fruit")
6   )
7 }

```

Ukázka kódu 10.5: Implementace metody pro jednotkové testování

V jednotkovém testu metodou *testFile()* načteme příslušný Ops program. Načtením souboru dojde ke zkontrolování jeho syntaktické správnosti, ale také k sestavení abstraktního stromu nad nímž je provedena typová kontrola. Test úspěšně končí, vyskytuje-li se na řádce 9 chyba o existenci atributu *name* v některé ze tříd v hierarchii *Fruit*.

Sadou jednotkových testů jsme ověřili, že se implementace typového systému chová korektně na vybraných testovacích datech. Avšak nemáme zaručenu její 100% správnost.

Kapitola 11

Závěr

Cílem této práce bylo, v rámci frameworku MigDb, vytvořit doménově specifický jazyk Ops zjednodušující refaktoring aplikačního modelu struktury databáze využitím operací. Vznikla formální specifikace vyvíjeného jazyku definující jeho syntaxi, operační sémantiku a typový systém. Na základě formální specifikace byly dokázány vlastnosti *Terminace* a *Soundness* zaručující, že vyhodnocení libovolného syntakticky správného Ops programu vždy skončí po konečně mnoho krocích a výsledkem je hodnota.

Dle formální specifikace byl naimplementován jazykový plug-in, do vývojového prostředí Eclipse IDE, který umožňuje snadnou tvorbu a editaci programů napsaných v jazyce Ops. Vzniklý plug-in byl integrován s ostatními částmi MigDb, tím bylo docíleno uživatelsky příjemnější manipulace se samotným frameworkem. Nicméně, celý framework MigDb by bylo možné v budoucích verzích rozšířit o podporu dalších typů refaktoringů a manipulaci se vstupní strukturou zachycující jednotlivé entity software.

V závěru práce byl věnován prostor pro testování plug-inu. Bylo ukázáno, že se implementace chová korektně, v souladu s formální specifikací, na testovacích příkladech. Všechny body ze zadání práce proto považuji za zdárně dokončené.

Literatura

- [1] CARDELLI, L. Type systems. *ACM Computing Surveys*. 1996, 28, 1, s. 263–264.
- [2] CURINO, C. A. Graceful database schema evolution: the prism workbench. *Proceedings of the VLDB Endowment*. 2008, s. 761–772.
- [3] David H. Hansson. *activerecord*, *RubyGems.org* [online]. 2014. [cit. 2. 5. 2014]. Dostupné z: <<https://rubygems.org/gems/activerecord>>.
- [4] David H. Hansson. *Ruby On Rails* [online]. 2014. [cit. 2. 5. 2014]. Dostupné z: <<http://rubyonrails.org>>.
- [5] DOMÍNGUEZ, E. MeDEA: A database evolution architecture with traceability. *Data & Knowledge Engineering*. 2008, s. 419–441.
- [6] FOWLER, M. *Domain-Specific Languages*. Addison-Wesley Professional, 1. vyd., 2010. ISBN 978-0-321-71294-3.
- [7] FOWLER, M. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1. vyd., 1999. ISBN 978-0-201-48567-7.
- [8] HERRMANNSSDOERFER, M. COPE-automating coupled evolution of metamodels and models. In *ECOOP 2009–Object-Oriented Programming*. Springer, 2009. s. 52–76.
- [9] HICK, J.-M. Database application evolution: a transformational approach. *Data & Knowledge Engineering*. 2006, s. 534–558.
- [10] JBoss Community. *Hibernate. Everything data*. [online]. 2014. [cit. 2. 5. 2014]. Dostupné z: <<http://www.hibernate.org>>.
- [11] JBoss Community. *Hibernate ORM documentation: Persistent Classes* [online]. 2014. [cit. 2. 5. 2014]. Dostupné z: <<https://docs.jboss.org/hibernate/orm/3.3/reference/en/html/persistent-classes.html>>.
- [12] JEŽEK, J. *Modelem řízená evoluce objektů*. Bakalářská práce, ČVUT, 2012.
- [13] LUKEŠ, M. *Transformace objektových modelů*. Bakalářská práce, ČVUT, 2011.
- [14] LUKEŠ, M. *Dokončení projektu MigDb*. Diplomová práce, ČVUT, 2014.
- [15] LUKSCH, D. *Refactoring Catalogue of the MigDB Framework*. Bakalářská práce, ČVUT, 2013.

- [16] MAZANEC, M. *Jazyky pro textové modelování*. Bakalářská práce, ČVUT, 2012.
- [17] Nathan Voxland. *Liquibase, Database Refactoring* [online]. 2014. [cit. 2. 5. 2014]. Dostupné z: <<http://www.liquibase.org>>.
- [18] OMG. *Model Driven Architecture* [online]. 2014. [cit. 2. 5. 2014]. Dostupné z: <<http://www.omg.org/mda/>>.
