ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Podpora konceptuálního modelování pro jazyk Haskell
Student:	Bc. Marek Suchánek
Vedoucí:	Ing. Robert Pergl, Ph.D.
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2017/18

Pokyny pro vypracování

Cílem práce je návrh a implementace systému pro vytváření doménových modelů v jazyku Haskell s podporou vizualizace a validace, podobného systémům popsaným v literatuře [2]-[6].

1. Proveďte stručnou rešerši základů konceptuálního modelování, dále pak způsobů vizualizace, verifikačních a validačních přístupů konceptuálních modelů.
2. Analyzujte možnosti jazyka Haskell z hlediska kódování konceptuálních modelů.
3. Navrhněte systém pro kódování konceptuálních doménových modelů v jazyce Haskell a jejich vizualizace, verifikace a validace. Na základě provedené rešerše zvolte ve spolupráci s vedoucím vhodnou implementační platformu (webová aplikace/desktopová).
4. Implementujte prototyp systému a vytvořte metodiku pro využití systému pro validaci modelu se zákazníkem.
5. Prototyp demonstrujte na případové studii dostatečně komplexního doménového modelu. Demonstrujte převod konceptuálního modelu do kódu a implementované vizualizace a verifikace.
6. Vyhodnoťte výsledky.

Seznam odborné literatury

[1] http://ccmi.fit.cvut.cz/programovaci-jazyky/haskell/
[2] http://alloy.mit.edu/alloy/
[3] https://sourceforge.net/projects/useocl/
[4] http://overturetool.org/
[5] https://en.wikipedia.org/wiki/B-Method
[6] http://czt.sourceforge.net/

L.S.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 21. listopadu 2016

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra softwarového inženýrství

Diplomová práce

Podpora konceptuálního modelování pro jazyk Haskell

Bc. Marek Suchánek

Vedoucí práce: Ing. Robert Pergl, Ph.D.

1. května 2017

Poděkování

Děkuji Ing. Robertu Perglovi, Ph.D., vedoucímu této diplomové práce, za odborné konzultace a řízení Centra pro konceptuální modelování a implementace na Fakultě informačních technologií ČVUT v Praze. Za konzultování částí této práce děkuji Ing. Davidu Buchtelovi, Ph.D. Mé poděkování náleží také celé Fakultě informačních technologií ČVUT v Praze za vše během pěti let mého studia a dále pak samotnému ČVUT v Praze a NTK za poskytnutí licencí k nástrojům a přístupu ke zdrojům pro tvorbu této práce. Velké díky patří též mé rodině a přátelům za jejich dlouholetou podporu při studiu.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 1. května 2017 .....

Odkaz na tuto práci

Suchánek, Marek. *Podpora konceptuálního modelování pro jazyk Haskell.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.

Abstrakt

Diplomová práce se zabývá problematikou provázanosti konceptuálního modelování s implementací informačních systémů a navrhuje řešení pro reprezentaci konceptuálních modelů přímo v programovacím jazyce Haskell včetně jejich vizualizace a postupů validace a verifikace. V rámci práce jsou rozebrány základní pojmy a významné přístupy v oblasti konceptuálního modelování a formální specifikace modelů, stejně jako nástroje pro jejich podporu. Dále jsou v práci analyzovány vybrané vlastnosti programovacích jazyků a možnosti jejich využití pro konceptuální modelování, požadavky na reprezentaci konceptuálních modelů a vlastní expresivita jazyka Haskell. Systém, navržený na základě analýzy a rešerše, ve formě implementovaného prototypu v jazyce Haskell je demonstrován na případové studii a jsou diskutovány jeho výhody a nevýhody z hlediska výsledného SW produktu i projektového řízení. Výsledkem práce je zdokumentovaný open-source prototyp systému včetně metodiky jeho užití pro tvorbu, verifikaci a validaci konceptuálních modelů v jazyce Haskell. Práce také obsahuje návrh budoucího vývoje a rozvoje systému i další možnosti výzkumu v této oblasti.

Klíčová slova konceptuální modelování, Haskell, verifikace, validace, vizualizace, MDD, formální specifikace, kompilátorem řízené modelování

Abstract

The diploma thesis deals with the interconnection of conceptual modelling with the implementation of information systems and proposes solution to represent conceptual models directly in the Haskell programming language including their visualization as well as validation and verification procedures. Basic concepts and significant approaches to conceptual modelling and formal specification with their supporting tools are discussed in this work. Furthermore, the work contains analysis of the selected properties of programming languages to be used for conceptual modelling, requirements for representation of conceptual models and Haskell's own expressiveness is analyzed too. The proposed solution, which is based on previous discussion and analysis, in the form of an implemented Haskell library prototype is demonstrated in a case study and its advantages and disadvantages are discussed in terms of the resulting software product and project management. The result of the thesis is a documented open-source prototype, including its methodology for the construction, validation and verification of conceptual models. The thesis also contains proposals for the future development of the system and other research possibilities in this topic.

Keywords conceptual modelling, Haskell, verification, validation, visualization, MDD, formal specification, compiler-driven modelling

Obsah

Úvod **1**
Cíle práce 2
Metodika a struktura práce 2

I Teoretická část **5**

1 Konceptuální modelování **7**
1.1 Úvod do konceptuálního modelování 7
1.2 Verifikace a validace 11
1.3 Výhody a možné problémy 12
1.4 Historický vývoj 14
1.5 Souvislost s ontologií a programováním 16

2 Metody zápisu konceptuálních modelů **21**
2.1 Reprezentace konceptuálních modelů 21
2.2 Grafická reprezentace 22
2.3 Textová reprezentace 26
2.4 Společné rysy metod 32

3 Přehled nástrojů **33**
3.1 Alloy a Alloy Analyzer 33
3.2 Overture 34
3.3 Enterprise Architect 35
3.4 TestEra 36
3.5 CZT: Community Z Tools 37
3.6 Menthor Editor 38
3.7 USE 39

II Praktická část **41**

4 Uplatnění programovacích jazyků pro konceptuální modelování **43**
4.1 Programovací jazyk jako prostředek modelování 43
4.2 Vlastnosti jazyků 45
4.3 Existující řešení 48

5 Analýza požadavků na reprezentaci modelů **53**
5.1 Entity 53
5.2 Vztahy 54
5.3 Omezení 55
5.4 Chování 56
5.5 Nefunkční požadavky 56
5.6 Verifikace a validace 57
5.7 Vizualizace 58

6 Analýza jazyka Haskell z pohledu konceptuálního modelování **59**
6.1 Vývoj a vlastnosti jazyka Haskell 59
6.2 Běžné konstrukce jazyka Haskell 60
6.3 Rozšíření a knihovny jazyka Haskell 65
6.4 Frameworky pro testování 68
6.5 Zhodnocení expresivity a použitelnosti 70

7 Systém pro kódování konceptuálních modelů **71**
7.1 Reprezentace konceptuálních modelů 71
7.2 Metamodel 77
7.3 Verifikace a validace 78
7.4 Vizualizace 82
7.5 Validace modelu se zákazníkem 85
7.6 Distribuce systému 86

8 Případová studie **87**
8.1 Popis problémové domény 87
8.2 Příprava modelování 88
8.3 Zakódování do systému 89
8.4 Verifikace 94
8.5 Validace 95
8.6 Vizualizace 97
8.7 Zakódovaný konceptuální model 99

9 Zhodnocení výsledného systému **101**
9.1 Vlastnosti prototypu systému 101
9.2 Ekonomicko-manažerské přínosy 104
9.3 Další rozvoj systému 107

Závěr **109**

Literatura **111**

A Seznam použitých zkratek **119**

B Obsah přiloženého CD **123**

Seznam obrázků

1.1 Ullmannův trojúhelník 9
1.2 Cyklus ASIS a TOBE 10
1.3 Úrovně modelování 11
1.4 Verifikace a validace 11
1.5 Validace - Vennův diagram 12
1.6 Jednoduchá sémantická síť 14
1.7 Úrovně ontologií 17

2.1 Příklad EER modelu zápisu studenta 22
2.2 ORM model vítězství ceny v soutěži 23
2.3 Příklad UML diagramu tříd 24
2.4 OntoUML model s Relator vzorem 25
2.5 SysML diagram klimatizace 26
2.6 Příklad specifikace v notaci Z 31

3.1 Alloy Analyzer vizualizace 33
3.2 Overture IDE 34
3.3 Enterprise Architect 35
3.4 Community Z Tools 37
3.5 Menthor Editor 38
3.6 UseOCL 39

4.1 Souvislost MBE, MDE, MDD a MDA 49
4.2 MDA modely 50

6.1 Typové třídy číselných typů 61

7.1 Typové třídy prototypu systému 72
7.2 Metamodel prototypu systému 78
7.3 Ukázka vizualizace modelu 83
7.4 Ukázka vizualizace instance modelu 84

7.5 Diagram procesu validace modelu 85

8.1 Vizualizace zakódovaného modelu 97
8.2 Vizualizace instance modelu 98
8.3 Vizualizace generovaných instancí 98

9.1 Projektový trojimperativ 104

Úvod

Konceptuální modelování jako činnost tvorby modelů zachycujících koncepty a vztahy mezi nimi lze využít v různých odvětvích i mimo tvorbu informačních systémů. Konceptuální modely slouží k porozumění dané části reality, a to ve vhodně zvolené míře abstrakce a rozsahu. Zdali je zachycení reality ve formě takového modelu správné a skutečně odpovídá realitě lze ověřovat procesem validace. Neméně důležitý proces verifikace naopak slouží k ověření, jestli operační model (tj. simulační program, prototyp nebo i již samotný informační systém) pracuje správně a v souladu s konceptuálním modelem.

V určité části vývoje informačního systému je nutné přejít od konceptuálního modelu k implementaci. Takový přechod může být řešen lidskou silou, kdy vývojáři dle modelu programují jednotlivé části informačního systému či jen jednoduché aplikace. To může přivodit řadu chyb a také časté opakování podobných kroků. Modelem řízený vývoj má za cíl automatizovaně přímo z modelu vytvářet zdrojový kód, po následných úpravách kódu však mohou opět vznikat chyby a nemusí být zachována konzistence modelu a kódu. Mezi úrovní konceptuálního modelu a implementace se tak nachází pomyslná problematická mezera, jejíž správné překlenutí může mít za následek výrazné zefektivnění vývoje informačních systémů.

Využití samotného programovacího jazyka, jako alternativy k jazykům pro formální specifikaci, pro tvorbu konceptuálních modelů, jejich simulaci, verifikaci a validaci se jeví jako zajímavé řešení zmíněného problému. Čistě funkcionální programovací jazyk Haskell se těší čím dál tím větší oblibě i za hranicemi akademické sféry. Jeho typový systém, nestriktní vyhodnocování, možnosti zjišťování chyb při překladu, přehledná syntaxe a další vlastnosti z něj dělají kandidáta na vhodný prostředek pro reprezentaci konceptuálních modelů.

Cíle práce

Cílem této práce je seznámit se s problematikou konceptuálního modelování, prozkoumat současné významné metody i nástroje pro tvorbu, vizualizaci, verifikaci a validaci modelů zapsaných pomocí těchto metod. Na základě získaných znalostí, analýzy vybraných vlastností programovacích jazyků a současných řešení propojení konceptuální roviny s implementací poté identifikovat obecné požadavky na zápis konceptuálních modelů a podle nich navrhnout systém a metodiku pro tvorbu konceptuálních modelů v programovacím jazyce Haskell.

Navržená metodika současně stanoví postupy verifikace a validace modelů. Záměrem je tudíž také stanovit postupy pro simulaci či využití zakódovaného modelu v rámci spustitelného programu. Pochopitelně se bude jednat o textovou reprezentaci modelu, pro vizuální kontrolu i použití modelů ke konzultacím je však často vhodnější grafická forma. Dalším cílem je umožnit modely z jazyka Haskell v rámci požadavku vizualizace převést do grafické podoby.

Nakonec je cílem vypracovat případovou studii s prototypem systému pro kódování konceptuálních doménových modelů v jazyce Haskell, zhodnotit toto navržené řešení z pohledu stanovených požadavků i jeho ekonomicko-manažerských přínosů. Účelem vyhodnocení je také navrhnout další možnosti rozvoje a výzkumu v této oblasti, a to jak v obecné rovině využití programovacích jazyků pro modelování, tak i rozšiřování popsané metodiky a vytvořených nástrojů.