- [19] OMG. *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification* [online]. 2014. [cit. 2. 5. 2014]. Dostupné z: <<http://www.omg.org/spec/QVT/1.1/>>.
- [20] OMG. *XMI* [online]. 2014. [cit. 2. 5. 2014]. Dostupné z: <<http://www.omg.org/spec/xmi/2.4.1/>>.
- [21] PIERCE, B. C. *Types and Programming Languages*. MIT press, 1. vyd., 2002. ISBN 978-0-262-16209-1.
- [22] TARANT, P. *Modelem řízená evoluce databáze*. Bakalářská práce, ČVUT, 2012.
- [23] The Eclipse Foundation. *Eclipse* [online]. 2014. [cit. 2. 5. 2014]. Dostupné z: <<http://www.eclipse.org>>.
- [24] The Eclipse Foundation. *Xtend - Modernized Java* [online]. 2014. [cit. 2. 5. 2014]. Dostupné z: <<http://www.eclipse.org/Xtend/>>.
- [25] The Eclipse Foundation. *Xtext - Language Development Made Easy!* [online]. 2014. [cit. 2. 5. 2014]. Dostupné z: <<http://www.eclipse.org/Xtext/>>.
- [26] The Eclipse Labs. *The Modeling Workflow Engine 2* [online]. 2014. [cit. 2. 5. 2014]. Dostupné z: <<http://www.eclipse.org/Xtext/documentation.html#MWE2>>.
- [27] The Eclipse Labs. *xtext-utils - A collection of useful utilities and samples for Xtext based projects* [online]. 2014. [cit. 2. 5. 2014]. Dostupné z: <<https://code.google.com/a/eclipselabs.org/p/xtext-utils/>>.
- [28] TURBAK, F. A. *Design Concepts in Programming Languages*. MIT press, 1. vyd., 2008. ISBN 978-0-262-20175-9.
- [29] ČVUT. *Teorie programovacích jazyků* [online]. 2014. [cit. 2. 5. 2014]. Dostupné z: <<https://edux.feld.cvut.cz/courses/A4M36TPJ>>.

Příloha A

Jazyk Ops

A.1 Gramatika

<i>Operation</i>	::= <i>AddClass</i> <i>RemoveClass</i> <i>AddProperty</i> <i>RemoveProperty</i> <i>RenameProperty</i> <i>MoveProperty</i> <i>PullUpProperty</i> <i>PushDownProperty</i> <i>SetParent</i> <i>RemoveParent</i> <i>ExtractClass</i> <i>ExtractSubClass</i> <i>ExtractSuperClass</i> <i>unit</i> <i>error</i> <i>Operation ; ? Operation</i>
<i>AddClass</i>	::= <i>add Class</i>
<i>RemoveClass</i>	::= <i>remove Class</i>
<i>AddProperty</i>	::= <i>add Class.Property : TypeName</i> <i>add Class.Property : TypeName [Number]</i> <i>add Class.Property : TypeName [Number . . Number]</i>
<i>RemoveProperty</i>	::= <i>remove Class.Property</i>
<i>RenameProperty</i>	::= <i>rename Class.Property -> Property</i>
<i>MoveProperty</i>	::= <i>move Class.Property / Property -> Class</i>
<i>PullUpProperty</i>	::= <i>pullUp Class.Property</i>
<i>PushDownProperty</i>	::= <i>pushDown Class.Property</i>
<i>SetParent</i>	::= <i>set Class <: Class</i>
<i>RemoveParent</i>	::= <i>remove Class <: Class</i>

$ExtractClass \quad ::= \text{extract } Class.Property / Property \rightarrow Class.Property$
 $ExtractSubClass \quad ::= \text{extractSub } Class.Property \rightarrow Class$
 $ExtractSuperClass \quad ::= \text{extractSuper } \{Class^+\}.Property \rightarrow Class$
 $TypeName \quad ::= Primitive$
 $\quad \quad \quad | \quad Class$

A.2 Operační sémantika

$$\frac{}{(\mathcal{S}, \text{add } c) \Rightarrow (\mathcal{S}[c \mapsto (Top, \emptyset)], \text{unit})} \quad (\text{A.1})$$

$$\frac{}{(\mathcal{S}, \text{remove } c) \Rightarrow (\mathcal{S} \setminus (c, \mathcal{S}(c)), \text{unit})} \quad (\text{A.2})$$

$$\frac{}{(\mathcal{S}, \text{add } c.p : t) \Rightarrow (\mathcal{S}, \text{add } c.p : t[0..1])} \quad (\text{A.3})$$

$$\frac{}{(\mathcal{S}, \text{add } c.p : t[n]) \Rightarrow (\mathcal{S}, \text{add } c.p : t[0..n])} \quad (\text{A.4})$$

$$\frac{\mathcal{S}(c) = (d, ps)}{(\mathcal{S}, \text{add } c.p : t[n..m]) \Rightarrow (\mathcal{S}[c \mapsto (d, ps \cup \{p : t[n..m]\})], \text{unit})} \quad (\text{A.5})$$

$$\frac{\mathcal{S}(c) = (d, ps)}{(\mathcal{S}, \text{remove } c.p) \Rightarrow (\mathcal{S}[c \mapsto (d, ps \setminus \{\mathcal{S}(c.p)\})], \text{unit})} \quad (\text{A.6})$$

$$\frac{\mathcal{S}(c.p) = p : t[n..m]}{(\mathcal{S}, \text{rename } c.p \rightarrow q) \Rightarrow (\mathcal{S}, \text{remove } c.p; \text{add } c.q : t[n..m])} \quad (\text{A.7})$$