Metodika a struktura práce

Práce je strukturována podle svých cílů a metodického postupu spojeného s vývojem SW. Teoretická část obsahuje nejprve úvod do konceptuálního modelování, kde jsou na základě odborné literatury rozebrány důležité pojmy a souvislosti, stručná historie specifikace konceptuálních modelů, výhody využití konceptuálního modelování a návaznosti na další oblasti lidské činnosti. Následuje rešerše vybraných textových a grafických metod pro reprezentaci konceptuálních modelů. V další kapitole se pak nachází přehled nástrojů podporujících tyto metody a je stručně zhodnocena jejich použitelnost. Informace o jednotlivých metodách a nástrojích jsou čerpány z literatury i na základě praktického vyzkoušení základních postupů demonstrovaných v práci na krátkých příkladech.

Praktická část práce, která staví na znalostech načerpaných v předchozí teoretické části, začíná ve formě zhodnocení obecných vlastností programovacích jazyků a jejich možném vlivu na reprezentaci doménových konceptuálních modelů. Na základě teoretické části jsou zanalyzovány základní požadavky na reprezentaci konceptuálních modelů, které slouží jako podklad pro tvorbu vlastního systému pro kódování konceptuálních doménových modelů.

Dále jsou zanalyzovány vlastnosti jazyka Haskell včetně expresivity jako platformy pro naplnění stanovených požadavků. V souladu s analýzou je navržen samotný systém, jehož prototyp je současně implementován a návrh slouží přímo jako jeho popis.

Vytvořený prototyp systému společně s navrženými postupy užití je demonstrován na případové studii komplexního doménového modelu včetně vizualizace, verifikace a validace. V závěru je provedeno zhodnocení výsledků práce společně s ekonomicko-manažerskými přínosy výsledného systému a je navržen další postup vývoje i možností výzkumu v této oblasti.

Část I

Teoretická část

Kapitola 1

Konceptuální modelování

Tato kapitola slouží jako úvod do problematiky konceptuálního modelování, vysvětluje základní pojmy a souvislosti, zdůvodňuje důležitost konceptuálního modelování nejen ve světě softwarového inženýrství, stručně popisuje historický vývoj a samotné přístupy k modelování. Z informací uvedených v této kapitole se postupně dále v práci odvíjí vlastní řešení podpory konceptuálního modelování pro jazyk Haskell.

1.1 Úvod do konceptuálního modelování

Konceptuální modelování je činnost, během níž se popisují vybrané aspekty reálného (fyzického i sociálního) světa kolem nás za účelem jejich pochopení a komunikace o nich [1]. Výsledný popis vybraného výseku reality, často označovaného jako problémová doména nebo jen doména, je bez ohledu na jeho konkrétní reprezentaci určený k použití primárně lidmi [1, 2]. Jak název napovídá, konceptuální modelování staví na konceptualizaci, tedy souboru konceptů a jejich vztahů umožňující zjednodušující pohled na vybranou problémovou doménu. Právě použití konceptů jako výsledku abstrakce a generalizace lidem usnadňuje a umožňuje cestu k poznání a pochopení [3, 4].

1.1.1 Konceptualizace a koncepty

Koncepty se definují jako zástupné myšlenky abstrahující věci reálného světa [5]. Abstrakce umožňuje nejen popis libovolného výseku reality, ale i libovolnou volbu míry detailu tohoto popisu [3, 6]. Popisovaný svět lze namodelovat v rovině zcela obecných pojmů jako je *entita*, ale na druhé straně je možné popisovat problémovou doménu do nejjemnějších detailů, až do popisu kvantového stavu. Jistě ani jedna z uvedených extrémních variant není ve většině případů ideální a demonstrují, že mít cit pro použití správné úrovně abstrakce je při modelování zásadní [7, 8].

Pokud se konceptualizace řídí přesnými pravidly, pak se jedná o formální konceptualizaci, jejíž výsledkem je formalizovaný konceptuální model. Oproti neformální konceptualizaci má výhody především právě v přesně specifikovaném vyjadřování, což výrazně usnadňuje komunikaci nad modelem. Formalizovaný konceptuální model je také možné snadněji validovat, simulovat, transformovat a udržovat ho konzistentní. [1, 3, 9]

Konceptualizace je pro lidi, ale i další živé tvory, zcela přirozená a velmi důležitá. Podle Piagetovy teorie kognitivního vývoje pracujeme s koncepty již v předoperačním stádiu, tj. od staří 2 let [10]. Psychologické experimenty však ukázaly, že používáme generalizaci při poznávání okolního světa dokonce ještě dříve. Známý experiment *Little Albert* [11], během kterého se subjekt po vyvolání strachu začal bát nejen krysy, ale také jím podobným zvířatům a předmětům, ukázal, že již devítiměsíční dítě je schopné použít generalizaci. Schopnosti abstrakce a generalizace jsou očividně rozhodující pro přežití.

1.1.1.1 Aristotelské koncepty

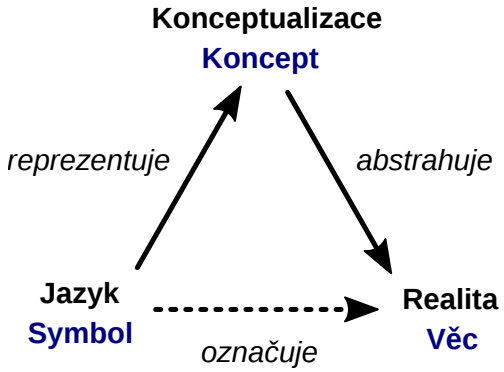
Svůj pohled na koncepty přinesl již Aristoteles, podle kterého je fenoménem (předmětem vnímání člověka) věc existující v realitě, ale také pouze v mysli. Koncept je potom generalizovaná myšlenka o souboru fenoménů založená na znalosti jejich společných vlastností. Na koncept se váží tři pojmy *extenze*, *intenze* a *designace*. Extenze konceptu popisuje kolekci fenoménů, které koncept zahrnuje, naopak intenze kolekci vlastností, které jsou pro tyto fenomény společné v rámci konceptu. Designace neboli označení je kolekce názvů, pod kterými je koncept znám. [12]

Příkladem tohoto Aristotelského přístupu ke konceptům je koncept *Univerzita*, do jehož extenze patří například *ČVUT v Praze*, *Západočeská univerzita* nebo *Stanford University*. Intenze zahrnuje vlastnosti jako *oficiální název*, *výuka studentů*, *vědecká činnost* a další. Koncept je znám i pod dalšími označeními *Vysoká škola* nebo *Vysoké učení*.

1.1.1.2 Reprezentace konceptů

Koncepty jako myšlenky je nutné při konceptuálním modelování nějakým způsobem reprezentovat, aby bylo možné o nich komunikovat [3]. Touto problematikou se zabývá sémiotika neboli nauka o znakových systémech, zahrnující sémantiku, syntaxi a pragmatiku [13]. Libovolný symbol označuje věc reálného světa tím, že reprezentuje koncept abstrahující tuto věc [14]. Popsaný vztah znázorňuje Ullmannův trojúhelník vycházející ze sémiotického trojúhelníku, někdy také známého jako trojúhelník reference [3].

Soustava symbolů tvoří společně s pravidly jejich užití jazyk, ten může být grafický, textový nebo smíšený, právě podle použitých symbolů. Volbu jazyka je nutné zvolit vzhledem k účelu, za jakým je konceptuální model tvořen [3]. V některých případech existují nástroje pro automatickou konverze modelu



Obrázek 1.1: Ullmannův trojúhelník (podle [14])

mezi jazyky (například z BPMN do UML diagramu aktivity [15] nebo z OntoUML do Alloy [16]). Tento převod však může být ztrátový buď důsledkem toho, že cílový jazyk má nižší úroveň expresivity než jazyk původní nebo že nástroj neumožňuje využití všech konstruktů zdrojového nebo cílového jazyka.

1.1.2 Využití v SW inženýrství

Konceptuální modelování lze uplatnit napříč všemi obory lidské činnosti. Využití najde všude, kde je potřeba důkladně pochopit problémovou doménu. Jedním z typických užití konceptuálního modelování je analýza fungování organizací bez ohledu na konkrétní zaměření za účelem optimalizace jejich struktury či procesů [3]. Podobným případem je využití v softwarovém inženýrství při tvorbě informačních systémů podporujících činnosti organizace [17].

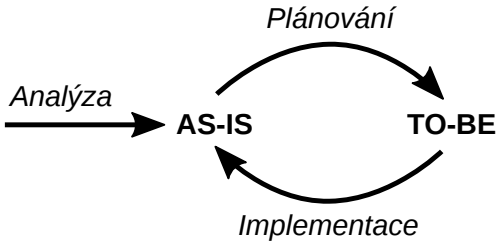
Vývoj informačního systému probíhá dle zvolené metodiky v určitých fázích. Ať už se jedná o klasickou vodopádovou posloupnost analýza, návrh, implementace a údržba nebo o agilní iterační či inkrementální metodiku, je každopádně nejprve nutné vědět, co a proč se dělá. Dobrá znalost problémové domény je klíčová [17].

1.1.3 Čas modelování

Z pohledu času je možné vytvářet dva typy modelů. Prvním je deskriptivní model, častěji označovaný jako *as-is*, který popisuje současný stav dané problémové domény. Jeho účelem je pochopit, jak v daném čase funguje problémová doména, odhalit její silné i slabé stránky a nalézt případná řešení problémů právě skrze pochopení všech souvislostí a potřeb. [8, 18]

Druhým typem je preskriptivní *to-be* model popisující, jaká bude problémová doména po zavedení nějakých změn [18]. Pod těmito změnami si lze představit různé optimalizace procesů, restrukturalizace anebo zavedení informačního systému. Požadovaným stavem je, aby se *as-is* model v budoucnosti

shodoval se současným *to-be* modelem. V takovém případě je logicky nutné, aby byl *to-be* model realizovatelný.



Obrázek 1.2: Proces modelů *as-is* a *to-be*

Poslední možností typu modelu z časového hlediska je model minulosti, ale ten se takto zpětně vytváří zřídka, neboť modelování by mohlo být problematické a nepřesné. Reálné využití najde například při studiu domény z pohledu historie. Naopak uchovávání minulých *as-is* modelů může být zajímavé z hlediska pozorování vývoje modelované domény [19]. Ideální cyklus *as-is* a *to-be* modelů je provázán s Demingovým cyklem kontinuálního zlepšování PDCA:

P = plánování (vytvořit *to-be* model),

D = realizace (implementace *to-be*),

C = ověření (zjištění, zda plánovaný *to-be* je nyní *as-is*),

A = analýza přínosů, přijetí nebo zamítnutí [20].

1.1.4 Úrovně modelování

Konceptuální model popisuje vybrané aspekty světa, ale samotný konceptuální model je také součástí světa. Lze tedy vytvořit model konceptuálního modelu, lépe řečeno metamodel [8]. Součástí světa je však i metamodel, je tudíž možné vytvořit metametamodel. Na úrovni metametamodelů (M3) je již míra abstrakce taková, že další úrovně nejsou třeba a metametamodel je modelem sama sebe, případně je natolik jednoduchý, že jej není potřeba dále modelovat [18]. Jsou definovány úrovně:

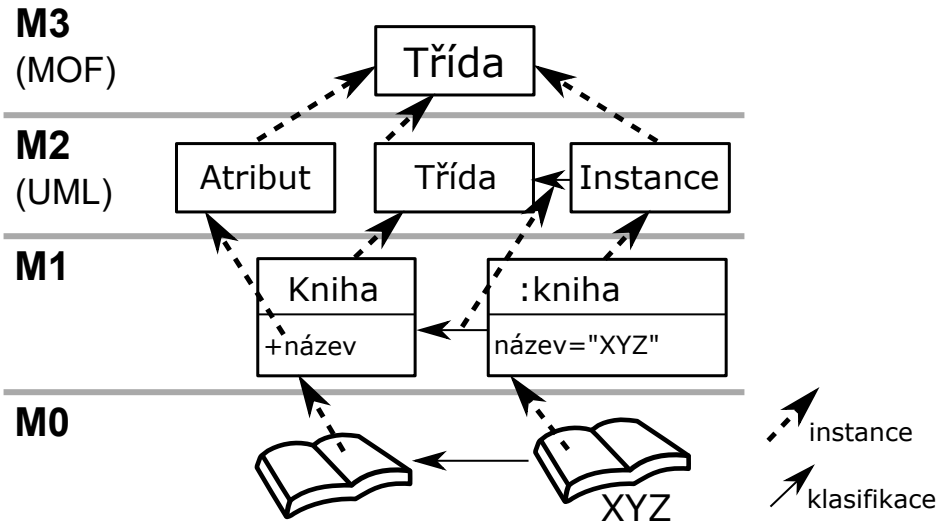
M0: systémy reálného světa,

M1: doménové modely,

M2: metamodely (DSL),

M3: metametamodely [21].