$$\frac{\mathcal{S}(c.p) = p : t[n..m]}{(\mathcal{S}, \text{move } c.p / q \rightarrow d) \Rightarrow (\mathcal{S}, \text{remove } c.p; \text{add } d.p : t[n..m])} \quad (\text{A.8})$$

$$\frac{\text{par}_{\mathcal{S}}(c) = d \quad \mathcal{S}(c.p) = p : t[n..m]}{(\mathcal{S}, \text{pullUp } c.p) \Rightarrow (\mathcal{S}, \text{remove } c.p; \text{add } d.p : t[n..m])} \quad (\text{A.9})$$

$$\frac{\mathcal{S}(c.p) = p : t[n..m]}{(\mathcal{S}, \text{pushDown } c.p) \Rightarrow (\mathcal{S}, \text{remove } c.p; \Delta)} \quad (\text{A.10})$$

kde $\Delta = \text{add } c_1.p : t[n..m]; \dots; \text{add } c_j.p : t[n..m]$ pro $c_i \in \text{sub}_{\mathcal{S}}(c)$

$$\frac{\mathcal{S}(c) = (e, ps)}{(\mathcal{S}, \text{set } c <: d) \Rightarrow (\mathcal{S}[c \mapsto (d, ps)], \text{unit})} \quad (\text{A.11})$$

$$\frac{\mathcal{S}(c) = (d, ps)}{(\mathcal{S}, \text{remove } c <: d) \Rightarrow (\mathcal{S}[c \mapsto (Top, ps)], \text{unit})} \quad (\text{A.12})$$

$$\frac{}{(\mathcal{S}, \text{extract } c.p / q \rightarrow d.r) \Rightarrow (\mathcal{S}, \text{add } d; \text{add } d.r : c[1..1]; \text{add } c.q : d[1..1]; \text{move } c.p / q \rightarrow d)} \quad (\text{A.13})$$

$$\frac{}{(\mathcal{S}, \text{extractSub } c.p \rightarrow d) \Rightarrow (\mathcal{S}, \text{add } d; \text{set } d <: c; \text{pushDown } c.p)} \quad (\text{A.14})$$

$$\frac{}{(\mathcal{S}, \text{extractSuper } \{c_1 \dots c_j\}.p \rightarrow d \Rightarrow (\mathcal{S}, \text{add } d; \Delta; \text{pullUp } c_1.p)} \quad (\text{A.15})$$

kde $\Delta = \text{set } c_1 <: d; \dots; \text{set } c_i <: d; \text{remove } c_2.p; \dots; \text{remove } c_j.p$

$$\frac{(\mathcal{S}, o_1) \Rightarrow (\mathcal{S}', o'_1)}{(\mathcal{S}, o_1; o_2) \Rightarrow (\mathcal{S}', o'_1; o_2)} \quad (\text{A.16})$$

$$\frac{}{(\mathcal{S}, \text{unit}; o_2) \Rightarrow (\mathcal{S}, o_2)} \quad (\text{A.17})$$