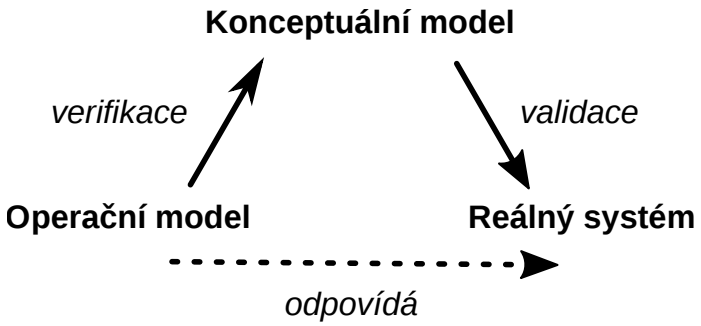
Vývoj nové metodiky tvorby konceptuálních modelů je ekvivalentní k vytvoření metamodelu případně i příslušného metametamodelu [18]. Za demonstrativní příklad úrovní modelování si lze vzít z reálného světa knihu, její UML model, jazyk UML a Meta-Object Facility (MOF), jak je znázorněno na obrázku 1.3. Entity v nižší vrstvě jsou instancemi entit(y) ve vyšší vrstvě [21].



Obrázek 1.3: Úrovně modelování a metamodelování (podle [21])

1.2 Verifikace a validace

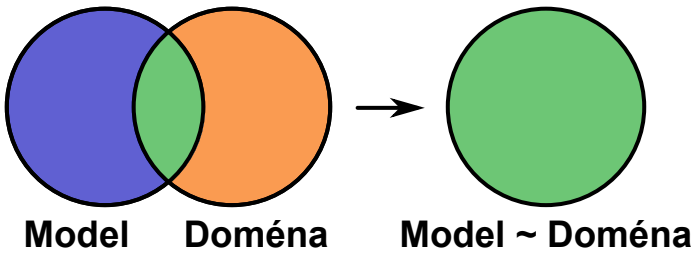
Každý konceptuální model popisuje množinu instancí světů, která jej splňuje [6]. Například jednoduchý model, kde kniha má autora a autor má jednu nebo více knih. Instancí modelu je svět, kde neexistuje žádný autor ani kniha, kde existuje autor s deseti knihami nebo tři autoři s jednou knihou a spočetně nekonečně mnoho dalších.



Obrázek 1.4: Verifikace a validace (podle [22])

Ověřování, zda je vytvořený konceptuální model ve shodě s popisovaným výsekem reality, se nazývá validace [23, 24]. Postup může být různý dle zvolené metodiky. Častým způsobem validace je generování instancí podle modelu, jejich manuální zkoumání a následné zamítnutí nebo přijetí. Při takovémto generování je vhodné nejprve generovat v jistém smyslu mezní instance. Validaci lze provádět ovšem také opačně tak, že se specifikují validní a případně i nevalidní instance, a poté se zjišťuje, zda patří do množiny instancí generovaných modelem či nikoliv [6].

Na druhé straně je verifikace, tedy ověření, že operační model je ve shodě s konceptuálním modelem [22]. Může se jednat přímo o implementovaný informační systém nebo jen simulační aplikaci, vytvořenou za účelem validace. Takové ověření může být automatizované, stačí umět generovat instance světů, které model popisuje i které již nepopisuje a ověřit, jak s nimi implementace pracuje [24].



Obrázek 1.5: Vennův diagram korespondence modelu a domény (podle [24])

Požadovaným výsledkem konceptuálního modelování a následné implementace je, aby popsaný svět v modelu lícoval se světem popisovaným a stejně tak, aby implementace lícovala s modelem, jak znázorňuje Vennův diagram 1.5. Modrá část reprezentuje instance, které model připouští a neměl by, protože nejsou reálné v doméně. Naopak oranžová část značí instance, které model nepřipouští a měl by, protože v doméně existují. Zelená část obsahuje instance, které model správně připouští, protože jsou reálné [23, 25].

1.3 Výhody a možné problémy

Pochopení problémové domény při vývoji informačního systému i při hledání jiných optimalizací je klíčové. Přes některé mylné argumenty ospravedlňující zanedbávání konceptuálního modelování přináší v softwarovém inženýrstvím řadu výhod a jeho zanedbání může vyústit až v těžko řešitelné problémy [3, 6].

1.3.1 Součást dokumentace

Konceptuální model je důležitou součástí dokumentace [7, 26]. Jednak sjednocuje pohled na problémovou doménu všech zúčastněných osob (tzv. *stakeholders*), ale dále se na jeho základě mohou specifikovat funkční požadavky

či závazně definovat pojmy a vztahy v problémové doméně [27, 28]. Vhodné je namodelovat *to-be* stav a přiložit tento model do dokumentace, kde slouží pro ukázání vlivu na fungování organizace i k ověření, že smluvené požadavky byly splněny a informační systém je z tohoto pohledu hotový [26].

Pro dokumentační účely jsou přínosné především grafické reprezentace modelů srozumitelné všem čtenářům, případně se stručným vysvětlením [29]. Pokud je nutné ověřit, že model opravdu koresponduje se skutečností, pak pro validaci je zpravidla nutné použít i mode ve formě textové formální specifikace (například v jazyce Alloy) [6].

1.3.2 Úspory prostředků

Jedním z argumentů, proč se konceptuálním modelováním nezabývat nebo se mu věnovat jen povrchně, jsou úspory finančních prostředků. Taková úspora spočívá především v tom, že nejsou potřeba specializovaní analytici a stačí provést sběr požadavků na jejichž základě se přímo navrhne řešením, které se implementuje s možným využitím dříve realizovaných projektů. Problémy s opomenutými požadavky se řeší později ad-hoc nebo až v rámci změnového řízení, ale právě tím se úspora v úvodní fázi může vyrovnat nebo dokonce dojít ke zbytečné ztrátě. Stejně tak bez řádné analýzy může být implementována zbytečná funkcionalita navíc, což opět vyústí ve zbytečně vynaložené prostředky. [6, 17, 26]

Při vhodně zvolené granularitě a detailu konceptuálního modelu dojde k dostatečnému porozumění problémové doméně a zároveň celkové nižší časové náročnosti projektu. Tento postup je použitelný i pro iterativní (agilní) metodiky, kde se postupně konceptuální model doplňuje a upravuje [30].

1.3.3 Kvalita výsledku

Pokud výsledek odpovídá naplánovanému *to-be* stavu a příslušnému konceptuálnímu modelu, pak je informační systém dodán v plánované kvalitě a zaručeně podporuje činnost organizace [26]. Díky konceptuálnímu modelování může být odhaleno, že je možné použití již existujícího „krabicového“ řešení nebo naopak že je nutné vytvořit řešení na míru z důvodu specifických činností, postupů či struktury v organizaci [7]. V obou případech konceptuální modelování zajistí, že výsledek je pro organizaci vhodný a zcela použitelný, ideálně umí přesně to, co je požadováno, tzn. nic nechybí a nic zbytečně nepřebývá [6, 17].

1.3.4 Ochrana majetku a života

Ověření, že informační systém je v souladu s požadavky, může být provedeno automatizovaně pomocí verifikace proti validovanému konceptuálnímu modelu [7, 22]. Důraz na tento soulad s realitou skrze konceptuální model je kladen

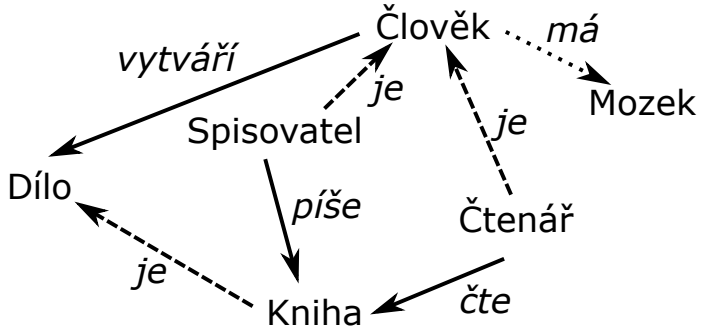
především u informačních systémů, jejich součástí a aplikací, kde chyba může mít fatální následky ve formě ztráty lidského života nebo velkého množství aktiv [6, 17]. Příkladem těchto kritických systémů jsou:

- systémy řízení provozu dopravy (letadla, vlaky, dopravní semafory, ...),
- vojenské systémy (rakety, bezpilotní letouny, ...),
- bankovní systémy,
- zabezpečovací systémy,
- systémy pro řízení rozvodných sítí (elektřina, voda, plyn, ...),
- systémy spravující přísně tajné informace,
- systémy ve zdravotnictví řídící podporu života nebo používané při operacích (například robotické operace).

1.4 Historický vývoj

1.4.1 Sémantické sítě

Za první milník v užití formálního konceptuálního modelování v informatice se udává rok 1966 a sémantické sítě, které znázorňují jak koncepty, tak i jejich vztahy [31]. Sémantické sítě měly být modelem struktury lidské paměti [8]. Jedná se o orientovaný graf, jehož vrcholy tvoří koncepty a hrany značí různé vztahy mezi nimi. Relace *je* (*is-a*) a *má* (*has*) se zpravidla reprezentují odlišně pro jejich specifičnost [8, 31].



Obrázek 1.6: Příklad jednoduché sémantická sítě

1.4.2 Simula 67

V roce 1967 byl uveden první funkční objektově orientovaný programovací jazyk Simula 67. Jak může název napovídat, jeho záměrem bylo provádění simulací a třídy měly sloužit k abstrakci, jako definice vlastností společných pro instance a k vytvoření hierarchie dědičnosti [31]. Jednotlivé třídy se začaly zakreslovat způsobem podobným pozdějšímu ER a UML.

1.4.3 ER a EER modely

V roce 1976 doktor Peter Pin-Shan Chen představil ve svém článku *The Entity-Relationship Model: Toward a Unified View of Data* [32] ER model. Na ER modelech lze pozorovat podobnost s sémantickými sítěmi [8]. ER model zpřehledňuje pohled na koncepty, jejich vlastnosti a vztahy [27]. Obdélníky znázorňují koncepty, jejichž atributy jsou vepsány do elipsy spojené s obdélníkem čarou. V Chenově notaci jsou relace značeny kosočtvercem a čáry spojují tento kosočtverec s koncepty, které jsou v relaci. U těchto čar se uvádí kardinalita, a to blíže u kosočtverce. [32]

Postupem času vznikly různé alternativní konvence, kde se především relace zakreslují jednodušeji [31]. Přidáním speciálního zachycení generalizace/specializace vznikl Enhanced Entity-Relationship Model, zkráceně EER nebo také E^2R. ER modely se v novější podobě stále používají pro zachycení například databázového relačního schéma [27].

1.4.4 UML, OCL a OntoUML

Na počátku 90. let 20. století již díky rozvoji objektově orientovaného programování existovalo více než 50 různých modelovacích metod. To bylo důvodem vzniku nejznámějšího modelovacího jazyka UML, který byl spojením principů několika původních metod a jazyků. Verze UML postupně zdokonalovaly jeho expresivitu a schopnosti popsat jak strukturu, tak i chování. [8, 27]

V UML chyběla možnost specifikace různých pravidel, integritních či jiných omezení. Pro tento účel byl vytvořen deklarativní jazyk Object Constraint Language (OCL). Výhoda oproti vlastním textovým poznámkám u modelu je, že s OCL lze automatizovaně provádět kontroly a případně pravidla přeložit do jiného (programovacího) jazyka [33].

I přes postupný vývoj UML se jedná o modelovací jazyk především určený pro implementačně závislé modely [34]. Proto vznikla potřeba vytvářet rozšíření podporující práci s koncepty. Významným rozšířením je OntoUML postaveným na ontologii UFO, jejíž autorem je Giancarlo Guizzardi [3]. Jak název napovídá přináší do UML prvky z ontologie v podobě přidaných stereotypů a jejich vlastností [35].

1.4.5 Jazyky formální specifikace

Mimo grafických notací samozřejmě vznikaly postupem času i textové notace. Ty přinášejí výhodu především v automatických kontrolách správnosti zápisu i prvky validace a verifikace modelů [17]. S první významnou formální metodu přišli v 70. letech 19. století výzkumníci z IBM laboratoří ve Vídni, a proto se jmenuje Vienna Development Method (zkráceně VDM). Mimo specifikačního jazyka VDM-SL a později VDM++ přinesla i nástroje a definované postupy [36]. V 70. letech byla uvedena i notace Z jako „dokonalý jazyk“ založený na matematických symbolech [37].