$$\frac{}{(\mathcal{S}, \text{error}; o_2) \Rightarrow (\mathcal{S}, \text{error})} \quad (\text{A.18})$$

A.3 Typový systém

$$\frac{}{\Gamma \vdash \epsilon : \text{Operation}} \quad (\text{A.19})$$

$$\frac{c \notin \Gamma \quad \Gamma, c \vdash o : \text{Operation}}{\Gamma \vdash \text{add } c; o : \text{Operation}} \quad (\text{A.20})$$

$$\frac{c \in \Gamma \quad \text{prop}_\Gamma(c) = \text{sub}_\Gamma(c) = \emptyset \quad d.q : c[n..m] \notin \Gamma \quad \Gamma \setminus \{c\} \vdash o : \text{Operation}}{\Gamma \vdash \text{remove } c; o : \text{Operation}} \quad (\text{A.21})$$

$$\frac{\Gamma \vdash \text{add } c.p : t[0..1]; o : \text{Operation}}{\Gamma \vdash \text{add } c.p : t; o : \text{Operation}} \quad (\text{A.22})$$

$$\frac{\Gamma \vdash \text{add } c.p : t[0..n]; o : \text{Operation}}{\Gamma \vdash \text{add } c.p : t[n]; o : \text{Operation}} \quad (\text{A.23})$$

$$\frac{c \in \Gamma \quad (t \in \Gamma \vee t \in \text{Primitive}) \quad m \geq n \geq 0 \quad m > 0 \quad p \notin \text{allprop}_\Gamma(c) \quad \Gamma, c.p : t[n..m] \vdash o : \text{Operation}}{\Gamma \vdash \text{add } c.p : t[n..m]; o : \text{Operation}} \quad (\text{A.24})$$

$$\frac{c, c.p : t[n..m] \in \Gamma \quad \Gamma \setminus \{c.p : t[n..m]\} \vdash o : \text{Operation}}{\Gamma \vdash \text{remove } c.p; o : \text{Operation}} \quad (\text{A.25})$$

$$\frac{q \notin \text{allprop}_\Gamma(c) \quad \frac{c, c.p : t[n..m] \in \Gamma \quad \Gamma \setminus \{c.p : t[n..m]\}, c.q : t[n..m] \vdash o : \text{Operation}}{\Gamma \vdash \text{rename } c.p \rightarrow q; o : \text{Operation}}}{\Gamma \vdash \text{rename } c.p \rightarrow q; o : \text{Operation}} \quad (\text{A.26})$$

$$\frac{p \notin \text{allprop}_\Gamma(d) \quad \frac{c, d, c.p : t[n..m], c.q : d[1..1] \in \Gamma \quad \Gamma \setminus \{c.p : t[n..m]\}, d.p : t[n..m] \vdash o : \text{Operation}}{\Gamma \vdash \text{move } c.p / q \rightarrow d; o : \text{Operation}}}{\Gamma \vdash \text{move } c.p / q \rightarrow d; o : \text{Operation}} \quad (\text{A.27})$$