Na Z byla později postavena metoda B, která byla použita například při návrhu systémy pro pařížské metro nebo raketu Ariane 5 [38]. Před přelomem tisíciletí tým Software Design Group z MIT představil první verzi jazyka Alloy silně ovlivněného notací Z a OCL. Původně uměl pouze modelovat objekty, ale postupně se zdokonaloval a vznikl nástroj Alloy Analyzer pro validace [6].

1.4.6 Procesní modelování

Konceptuální modelování není pouze o modelování struktury, ale také chování [7]. Ačkoliv zde existovaly stavové automaty, Petriho sítě a později i diagramy chování (sekvenční, aktivit, komunikační, ...) v UML, pro modelování složitějších procesů v problémových doménách nebyly příliš vhodné [33]. Za účelem modelování ekonomických procesů vzniklo BPMN. Podobně jako u UML se postupem času notace rozšiřovala o nové prvky a také je společně s UML spravováno Object Management Group. Jednou z výhod je možnost transformace do BPEL a orchestrace procesu [39].

Již před uvedením BPMN profesor Jan Dietz představil metodiku DEMO založenou na transakčním vzoru a třech úrovních organizací – datalogické, infologické a ontologické. V modelech DEMO lze zachytit jak strukturu organizace, tak i návaznosti jednotlivých akcí [40].

1.5 Souvislost s ontologií a programováním

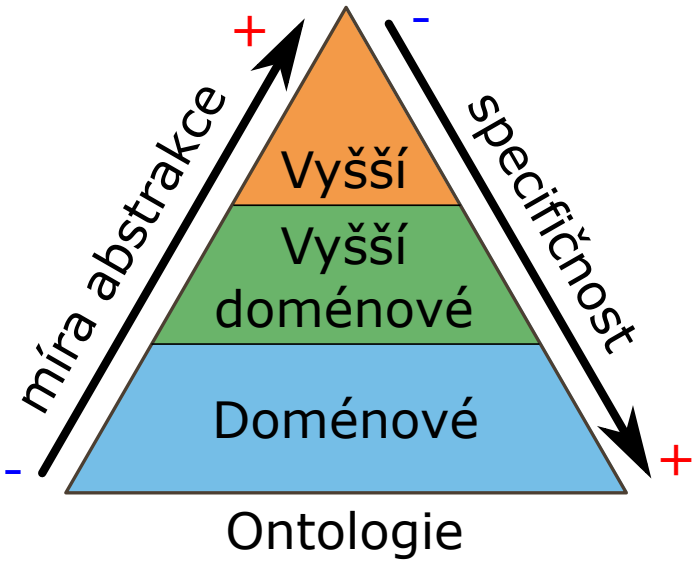
1.5.1 Ontologie

Ontologie je filozofická disciplína zabývající se jsoucnem, existencí, realitou, kategoriemi bytí a jejich vztahy. V informace se pod pojmem ontologie rozumí formální vymezení pojmů včetně definice jejich typů, vlastností a vztahů v rámci vybrané domény [4]. Ontologie obsahuje *glosář* neboli definici pojmů a *thesaurus*, tedy vztahy definovaných pojmů [3]. Z této definice vyplývá zřejmá souvislost s konceptuálním modelováním, které má za cíl popis problémové domény a k tomu je nutné znát pojmy a vztahy v této doméně [8].

Ontologie se dělí na nižší (doménové), vyšší (základní) a hybridní. Doménové ontologie jsou v jistém smyslu formou a součástí konceptuálního modelu,

neboť právě ty popisují nějakou problémovou doménu. Vyšší ontologie definují základní obecné pojmy a vztahy, které lze poté používat v nižších ontologiích [3, 4]. Mezi vyšší ontologie patří UFO, BFO, GFO, DOLCE, SUMO a další [3, 41]. Podobně jako modely staví na pojmech a vztazích definovaných v metamodelu, tak vyšší ontologie charakterizují pojmy a jejich vztahy, s nimiž dále pracují nižší ontologie [3].

Mezi vrstvy doménových a vyšších ontologií lze ještě zařadit ontologie specializované na určitou doménu ale stále ještě v obecné rovině, například na medicínu, elektrotechniku, zemědělství, tyto jsou označovány jako ontologie střední vrstvy nebo vyšší doménové. Jednou z takových ontologií je například BioTop pro přírodní vědy [42].



Obrázek 1.7: Úrovně ontologií (dle [42])

Využití ontologií jde napříč obory lidské činnosti a zájmů, uplatnění nachází ve zdravotnictví, přírodních vědách, výuce, historii, geografii a dalších. Konkrétně v informatice se ontologie objevují mimo jiné v oblastech big data, sémantickém webu (OWL a RDF) a právě konceptuálním modelování jako součásti softwarového inženýrství [3].

1.5.2 Programování a KM

Konceptuální modelování je výpočetně i platformě nezávislé. Příjemnou možností některých modelovacích nástrojů je vygenerování alespoň implementačně závislého modelu nebo přímo kostry implementace z konceptuálního modelu [28]. Úskalím těchto generování je, že vygenerovaný model či kód je nutné upravit a změny se v konceptuálním modelu již nepropojí. Při následné změně konceptuálního modelu se musí přegenerovat i kód a mohou s provedenými

změnami vzniknout kolize nebo jiné problémy. Udržování konceptuálního modelu vždy ve shodě s implementací je náročné, o průlom v této oblasti se snaží například teorie normalizovaných systémů [43].

Jelikož se jedná o modelování světa, kterému je svým principem bližší objektově orientované programování a lze to pozorovat kupříkladu i na podobnosti grafických konceptuálních modelů struktury a modelu tříd. Bohužel tím často dochází k degeneraci konceptuálního modelování na pouhou tvorbu modelu tříd bez vlastního důkladného porozumění problémové doméně [44].

U některých jazyků formální specifikace lze pozorovat výraznou podobnost s programovacími jazyky. Vyvstává tak otázka, proč se nepoužívají přímo programovací jazyky a zavádí se tyto specializované jazyky pouze pro modelování. Většinou jsou jazyky formální specifikace výrazně jednodušší a výrazově přesnější pro účely modelování. V programovacím jazyku je možné konceptuální model zapsat také, ale většinou buď složitěji nebo ne tak detailně. [8, 17]

1.5.3 Doménově specifické jazyky

Doménově specifické jazyky (DSL) jsou specializované pro určitou doménu na rozdíl například od běžných programovacích jazyků určených k obecnému řešení problémů z libovolných domén [45]. Stejně jako všechny jazyky jsou i DSL určené ke komunikaci. Výhodou těchto jazyků je, že lze snadno a přímo zachytit koncepty z problémové domény a díky tomu jim rozumí i doménoví experti, kteří však neumí programovat.

Vytvářet nové DSL je možné jako nadstavby programovacích jazyků, v takovém případě se jedná o interní DSL [45]. Popis napsaný v DSL pak může být snadno použit jako součást celého informačního systému. Výsledný kód je přehlednější a lépe udržitelný. Moderní programovací jazyky jako Ruby, Groovy, Scala a další prezentují dobrou podporu DSL jako jednu ze svých významných předností [46].

Naopak DSL s vlastní syntaxí, které se do programovacího jazyka přeloží nebo se interpretuje se nazývá externí DSL. Logicky však existují i nevýhody, a to především ve formě nákladů, které vzniknou s potřebou naučit se s ním pracovat, udržovat ho a případně jej i vytvořit [45].

1.5.4 Programování konceptuálních modelů

Zajímavou myšlenku přináší *Conceptual Model Programming: A Manifesto* [47], která staví na tom, že konceptuální model je kód. Tím se odstraní nutnost verifikace, neboť výsledná aplikace vždy odpovídá modelu, protože je z něj přímo kompilovaný. Na konceptuální model jsou kladeny dva požadavky:

1. musí být kompletní a holistický (musí popisovat strukturu i chování) a
2. musí být přesný (vše v modelu musí být pečlivě a přesně definováno).

Rozšiřování a úpravy aplikace se pak dějí přímo nad modelem a nikoliv nad samotným kódem. To však vyžaduje, aby model obsahoval opravdu vše a to včetně popisu uživatelského rozhraní. CMP vychází v mnohem z myšlenek Model Driven Architecture (MDA), která se také snaží o zmenšení pomyslné vzdálenost konceptuálního modelu a aplikace [47].

Kapitola 2

Metody zápisu konceptuálních modelů

Konceptuální model je možné zapsat pomocí různých metod, každá z nich má své charakteristické vlastnosti, výhody a nevýhody. V této kapitole jsou popsány nejznámější metody používané pro tvorbu konceptuálních doménových modelů, ale i některé méně popularizované metody formální specifikace. Obě skupiny je vhodné prozkoumat, jelikož z nich lze čerpat cenné informace pro vývoj vlastní metody s použitím jazyka Haskell.

2.1 Reprezentace konceptuálních modelů

Aby bylo možné nad konceptuálním modelem komunikovat, je nutné jej reprezentovat pomocí symbolů [3, 8]. Symboly tvoří jazyk a ten může být buď textový, grafický nebo smíšený [29]. Expresivita zvoleného jazyka udává, co vše je možné za pomocí jeho symbolů zachytit. Při modelování musí být zvolen jazyk nebo i kombinace jazyků (například UML společně s OCL) tak, aby byly všechny požadované detaily modelu vyjádřitelné [26].

Při modelování je nutné také vzít v potaz nástrojovou podporu jazyka. Pokud je zvolen jazyk s dobrou expresivitou, ale nízkou podporou ze strany nástrojů a komunity, může rychle dojít k potížím model udržovat a komunikovat nad ním s jinými osobami. Expresivita je jenom jedním z hledisek, které by měly být zváženy při výběru vhodného způsobu pro reprezentaci konceptuálního modelu. Mezi důležité vlastnosti tedy patří:

- expresivita jazyka,
- nástrojová podpora,
- rozšířenost jazyka,
- rozšiřitelnost a přizpůsobitelnost jazyka.

Pro popis v této kapitole byly vybrány jazyky a metody, které mají dobrou nástrojovou podporu (rozvedeno v další, 3. kapitole) a patří mezi obecně známější. Zároveň jsou dostatečně univerzální a lze jimi popsat libovolný systém.

2.2 Grafická reprezentace

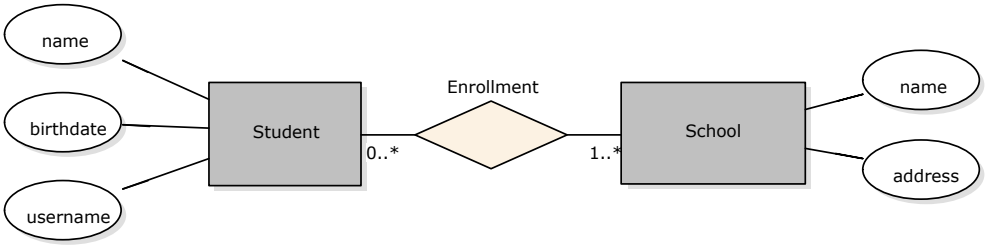
Výzkum z oblasti kognitivních studií říká, že reprezentace pomocí obrázků zefektivňuje proces pochopení a porozumění. Konceptuální model zapsaný diagramem přináší srozumitelnější pohled na model. V grafické reprezentaci se však často opomíjí některé detaily nebo jsou vyjádřeny kombinací různých typů diagramů, příkladem je časté oddělení modelů struktury a chování. Diagramy kombinují grafické symboly společně s textovými popisky. [26, 29]

Ačkoliv se jedná o grafický jazyk, v pozadí nástrojů se však často nachází textový popis modelu (XML, XMI, ...) a díky tomu je možné provádět ověřování správnosti. Některé nástroje umožňují i převod mezi různými jazyky právě na základě tohoto popisu.

2.2.1 ER/EER model

Jak již bylo uvedeno dříve, ER model je jedním z prvních formálních prostředků pro konceptuální modelování entit a jejich vztahů. Hlavním účelem i způsobem použití je modelování skutečností, které je nutné uchovávat pro potřeby nějakých (business) procesů [27].

V ER se zakreslují jednotlivé entity, na které se vážou jejich atributy. Atribut může být navíc klíčový, vícehodnotový, kompozitní nebo odvozený. Klíčové atributy slouží jako identifikátor instance entity, v databázích odpovídají primárním klíčům. Pokud entita nemá žádný vlastní klíčový atribut (má je pouze ze vztahů), pak se označuje jako slabá. [32]



Obrázek 2.1: Jednoduchý EER model zápisu studenta ve škole

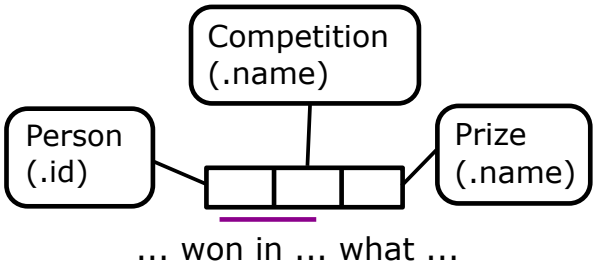
Atributy, které odkazují na jinou entitu, indikují vztahy nebo také relace. Obecně lze v ER namodelovat vztahy libovolného stupně (arity) a zakreslují se pomocí kosočtverce spojeného s participujícími entitami. U relace se u všech spojnic uvádí kardinalita (maximální počet instancí relace, ve kterých může entita participovat). Samotné relace mohou mít své vlastní atributy, dále se mohou podílet na identifikaci entity a mohou být pro entitu povinné. [32]

Jak již bylo zmíněno, EER obohacuje ER o speciální vztahy mezi entitami: generalizace/specializace (včetně vícenásobné), agregace a sjednocení. U relací byla přidána možnost specifikovat multiplicitu pomocí intervalu (minimální a maximální počet instancí relace pro entitu). [27]

2.2.2 Object Role Modeling

Object Role Modeling (ORM) se zabývá konceptuální modelováním za účelem návrhu a dotazováním nad informacemi ve smyslu základních faktů v doméně [48]. Tato metoda se někdy označuje také jako modelování orientované na fakta. S fakty v podobném smyslu pracuje například i zmíněná metodologie DEMO [40].

V notaci ORM lze pozorovat podobnost základních prvků s ER/EER modely. Základem jsou opět entity, typy hodnot, podtypy a vztahy mezi nimi, které jsou modelovány jako predikáty libovolné arity. Pomocí vztahů a entit se formují fakta. Hlavní odlišností ORM je možnost specifikovat celou řadu omezení. Velmi jednoduše se vyjádří požadavek na unikátnost, povinnost vztahu a jejich počet, kombinování vztahů pomocí operací a relačních vlastností, enumerace možných hodnot a další. [48]

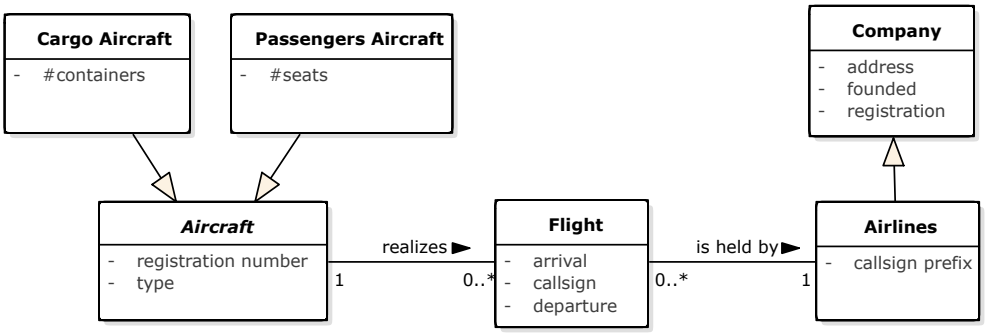


Obrázek 2.2: Jednoduchý ORM model vítězství ceny v soutěži

2.2.3 Unified Modeling Language

Unified Modeling Language (UML) je grafický jazyk pro vizualizaci, specifikaci, konstrukci a dokumentaci systémů. UML nabízí jednotný a standardní prostředek pro zápis, jak návrhu systému včetně konceptuálních prvků jako jsou business procesy a struktura domény, tak konkrétních prvků jako jsou jednotlivé funkce a metody, databázová schémata, komponenty aplikace či fyzické nasazení systému [27]. Standard UML definuje standardizační skupina OMG [49]. Poslední vydanou UML specifikací je verze 2.5, předchozí verze 2.4.1 je ale na rozdíl od 2.5 přijata jako ISO standard ISO/IEC 19505 [50].

UML podporuje objektově orientovaný přístup k analýze, návrhu a popisu programových systémů. UML je pouze jazykem a neobsahuje způsob, jak se má používat, ani metodiku, jak provádět analýzu či specifikaci systémů [27].



Obrázek 2.3: Zjednodušený UML diagram tříd systému evidence letů

Protože je jazyk navržen pro popis přímo architektury a implementace informačních systémů, jsou konceptuální modely v UML z ontologického hlediska příliš vágní, tento problém řeší například OntoUML.

2.2.3.1 Diagramy UML

V UML jsou diagramy rozděleny do tří kategorií. První skupinou diagramů, které UML definuje, jsou strukturální diagramy. V těchto diagramech se znázorňují jednotlivé části a jejich vztahy systému z různých perspektiv a úrovní abstrakce [51]. Již z názvů jednotlivých diagramů je zřejmé, že UML je vhodné především pro objektově orientované programování. Dalšími skupinami diagramů UML jsou diagramy chování a interakce [50].

Často používaný diagram tříd v určitých aspektech vychází z ER modelů, a to je také důvodem, proč se často používá i pro konceptuální modelování struktury. Pro popis chování je nejvhodnější diagram aktivit, který vychází z klasických vývojových diagramů (flow chart) [27]. Pro konceptuální modelování lze použít samozřejmě i další typy diagramů UML, ale jejich použití je ztíženo právě tím, že jsou navrženy pro popis implementace ve formě SW.

2.2.3.2 MDA, xtUML a fUML

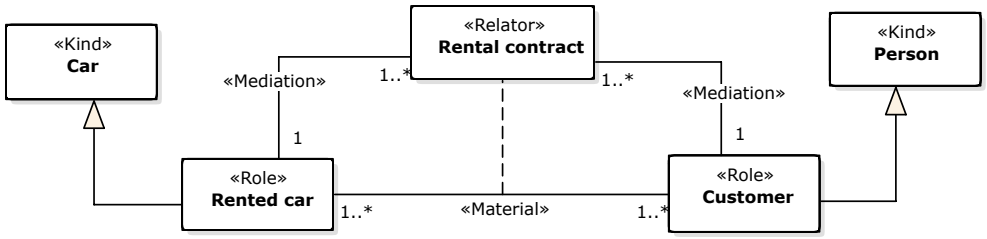
Tím, že primárním účelem UML je popisovat architekturu a implementaci aplikací, existuje řada nástrojů a postupů, jak z modelu v UML generovat kód určitého programovacího jazyka v rámci MDD a MDA spravovaného přímo OMG [28]. Jedním z přístupů k MDA je metoda a současně jazyk xtUML, který obsahuje podmnožinu UML obohacenou o exekuční sémantiku a pravidla. Modely v xtUML tak lze spouštět, testovat a kompilovat do nižších programovacích jazyků. Tato práce je podporována řadou komerčních i nekomerčních nástrojů [52].

Podobným přístupem je i fUML/ALF definované a standardizované OMG. Pomocí fUML, které je opět podmnožinou UML, se definuje struktura a chování systému. V textovém jazyce ALF, který se svou syntaxí a sémantikou

značně podobá jazykům C++ a Java, se definuje chování vzhledem k nějakému UML modelu. Výhodou je právě takto definované mapování mezi fUML a ALF [53].

2.2.4 OntoUML

OntoUML, jak je zřejmé z názvu, přináší do jazyka UML principy ontologie, konkrétně vyšší ontologie UFO vytvořenou Giancarlem Guizzardi a spravovanou včetně jazyka OntoUML a nástroje OLED skupinou NEMO. Rozšíření je řešeno pomocí tzv. stereotypů v UML, které lze používat téměř všude a tím definovat speciální užití konkrétních konstruktů. V případě OntoUML se jedná především o stereotypy tříd a relací. Takto jsou do UML přidány pojmy jako například role, fáze, kvantita, kategorie, kolektiv, člen, mediace a další definované v metamodelu OntoUML a ontologii UFO [3].



Obrázek 2.4: Model v OntoUML za použití Relator vzoru

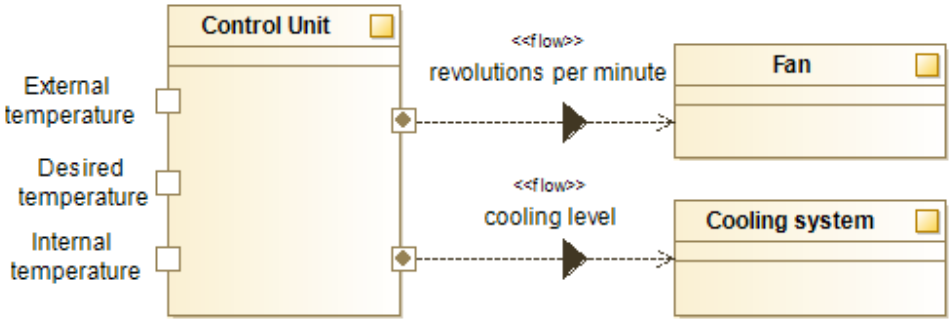
Použití definovaných pojmů z UFO vede k vyšší expresivitě a zanesení různých pravidel do modelů. Například je definováno, co může být podtypem a nadtypem čeho, co musí mít povinně vztah k jiné entitě, co poskytuje identitu i jaké jsou vlastnosti části uvnitř celku. Tím lze model snadno ověřovat a vnikají určité vzory, které se v modelu často vyskytují (patterns), i naopak, které by se v modelu neměly vyskytovat (anti-patterns). S rozvojem UFO se rozvijí i OntoUML, například později byl přidán stereotyp pro modelování událostí [35].

2.2.5 SysML

SysML je profilem UML, množinou vybraných prvků jazyka UML rozšířenou o nové prvky, určeným pro systémové inženýrství. Jazyk je vytvořen tak, aby podporoval analýzu, návrh, verifikaci a validaci různých jednoduchých i kompozitních systémů. Modely se v jazyce zachycují pomocí sedmi standardních UML diagramů, diagramu požadavků a parametrických diagramů. V SysML se mimo struktury a chování modelují i požadavky na systém a parametry včetně jejich vazeb v systému. [51, 54]

Právě přidaný model požadavků přidává propojení vlastností modelovaného systému s požadovanými vlastnostmi, tedy určité podmínky, které je

nutné splnit, aby byl systém požadován za akceptovatelný. Pomocí parametrů a jejich vztahů (rovnic) se poté modelují určité vnitřní vlastnosti systému, ze kterých rovněž vychází omezení (například fyzikální). Těmito podmínkami a omezeními lze pak snadno ověřovat správnost modelu a systému. [54]



Obrázek 2.5: Parametrický diagram SysML pro řízení klimatizace

2.3 Textová reprezentace

Konceptuální model je samozřejmě možné reprezentovat i přirozeným jazykem. V takovém případě se doporučuje dbát na výrazovou přesnost, úplnost, srozumitelnost a použití správných typografických elementů pro popis (seznamy, tabulky, dělení do odstavců a oddílů, ...) [29]. Nevýhodou tohoto přístupu je především délka a nepřehlednost. Automatizované zpracování a ověřování správnosti modelu je značně ztíženo. Z tohoto důvodu se používají specializované formální jazyky nebo jazyk Gellish, který je založen na přirozeném jazyce a jehož věty mají předepsané části [55].

Formální specifikace v oblasti computer science jsou matematicky založené techniky obecně používané pro popis struktury a chování systémů. Jejich výhodou je výrazová přesnost, definovaná syntaxe a sémantika, což výrazně usnadňuje automatizované kontroly modelů [17]. Nevýhodou je nutnost se naučit nový jazyk včetně jeho pravidel a tudíž nelze očekávat, že zákazník bude specifikaci v takovém jazyce rozumět.

2.3.1 Alloy

Alloy (česky slitina) je deklarativní specifikační jazyk pro popis struktur založený na matematické logice prvního řádu, který je včetně nástrojové podpory vyvíjen na MIT. Ačkoliv je podstatně ovlivněn notací Z, jeho syntaxe je bližší OCL. Model v jazyce Alloy tvoří sada omezení, které popisují množinu struktur, seskupených do modulů. Přes poměrně jednoduchou syntaxi jazyka je docíleno vysoké míry expresivity. Navíc díky tomu, že se jedná o deklarativní jazyk, pořadí výrazů nehraje žádnou roli. [6]

Zápis Alloy modelu se sestává z:

- signatur definujících množiny,
- faktů, které představují stálá omezení,
- predikátů, které jsou na rozdíl od faktů parametrickými omezeními,
- funkcí vracející hodnotu pro nějaké zadané parametry a
- tvrzení ohledně modelu, ke kterým mají být v rámci analýzy nalezeny protipříklady [6].