$$\frac{c, c.p : t[n..m], c <: d \in \Gamma \quad p \notin \text{allprop}_{\Gamma \setminus \{c <: d\}}(d) \quad \Gamma \setminus \{c.p : t[n..m]\}, d.p : t[n..m] \vdash o : \text{Operation}}{\Gamma \vdash \text{pullUp } c.p; o : \text{Operation}} \quad (\text{A.28})$$

$$\frac{\Gamma \setminus \{c.p : t[n..m]\}, d_1.p : t[n..m], \dots, d_j.p : t[n..m] \vdash o : \text{Operation} \quad \frac{c, c.p : t[n..m] \in \Gamma \quad \text{sub}_\Gamma(c) \neq \emptyset}{\Gamma \vdash \text{pushDown } c.p; o : \text{Operation}}}{\Gamma \vdash \text{pushDown } c.p; o : \text{Operation}} \quad (\text{A.29})$$

kde $d_i \in \text{sub}_\Gamma(c)$

$$\frac{c, d \in \Gamma \quad c \notin \text{preds}_\Gamma(d) \quad \text{allprop}_{\Gamma \setminus \{c <: d\}}(c) \cap \text{parprop}_\Gamma(d) = \emptyset \quad \Gamma, c <: d \vdash o : \text{Operation}}{\Gamma \vdash \text{set } c <: d; o : \text{Operation}} \quad (\text{A.30})$$

$$\frac{c, d, c <: d \in \Gamma \quad \Gamma \setminus \{c <: d\} \vdash o : \text{Operation}}{\Gamma \vdash \text{remove } c <: d; o : \text{Operation}} \quad (\text{A.31})$$

$$\frac{\Gamma' = \Gamma \setminus \{c.p : t[n..m]\}, d, c.q : d[1..1], d.r : c[1..1] \quad \Gamma', d.p : t[n..m] \vdash o : \text{Operation} \quad \frac{c, c.p : t[n..m] \in \Gamma \quad d \notin \Gamma \quad q \notin \text{allprop}_\Gamma(c)}{\Gamma \vdash \text{extract } c.p / q \rightarrow d.r; o : \text{Operation}}}{\Gamma \vdash \text{extract } c.p / q \rightarrow d.r; o : \text{Operation}} \quad (\text{A.32})$$

$$\frac{c, c.p : t[n..m] \in \Gamma \quad d \notin \Gamma \quad \Gamma' = \Gamma \setminus \{c.p : t[n..m]\}, d, d <: c \quad \Gamma', d_1.p : t[n..m], \dots, d_j.p : t[n..m] \vdash o : \text{Operation}}{\Gamma \vdash \text{extractSub } c.p \rightarrow d; o : \text{Operation}} \quad (\text{A.33})$$