Kód 2.1: Alloy specifikace nadřízených a podřízených pracovníků

```
module workplace

abstract sig Employee {}

sig Worker extends Employee {
} {
  some m: Manager | this in m.assign_tasks_to
}

sig Manager extends Employee {
  assign_tasks_to: set Employee,
  superior: lone Manager
} {
  this !in this.^@superior
  this != Boss => Boss in this.^@superior
  this != Boss => Boss !in this.^@assign_tasks_to
}

one sig Boss extends Manager {
} { no superior }

pred OneSuperior {
  all e: Manager - Boss | (one e.superior && Boss in e.^@superior)
}

pred show { }
run show for 5 but 3 Manager
```

2.3.2 B-method

B-method je metoda pro formální specifikaci, návrh a vybudování SW aplikací [38]. Podobně jako ostatní metody je i metoda B založená na teorii množin a logice prvního řádu. Staví na Abstract Machine Notation (AMN), ve které se vytváří abstraktní model popsaný pomocí stavového prostoru a přechodů. Tento model se pak převádí na konkrétní model, postupným přidáváním úrovní detailů vznikají tzv. *refinements* [17, 38]. Výsledkem je model, který se již překládá do programovacího jazyka se zárukou zachování specifikované funkčnosti. U modelů lze ověřovat, zda jsou samy o sobě konzistentní a ve vzájemné formální shodě [17, 56].

Model se zapisuje v AMN pomocí jednotlivých strojů (uvádí se konstrukcí `MACHINE`), které obsahují:

- název stroje,
- seznam proměnných, konstant, množin a jejich vlastností,
- invarianty (popis typů proměnných a jejich vztahů),
- inicializaci proměnných na začátku běhu,
- seznam operací tvořených podmínkou provedení a výsledkem [6, 38].

Jako další vývojový stupeň metody B je považováno Event-B, které přináší jednodušší notaci a nabízí lepší možnosti specifikace distribuovaných a souběžných systémů [38].

2.3.3 OCL

Object Constraint Language (OCL) je deklarativní jazyk pro specifikaci omezujících pravidel aplikovaných na UML model. Jazyk byl vyvinut původně v rámci metody Syntropy v IBM a nyní je standardem OMG. Jedná se o silně typovaný jazyk s předdefinovanými standardními datovými typy a kolekcemi. Dále OCL v základu obsahuje funkce pro práci s těmito kolekcemi a běžné operace. [6, 57]

Mimo dalších pomocných funkcí se v OCL definují samotná integritní omezení, kde každé obsahuje:

- popis kontextu, kterého se omezení týká (například třída nebo metoda),
- informaci, zda se jedná o omezení ve formě invariantu (obecně platného tvrzení) nebo podmínky před/po provedení nějaké akce,
- samotný výraz popisující omezení [57].

OCL těží především z návaznosti na velmi rozšířené UML. Velký počet CASE nástrojů pro UML má i podporu OCL a lze model snadno doplnit o specifikaci omezení.

Kód 2.2: Specifikace otevírání dveří pomocí metody B

```
MACHINE Doors
SETS
    DOOR_STATE = {closed, opened, closing}
    ALARM_STATE = {on, off}
VARIABLES
    state, alarm
INVARIANT
    doors : DOOR_STATE & alarm : ALARM_STATE
INITIALISATION
    doors := closed || alarm := off
OPERATIONS
    open  = PRE doors = closed & alarm = off
            THEN doors := opened & alarm := off
            END
    close = PRE doors = opened & alarm = off
            THEN doors := closing & alarm := on
            END
    lock  = PRE doors = closing & alarm = on
            THEN doors := close & alarm := off
            END
END
```

Kód 2.3: Příklad OCL omezení

```
context Person
    inv: self.age >= 0

context Person::mature(): void
    pre : self.age < 18
    post: self.age = 18

context Person::peopleCounter: int
    init: 0
```

2.3.4 VDM

Vienna Development Method (VDM) již v názvu prozrazuje, o jakou metodu se jedná a odkud metoda pochází. Jde o poměrně starou metodu vyvinutou v 70. letech 20. století ve vídeňských laboratořích IBM. VDM přinesla jako první řadu důležitých myšlenek, ze kterých ostatní metody později těžily a vy-

cházely. Jazyk VDM-SL byl v roce 1999 standardizován pod ISO a má dvě formy, první je tvořena pouze ASCII symboly a druhá používá matematické symboly. [6, 17]

Ve VDM-SL se popisuje systém jako aplikování funkcí na data a sestává se tedy z definic datových typů, funkcí a operací, které se nad daty provádí. Svou syntaxí připomíná pseudokód a jednodušší programovací jazyky. Funkcionalitu lze zapsat jako čisté funkce nebo pomocí operací, které mění stavové (globální) proměnné. [6, 36]

S rostoucím rozmachem objektově orientovaného programování vznikl jazyk VDM++, který umožňuje namísto importu a exportu modulů strukturování pomocí technik OOP: tříd, zapouzdření a vícenásobného dědění. Oba jazyky obsahují, podobně jako programovací jazyky, definice základních datových typů a běžné kolekce jako je množina, sekvence a mapa. [36]

Kód 2.4: Ukázka deklarace typů ve VDM-SL

```
types

String = seq of char | <nil>;
Year = nat | <nil>
inv e == e in set {1800,...,2017} union {<nil>};
Rating = nat
inv e == e in set {1,...,5};

Person::
    name    :String
    webpage :String;

Group  = seq of Person;

Author = Person | Group;

Book::
    author    :Author
    publisher :String
    title     :String
    year      :Year;

Review::
    reviewer  :Person
    book      :Book
    rating    :Rating;
```

2.3.5 Z-notation

Z-notation není kompletní metodou tvorby modelů, ale jak název napovídá, jedná se pouze o notaci, sadu konvencí pro užití matematických konstruktů k popisu výpočetních systémů (především SW, ale i HW) [58]. Z stojí na matematických základech z teorie množin a predikátové logiky prvního řádu. Součástí je mimo stavů a operací také schéma. Schémata jsou vzorem deklarací a omezení. Každý objekt v Z má svůj jeden pevně daný typ [6]. Z je notace založená na modelu, tudíž Z modeluje systém pomocí reprezentace jeho stavu, tzv. kolekce stavových proměnných, jejich hodnot a operací měnící tyto stavové hodnoty. Model, který je charakterizován svými operacemi se v Z nazývá abstraktní datový typ. [37, 58]

Ačkoliv Z není metoda, může notace Z podporovat mnoho různých metod. To, co se pomocí Z zapíše, je čistě na autorovi a nehraje roli, jestli bude popisovat členění informačního systému na moduly nebo například modelovat reálný fyzikální systém. Nejedná se ani o programovací jazyk, ani o spustitelný předpis (není exekuční). [6, 37]

ResearchGroup

$$director : PERSON$$
$$members : \mathbb{P}\ PERSON$$
$$director \in members$$

JoinResearchGroup

$$\Delta\ ResearchGroup$$
$$new : PERSON$$
$$\textbf{if } new \notin members \textbf{ then } members' = members \cup \{new\}$$
$$\textbf{else } members' = members$$
$$director' = director$$

Obrázek 2.6: Specifikace výzkumné skupiny a přidání nového člena v notaci Z

Matematická notace se skládá z malého jádra, ke kterému je přidán tzv. Z Mathematical Tool-Kit obsahující řadu užitečných objektů a operátorů [58]. Opět nejde o nějaký SW modul nebo program, ale o sadu matematických teorií, definic a zákonů rozšiřující základní jádro Z. Díky silným matematickým základům lze model ověřovat pomocí formálních důkazů. [6, 17, 49]

Notace Z není navržena pro zpracování strojem, ale pro lidi. Původně tato notace, která se začala používat v 80. letech 20. století, byla určena k návrhu systému pomocí tužky, papíru a pravítka [37]. Nyní je notace podporována řadou nástrojů, které jsou již nedílnou součástí práce s notací, pokud se jedná o větší modely. Nejznámější a nejpoužívanější sadou nástrojů pro práci s notací Z je CZT: Community Z Tools [59].

2.4 Společné rysy metod

Na všech uvedených a popsaných metodách lze pozorovat jisté shodné postupy a prvky, kterými se konceptuální modely zapisují. Obecně se vždy zachycují jednotlivé koncepty jako entity, jejich vztahy a poté ve větší či menší míře různá omezení, která zmenšují množinu možných instancí modelu. Mezi typická omezení patří kardinalita vztahů, unikátnost atributů a další jednoduchá pravidla pro omezení domén atributů.

Další specifickou skupinou omezení jsou speciální případy vztahů a entit. Například vztahy generalizace, kompozice, agregace nebo v OntoUML i různé další stereotypy vztahů představují klasickou relaci s předdefinovaným omezením a specifickým významem na úrovni metamodelu. Obdobně je tomu u entit a pohledů na ně.

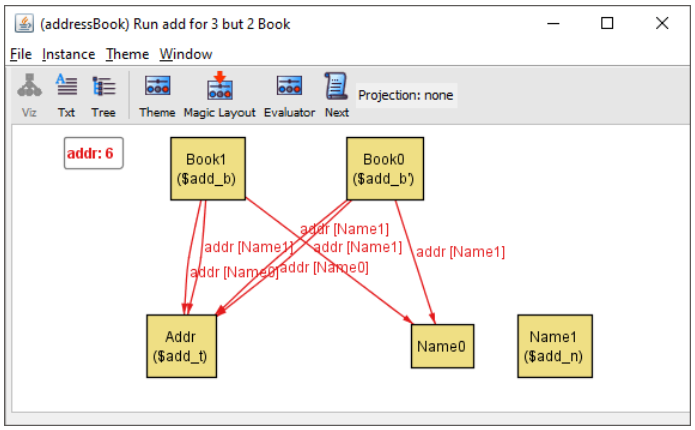
Kapitola 3

Přehled nástrojů

V předchozí kapitole byly představeny různé metody pro zápis konceptuálních modelů. Jedním ze zmíněných klíčových faktorů úspěchu těchto metod je dobrá nástrojová podpora. Tato kapitola obsahuje stručný popis vybraných nástrojů pro práci s představenými metodami. Jejich funkcionalita, výhody a nevýhody budou zohledněny při vývoji vlastního systému pro reprezentaci konceptuálních modelů v jazyce Haskell a podpůrných nástrojů.

3.1 Alloy a Alloy Analyzer

V nástroji Alloy lze vytvářet a upravovat modely ve stejnojmenném jazyce pro formální specifikace. Jedná se o jednoduchý textový editor, který pouze zvýrazňuje syntaxi. Důležitou součástí je však Alloy Analyzer, který umí model spouštět a případně upozornit na chyby v syntaxi. Umožňuje model vizualizovat, procházet instance modelu a detailně upravovat nastavení samotného analyzátoru. [6]

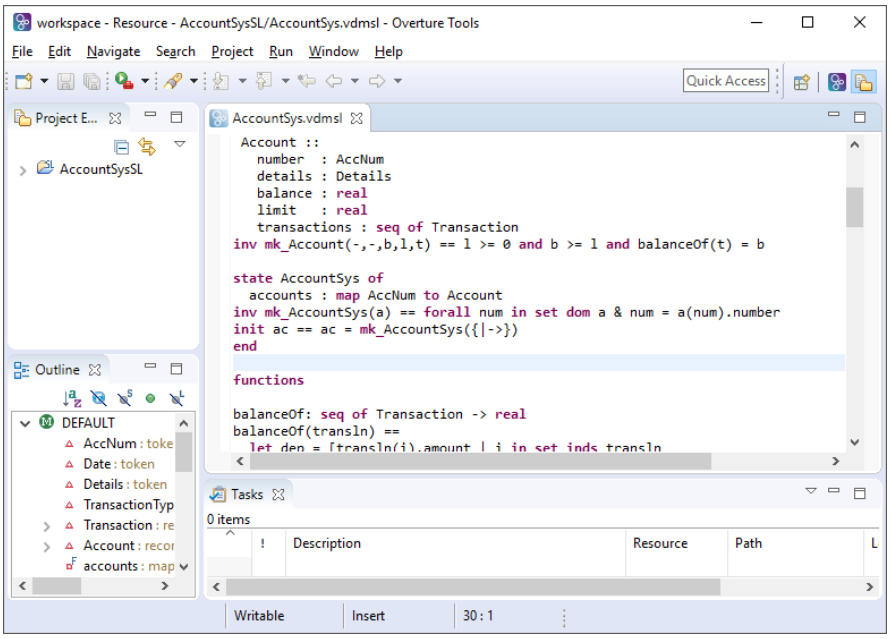


Obrázek 3.1: Vizualizace instance modelu v Alloy

Alloy Analyzer umí generovat příklady a hledat protipříklady. V pozadí se specifikace v Alloy přeloží do booleovské formule, která se poté řeší pomocí SAT řešiče. Konkrétní SAT řešič si je možné vybrat z nabídky a upravit jeho parametry. Nástroj je napsaný v jazyce Java a jako výchozí je tudíž vybrán SAT4J. Samotnou booleovskou formuli lze uložit do souboru v běžném formátu DIMACS nebo KodKod (Java kód). Stejně tak lze exportovat i vygenerované instance do formátu XML nebo DOT. Právě díky těmto možnostem nástroje Alloy Analyzer se objevují překladače z různých jiných modelovacích jazyků (OntoUML, Z notace, ...) do jazyka Alloy [16, 37].

3.2 Overture

Overture [60] je přední open-source nástroj pro práci s VDM. Jedná se o IDE postavené na platformě Eclipse (Java) obsahující nástroje pro vývoj a analýzu VDM modelů. Podporuje různé dialekty, jak ISO standard VDM-SL, tak i VDM++ a rozšíření pro real-time VDM-RT. Mimo intuitivního uživatelského rozhraní podobného klasickým vývojářským IDE nabízí běžná zpříjemnění práce jako je například zvýrazňování syntaxe a aktuálních výrazů nebo přehledné zobrazení struktury projektu a definic.



Obrázek 3.2: Specifikace v jazyce VDM-SL pomocí nástroje Overture

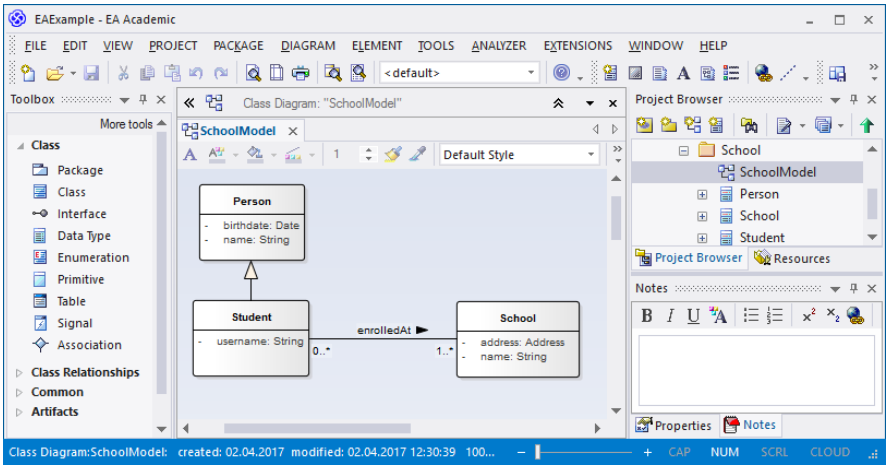
IDE přináší také možnosti debuggingu a ověřování modelu. Debugger má opět klasické chování jako u vývoje SW aplikací, lze nastavovat body zastavení s podmínkami nebo počítáním přístupů, krokovat, generovat logy či

report o pokrytí. Ověřování zahrnuje jednak kontrolu správnosti zápisu modelu z pohledu syntaxe a použitých datových typů, která probíhá během psaní (on-the-fly). Důležitější je však ověřování správnosti modelu pomocí formálních důkazů, kombinatorického testování a vyhodnocování podmínek i invariantů při simulaci. Důkazy lze zobrazit ve specializovaném prohlížeči, který je součástí aplikace.

Obsahem jsou i nástroje pro export a import. Nástroj umožňuje exportovat dokumentaci k vytvořenému modelu ve formátu LaTeX. Model lze rovněž ukládat ve formě UML diagramu tříd ve formátu XMI, v tomto případě je možný i import. Tým Overture také pracuje na nástroji pro generování kódu v jazyce C přímo z VDM [60].

3.3 Enterprise Architect

Enterprise Architect [61] je známý komerční nástroj vyvíjený společností Sparx Systems založený na UML. Podporuje mimo jiné kolaborativní modelování problémových domén, řízení projektů, tvorbu software od návrhu uživatelského rozhraní přes rozvržení komponent a modulů až po komunikaci jednotlivých objektů a to vše v rámci celého vývojového cyklu SW. Mimo UML podporuje i řadu dalších jazyků a standardů jako například BPMN, OCL, OWL, RDF, ArchiMate, SoaML a další. [62]



Obrázek 3.3: UML diagram tříd v Enterprise Architect

Nástroj umožňuje export jednotlivých diagramů, generování dokumentace pomocí šablon a nabízí i rozšíření pro kancelářský balík Microsoft Office. V rámci své podpory MDD umí nastavitelné generování kostry kódu pro podporované programovací jazyky. Na druhou stranu lze z existujícího kódu nebo databáze vygenerovat příslušné modely a dokumentaci.

Nespornou výhodou je dobře zdokumentované API pro tvorbu rozšíření, kterých je díky tomu celá řada. V Enterprise Architect jsou integrovány nástroje pro verzování, validace modelů pomocí základních pravidel včetně OCL i například simulátor a debugger pro různé modely. Jedná se o komplexní nástroj, který se stále vyvíjí a zdokonaluje. [62]

3.4 TestEra

TestEra není na rozdíl od ostatních zde uvedených nástrojů editorem ani vývojovým prostředím, nýbrž frameworkem. Tento specializovaný framework umožňuje testování programů napsaných v jazyce Java pomocí Alloy specifikací. Použití je velmi snadné a pro programátory minimálně zatěžující, jednotlivé specifikace se totiž zapisují do anotací `@TestEra` přímo ke třídám a jejich metodám. [63, 64]

V anotacích lze definovat invarianty, podmínky před provedením (předpoklady) a po provedení. Další anotace umožňují pracovat s nastavením převodu do Alloy, konkrétně zda se nějaká část kódu má z překladu vynechat, vytváření `run` příkazů a nastavení limitů generovaných testů [6, 64]. Anotace mohou být složité a několikařádkové, ale i velmi jednoduché jako v příkladu 3.1.

Kód 3.1: Jednoduchá ukázka TestEra anotací v Java třídě `Person`

```
@TestEra(
    invariant = { "all p:Person | p.age >= 0" }
)
class Person{
    public static final int MATURE_THRESHOLD = 18;
    private int age = 0;

    @TestEra(
        preCondition = { "!this.isMature" },
        postCondition = { "this.isMature" },
        runCommand = "10 Person"
    )
    public void mature(){
        age = MATURE_THRESHOLD;
    }

    public boolean isMature(){
        return age >= MATURE_THRESHOLD;
    }
}
```

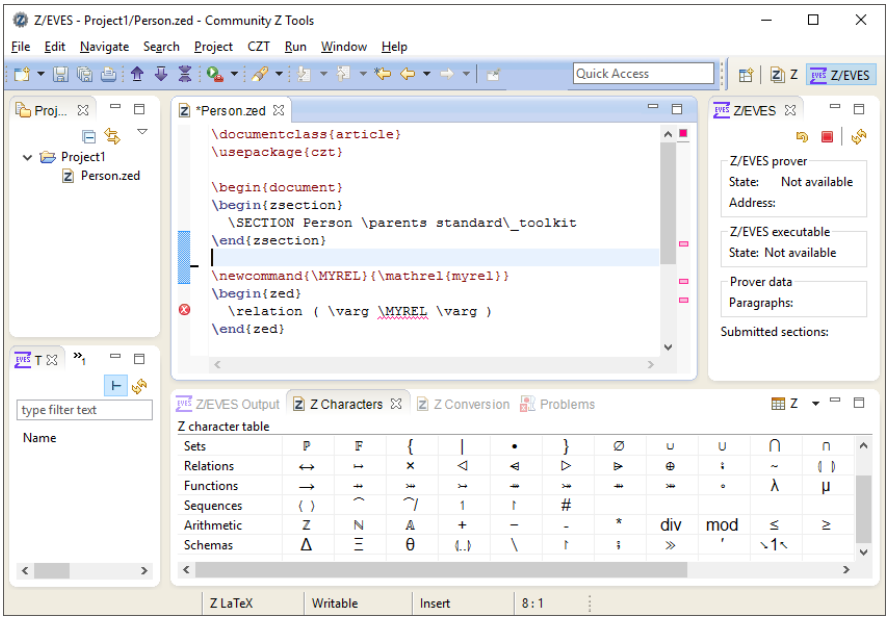
Plugin do Eclipse IDE z anotací umí vygenerovat Alloy model, na jeho základě generovat testy a zobrazovat vizualizace stejně jako Alloy Analyzer. Navíc je možné vygenerovat i JUnit testy, které odpovídají příkladům z Alloy vytvářených pomocí SAT řešiče. Nástroj tak vhodně kombinuje sílu testování přes Alloy a jednoduchý zápis s možností vlastních úprav. [6, 63]

Na tomto nástroji lze pozorovat ambice přinést metody formální specifikace do komerčního prostředí, kde patří jazyk Java k jedněm z nejpoužívanějších. Současně se snaží práci s formální specifikací usnadnit a umožňuje ji efektivně používat přímo v prostředí určeném pro vývoj aplikací. [63, 64]

3.5 CZT: Community Z Tools

Community Z Tools [59] je komunitní sada nástrojů pro práci s notací Z postavená na Eclipse platformě. V tom se shoduje například s nástrojem Overture a je tím dána i velká vizuální a funkční podobnost obou nástrojů. Nabízí shodnou základní funkcionalitu jako je zobrazení struktury projektu a definic, našeptávání nebo zvýraznění a kontrola syntaxe. Dále umožňuje zápis Z jak v jeho ASCII podobě, tak i matematickém zápisu, při kterém nabízí výběr matematických symbolů přímo na obrazovce.

To, že se jedná o sadu nástrojů, je znát z modulů. Velkou část tvoří moduly pro konverzi modelu z notace Z do jazyků Alloy, ZML a B. Dále pak obsahuje i moduly pro animace, vizualizace a obohacení uživatelského rozhraní. Součástí je také nástroj Z/EVES sloužící k formálním důkazům splnitelnosti.

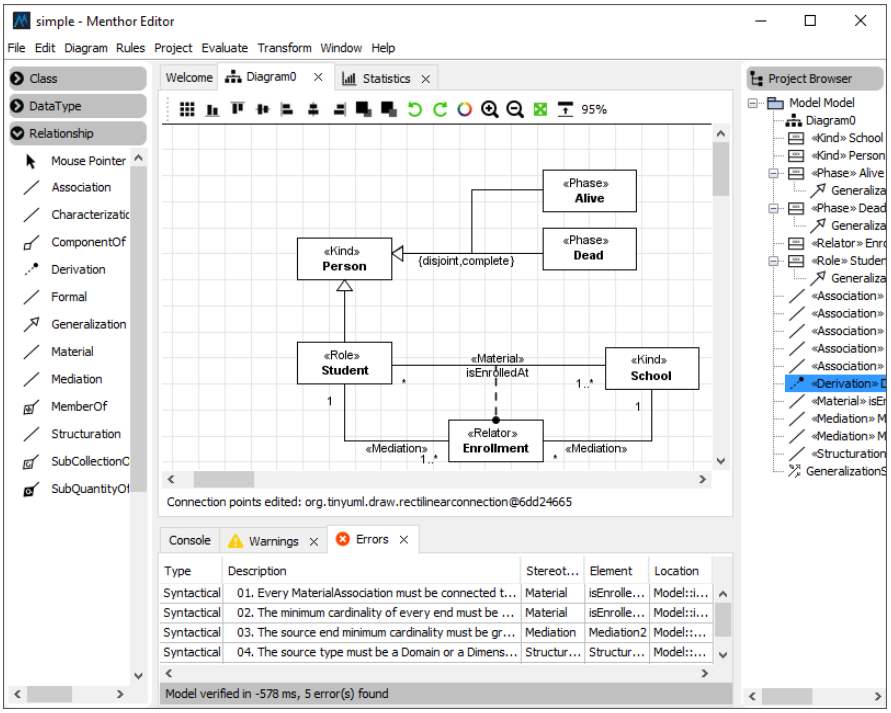


Obrázek 3.4: Z LaTeX soubor v CZT IDE

Bohužel tento ambiciózní projekt nepůsobí udržovaně za prvé tím, že je stále na již překonaném SourceForge.net s velmi nízkou aktivitou, a dále i tím, že verze 2 je stále označena jako beta a výrazný postup ve vývoji není znát. V uvedených informacích o projektu jsou chyby a nejasnosti [59]. Získat poslední verzi, se kterou se na svých stránkách chlubí, není vůbec triviální.

3.6 Menthor Editor

Skupina Menthor nabízející služby v oblasti návrhu systémů, konzultací a školení vyvíjí open-source Menthor Editor [65] jako nástupce nástroje OLED, původního OntoUML editoru vytvořeného přímo výzkumnou skupinou NEMO. Mimo běžného návrhu OntoUML modelů nabízí přidání omezení v OCL a překlad OntoUML modelu do jazyků UML, OWL či RDF. Z modelu je možné vygenerovat také dokumentaci ve formátu SBVR, který je srozumitelný i osobám s netechnickým zaměřením. Další zajímavou funkcionalitou je možnost validování, která ověřuje nejen správnost syntaxe, ale hledá i tzv. *anti-patterns*.