kde $d_i \in \text{sub}_{\Gamma'}(c)$

$$\frac{\Gamma', d, c_1 <: d, \dots, c_j <: d, d.p : t_1[n_1..m_1] \vdash o : \text{Operation} \quad \frac{c_1, \dots, c_j \in \Gamma \quad d, c_1 <: e, \dots, c_j <: e \notin \Gamma \quad |\{c_1, \dots, c_j\}| = j \quad c_1.p : t_1[n_1..m_1], \dots, c_j.p : t_1[n_1..m_1] \in \Gamma \quad \Gamma' = \Gamma \setminus \{c_1.p : t_1[n_1..m_1], \dots, c_j.p : t_1[n_1..m_1]\}}{\Gamma \vdash \text{extractSuper } \{c_1 \dots c_j\}.p \rightarrow d; o : \text{Operation}}}{\Gamma \vdash \text{extractSuper } \{c_1 \dots c_j\}.p \rightarrow d; o : \text{Operation}} \quad (\text{A.34})$$

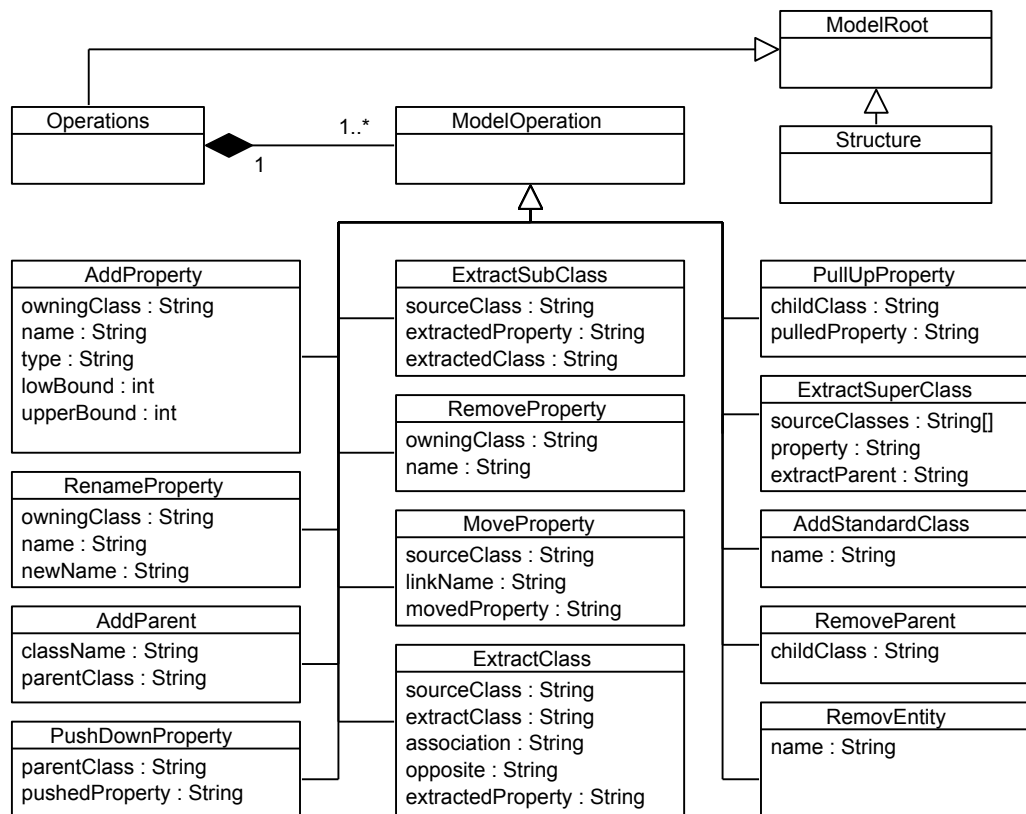
$$\frac{\Gamma \vdash o : \text{Operation}}{\Gamma \vdash \text{unit}; o : \text{Operation}} \quad (\text{A.35})$$

$$\frac{\Gamma \vdash o : \text{Operation}}{\Gamma \vdash \text{error}; o : \text{Operation}} \quad (\text{A.36})$$

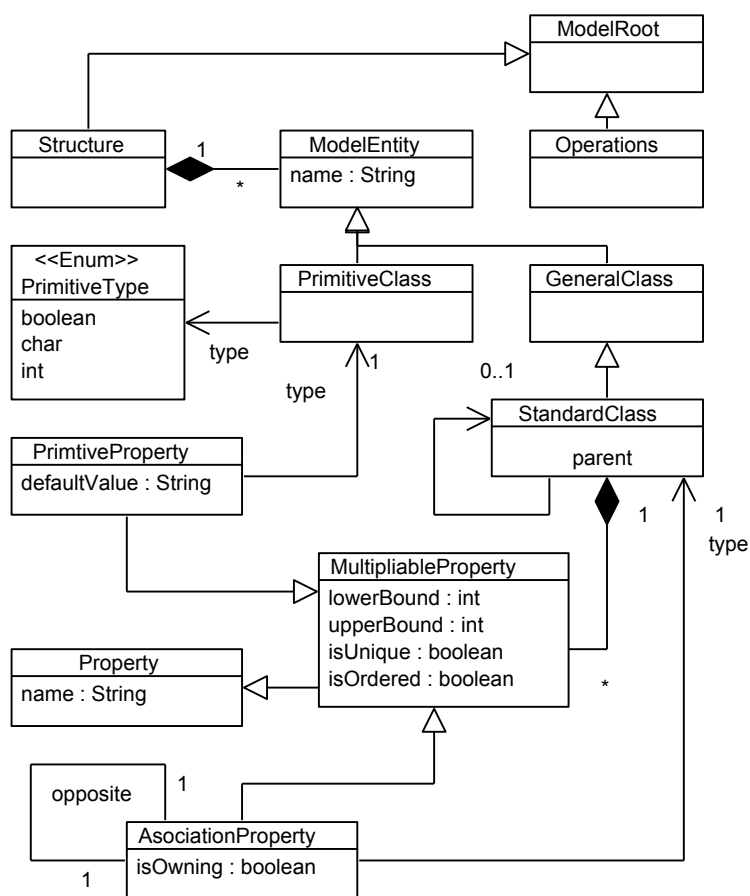
Příloha B

Metamodely MigDb

B.1 Aplikační metamodel

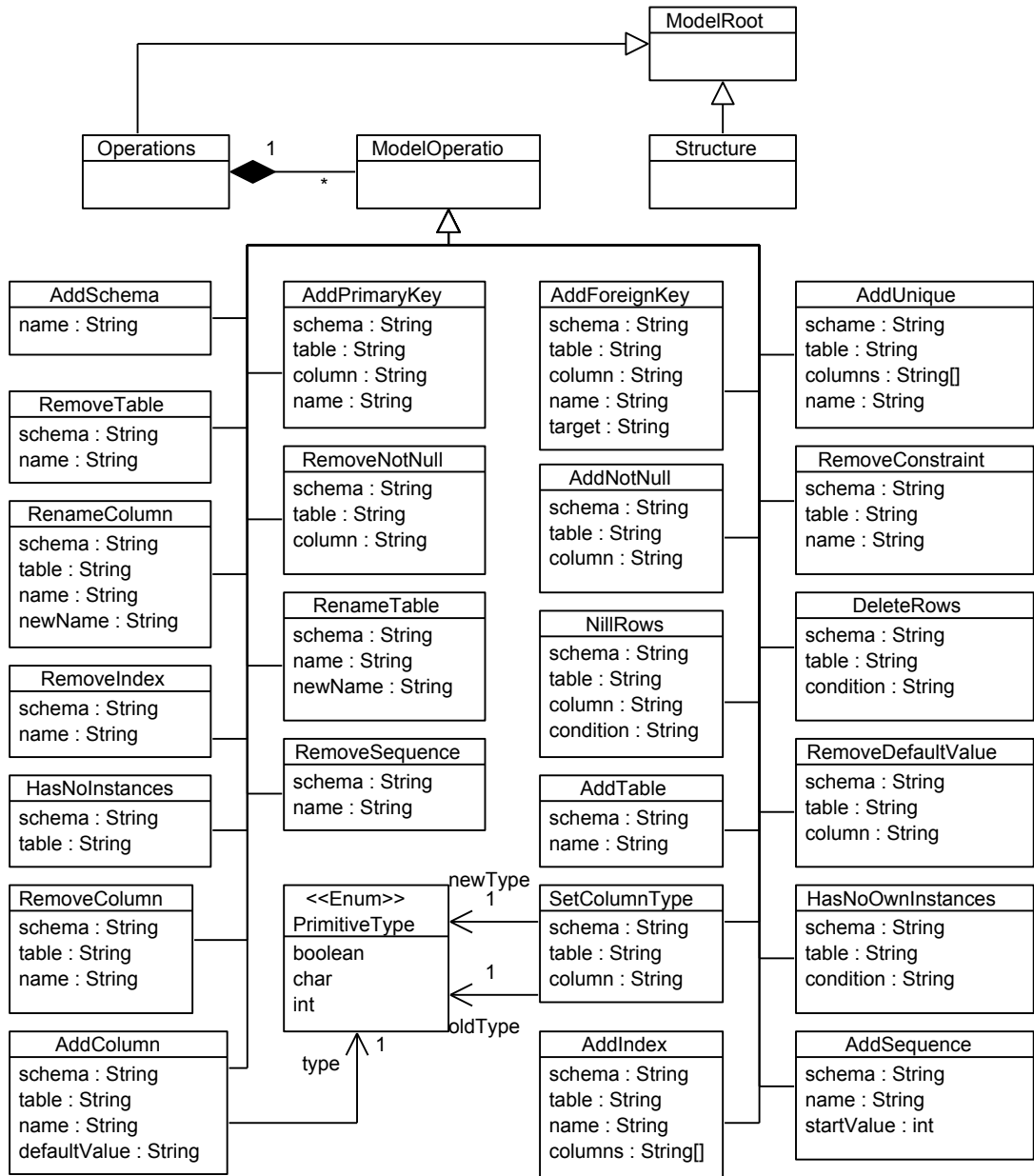


Obrázek B.1: Aplikační metamodel frameworku MigDb - operace

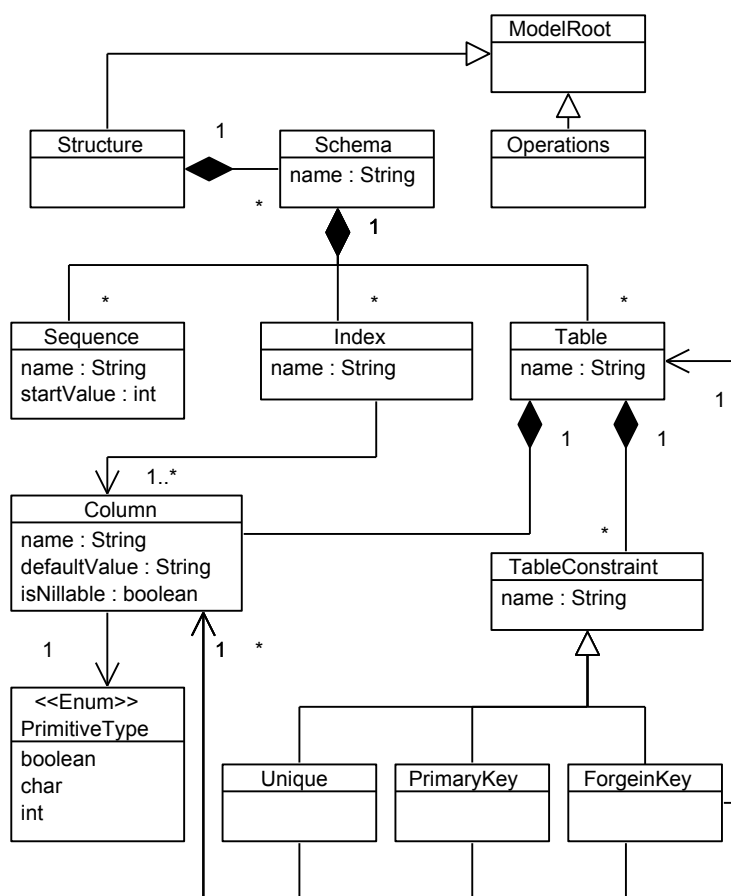


Obrázek B.2: Aplikační metamodel frameworku MigDb - struktura

B.2 Databázový metamodel



Obrázek B.3: Databázový metamodel frameworku MigDb - operace



Obrázek B.4: Databázový metamodel frameworku MigDb - struktura

Příloha C

Instalační a uživatelská příručka

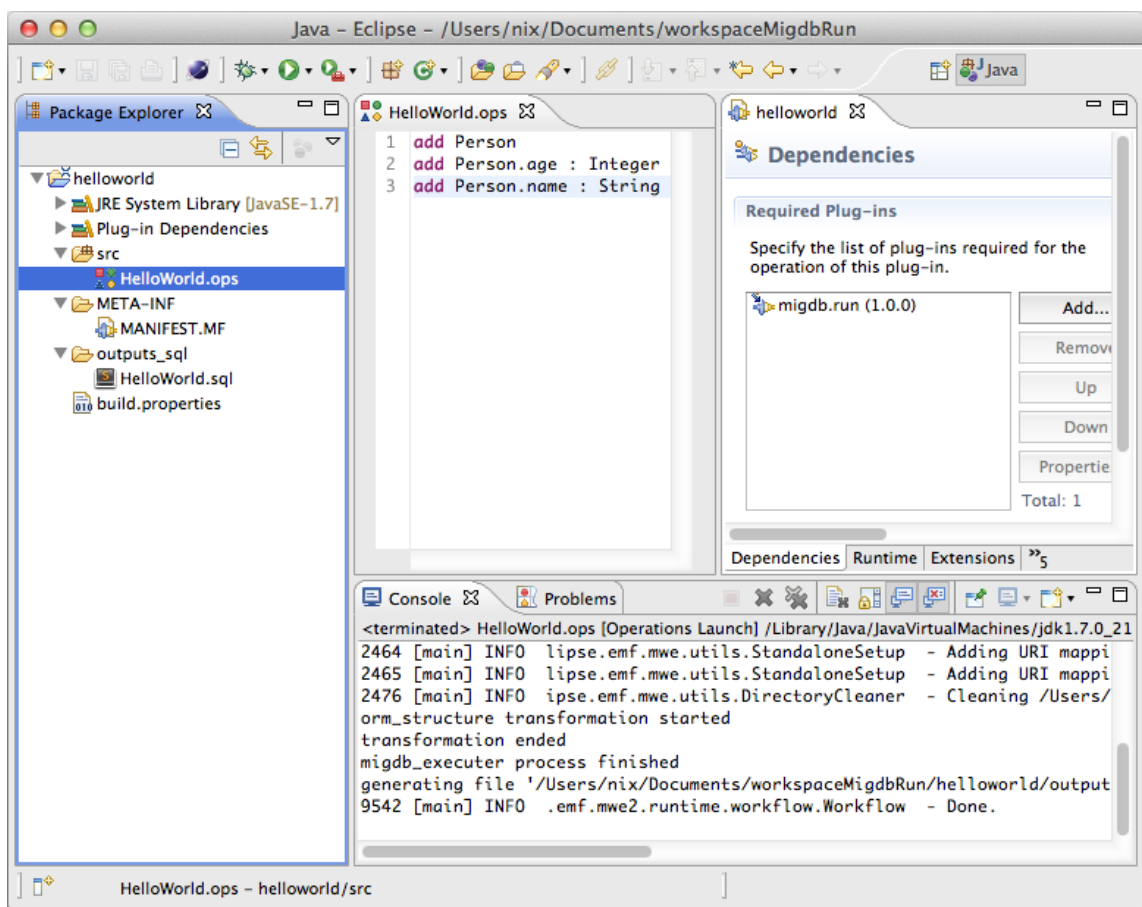
Příložené CD obsahuje předkonfigurované vývojové prostředí Eclipse Modeling Tools ve verzi **Indigo SR2**, které lze nalézt v adresáři `\eclipse`. Toto prostředí stačí zkopírovat do svého domovského adresáře a spustit. Nutnou prerekvizitou je nainstalovaná Java minimálně verze 1.6.

C.1 Vytvoření Hello World

1. Spusťte Eclipse Modeling Tools pocházející z příloženého CD.
2. Pomocí menu *File* → *New* → *Others...* vytvořte nový **Plug-in Project** pojmenovaný `HelloWorld`, v průvodci nechte defaultní nastavení a klikněte na tlačítko *Finish*.
3. Po vytvoření nového plug-in projektu najdete pomocí *Package Exploreru*, v právě vytvořeném projektu, soubor `META-INF\MANIFEST.MF` a otevřete jej.
4. Zvolte záložku *Dependencies* a tlačítkem *Add...* přidejte do *Required Plug-ins* balíček `migdb.run` a změny v souboru uložte.
5. Využitím nabídky menu: *New* → *Others...* → *MigDb* → *Operations* vytvořte v projektu `HelloWorld` prázdný *Ops* soubor pojmenovaný `HelloWorld`.
6. Vznikem souboru dojde k zobrazení dialogového okna s následujícího hláškou „*Do you want to add the Xtext nature to the project 'HelloWorld'?*“, potvrďte zvolením možnosti *Yes*, čímž dojde k načtení *Xtext* knihoven do aktuálního projektu.
7. Otevřete soubor `HelloWorld.ops`, není-li již otevřen, a napište do něj následující:

```
add Person
add Person.age : Integer
add Person.name : String
```

8. Za pomoci *Package Exploreru* najdete soubor `HelloWorld.ops` a pravým tlačítkem myši nad ním zvolte *Run As* → *Operations*, tím dojde ke spuštění pracovního toku `MigDb`, jehož průběh bude postupně vypisován v pohledu *Console*.
9. Úspěšným dokončením pracovního toku dojde k vytvoření nového adresáře `outputs_sql` ve kterém se nachází právě vygenerovaný SQL skript `HelloWorld.sql`.



Obrázek C.1: Vytvoření Hello World programu v jazyce Ops

Příloha D

Seznam použitých zkratk

AST	Abstraktní syntaktický strom
DSL	Doménově specifický jazyk
MDA	Model Driven Architecture
ORM	Objektově-relační mapování

Příloha E

Obsah příloženého CD

Příložené CD má následující strukturu:

<code>\bin</code>	-	Plug-iny projektu MigDb včetně jazyku Ops.
<code>\eclipse</code>	-	Předkonfigurované prostředí Eclipse IDE.
<code>\helloworld</code>	-	Ukázkový projekt s Ops příklady.
<code>\src</code>	-	Zdrojový kód implementace Ops.
<code>\text</code>	-	Text této diplomové práce v \LaTeX .