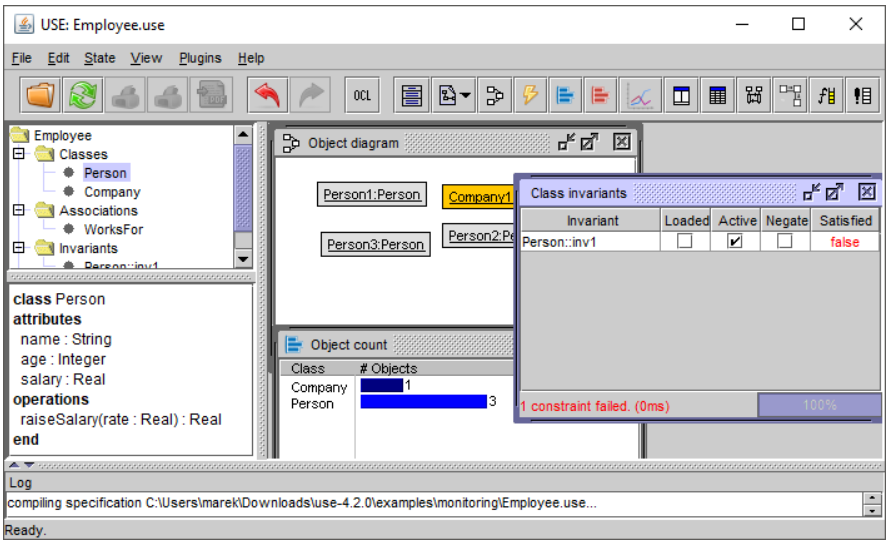
Obrázek 3.5: Příklad OntoUML modelu v Menthor Editor se zobrazením chyb

Mimo samostatného editoru napsaného v jazyce Java je dostupný i plugin pro Enterprise Architect, který umožňuje vytvářet OntoUML modely přímo v něm a potom je převést v případě potřeby do Menthor Editor. Součástí se v posledních verzích stal i překladač z OntoUML do Alloy, což umožňuje

detailnější validaci modelů včetně vizualizace v Alloy Analyzer či případně anotování Java aplikací pomocí TestEra.

3.7 USE

UML-based Specification Environment [66], zkráceně USE, je CASE nástroj pro specifikaci a validaci informačních systémů. V nástroji je možné použít podmnožinu jazyků UML a OCL, označovanou jako specifikační jazyk USE, k vytvoření modelu a zápisu omezení a pravidel. Nástroj je multiplatformní opět díky využití jazyku Java, dále umožňuje vytvářet pluginy pro rozšíření funkcionality.



Obrázek 3.6: Zobrazení objektů a invariantů v nástroji USE

Ačkoliv lze v nástroji používat omezenou část UML a OCL, díky jednoduchému ovládání, ať už pomocí GUI nebo v příkazové řádce, je možné snadno specifikovat model včetně invariantů, podmínek před a po provedení akce. GUI nabízí řadu pohledů na model a omezení, například pohled na třídy, invarianty, zásobník volání, komunikační diagram, počty objektů a jejich spojení či jiné vlastnosti objektů. Základem je vyhodnocení OCL výrazu a automatické kontroly modelu za účelem jeho validace.

Část II

Praktická část

Kapitola 4

Uplatnění programovacích jazyků pro konceptuální modelování

V této kapitole se rozebírají zjevné i hypotetické výhody a nevýhody použití programovacích jazyků jako prostředku pro reprezentaci konceptuálních modelů. Dále se kapitola zabývá vlivem jednotlivých vybraných vlastností různých programovacích paradigmat a skupin jazyků při jejich použití pro specifikaci systémů v rámci konceptuálního modelování. V závěru kapitoly jsou analyzovány existující přístupy pro propojení konceptuálního modelování se související implementací.

4.1 Programovací jazyk jako prostředek modelování

Programovací jazyk je prostředek pro zápis algoritmů ve formě zdrojového kódu programu pro řešení úlohy. Nižší programovací jazyky poskytují velmi malou nebo dokonce žádnou abstrakci od toho, jak funguje procesor počítače a pracuje se tak například přímo s jeho instrukcemi a symbolickými adresami. Oproti tomu vyšší programovací jazyky nabízí již větší míru abstrakce za účelem lepší srozumitelnosti, jednodušší práce se zdrojovým kódem a v důsledku úspory času. [67]

Vyšší programovací jazyky se dále dělí na imperativní, založené na zápisu posloupností výpočtů v rámci procedur nebo metod objektů, a deklarativní, kde posloupnosti příkazů nahrazují deklarace a definice říkající, co má jakou hodnotu, podobně jako v matematice [67]. Oba přístupy mají své silné i slabé stránky a je nutné zvážit, který je v dané situaci vhodnější.

Objektově orientované programování stojí stranou od tohoto dělení, neboť jej lze uplatnit napříč paradigmaty. Například funkcionální jazyk Scala je

objektově orientovaný ve smyslu, že všechno je objektem a to včetně funkcí. Některé jazyky se označují jako multiparadigmatické, pokud v nich lze psát program za použití různých programovacích paradigmat, a snaží se tím umožnit programátorům co největší volnost [67].

4.1.1 Výhody

- Modelu dobře rozumí programátoři implementující systém a nemusí se učit jiný jazyk pro formální specifikace ani pracovat s diagramy a textovou dokumentací domény. [6, 67]

- Není potřeba generovat nebo psát ručně zdrojový kód dle modelu, protože model již je zdrojovým kódem. Tím dochází k eliminaci zanedbání částí modelu a chybné transformaci, tudíž není nutné provádět verifikaci, ale pouze ověřit správnost zápisu a model validovat. [47]

- Změna ve zdrojovém kódu znamená změnu modelu a naopak. Není nutné ani možné změny provádět v modelu a kódu zvlášť. Konceptuální model je vždy konzistentní s implementací a jednotlivé verze lze porovnávat a ověřovat proti sobě. [47]

- Chyby modelu mohou být kontrolovány přímo kompilátorem v závislosti na použitém jazyku a jeho konstrukcích, knihovnách a rozšířeních.

4.1.2 Nevýhody

- Programovací jazyk nemusí být dostatečně expresivní a zápis některých skutečností, které je nutné popsat konceptuálním modelem, je složitý nebo dokonce nemožný. [18]

- Samotný zápis konceptuálního modelu v programovacím jazyce je v závislosti na syntaxi jazyka málo srozumitelný pro osoby, které daný programovací jazyk neznají. Proto je nutná dobrá nástrojová podpora, která umožní například i vizualizace modelů a instancí. [29]

- Pokud je požadováno, aby i tvůrce modelu mohl být neznalý programování, je nutné dodat i jednoduchý nástroj pro tvorbu modelů. [6, 49]

4.1.3 Shrnutí

Pokud programovací jazyk má dobrou úroveň expresivity a současně je dosažena dobrá nástrojová podpora, pak lze vytvořit metodiku tvorby konceptuálních modelů v tomto jazyce, která přináší výhody znamenající úsporu nákladů na vývoj i údržbu informačního systému.

4.2 Vlastnosti jazyků

4.2.1 Typová kontrola

Statická typová kontrola znamená, že datové typy všech konstant, proměnných a výrazů jsou známé v době kompilace, a to buď díky tomu, že jsou explicitně uvedeny ve zdrojovém kódu nebo že jsou automaticky odvozeny. Kontrola zajistí, že kód, ve kterém se s typy pracuje chybně, není zkompilován a programátor je zpravidla chybovým hlášením naveden k opravě chyby. Nevýhodou je logicky složitější proces kompilace, a tudíž i delší čas překladu. Naproti tomu dynamická typová kontrola probíhá až v době běhu programu a není nutné při kompilaci znát typy a uvádět je ve zdrojovém kódu. Výhodou je vyšší flexibilita, ale mezi zásadní nevýhody patří nutnost kontroly typů za běhu a následná možnost pádu programu v důsledku chybné práce s typy. Lze se setkat s různými přístupy umělého dopisování typů do dynamicky typovaných jazyků buď ve formě anotací nebo jako součást dokumentace ve zdrojovém kódu. Některé staticky typované jazyky zase naopak umožňují umělé vytváření dynamických typů za účelem dosažení vyšší flexibility v místech, kde je to vhodné (například pro tzv. *duck typing*). [67, 68]

Dalším přístupem, jak posuzovat typovou kontrolu, je na slabé a silné typy. Silně typovaný jazyk neumožňuje na rozdíl od slabě typovaného implicitní konverzi při provádění různých operací. Příkladem může být součet textového řetězce a celého čísla, v silně typovaném jazyce tato operace znamená chybu, pokud není explicitně stanoveno, na jaký typ má být první či druhý operand převeden. Chyba se projeví buď za běhu v případě dynamické typové kontroly nebo v době kompilace, pokud je typová kontrola statická. Naopak ve slabě typovaném jazyce dojde k implicitní konverzi například čísla na řetězec a poté dojde ke spojení dvou řetězců. V jiném slabě typovaném jazyce může být ale nejprve převeden řetězec na číslo a poté se provede součet dvou čísel. Tím mohou vznikat chyby a neočekávané výstupy. [68]

Pro účely konceptuálního modelování v programovacím jazyce se očividně více hodí silně a staticky typované jazyky, neboť provedou největší množství kontroly již v době kompilace a běh poté může být rychlejší a výrazně bezpečnější. Podle dokázaného tvrzení Robina Milnera [69] platí: „Dobře typovaný (*well-typed*) program se nemůže pokazit.“. Silná typová kontrola navíc nutí používat hierarchii typů ve smyslu různých úrovní abstrakce za účelem dosažení polymorfismu. [67, 69]

4.2.2 Algebraické datové typy

Algebraický datový typ, je takový typ, který je složený z jiných datových typů pomocí algebraický operací součet a součin. Tím je snadno určen počet hodnot, kterých může proměnná typu nabývat. Dále umožňují snadnou typovou kontrolu, *pattern matching* a přehlednou tvorbu rekurzivních datových typů

(například seznam je prázdný nebo obsahuje hodnotu a další seznam). Tyto typy jsou běžné pro funkcionální jazyky a na rozdíl od tříd v OOP neumožňují zapouzdření operací a ty musí být definovány zvlášť, případně v typových třídách či jiných konstrukcích umožňujících polymorfismus. [70, 71]

Pomocí algebraických datových typů lze snadno a přehledně zapsat složité datové struktury i s nimi dále pracovat. Nutí programátora zamyslet se nad strukturou typu a souvisejících vlastnostech s odstupem od jednotlivých operací, které budou s typem pracovat. Z těchto důvodů se jeví jako vhodné pro popis struktury entit v konceptuálních modelech.

4.2.3 Třídní systém a aktorový model v OOP

Třídy jsou šablonou pro vytváření objektů v OOP a umožňují zapouzdření dat náležících objektu a souvisejících funkcí (metod) pro práci s objektem. Od běžných typů se liší tím, že třída je již konkrétní implementací typu třída, neboť popisuje konkrétní datovou strukturou s nějakým definovaným rozhraním. Pomocí vícenásobné dědičnosti a abstraktních tříd lze vybudovat hierarchii podobně jako je tomu v reálném světě. [67, 68]

Existují OOP jazyky, které koncept tříd nemají a používají jiné přístupy, například prototyping v případě JavaScriptu. Aktorový model vychází z běžného života a biologických procesů, kdy jednotlivci mezi sebou komunikují a působí na sebe pomocí zpráv. Veřejně dostupné zprávy objektu jsou definovány jako rozhraní. Ostatní aktoři/objekty mají přístup pouze k tomuto rozhraní a jsou odstíněni od zapouzdřeného vnitřního fungování objektu, které často vede k delegaci dílčích úkonu na další objekty. [67]

Jednotlivé koncepty lze modelovat jako abstraktní či konkrétní třídy, kde se propojuje dohromady struktura i chování. Tato kombinace však může vést v nepřehledný zápis modelu a následné složité úpravy. Výhodou je však přehledná hierarchie pomocí vícenásobné dědičnosti. [67, 69]

4.2.4 Závislé typy

Závislý datový typ (dependent type) je takový datový typ, jehož definice závisí na hodnotě [72]. Příkladem takového typu mohou být sudá celá čísla (dělitelná dvěma beze zbytku), posloupnost čísel ve vzestupném pořadí, řetězce obsahující pouze znaky české abecedy a podobně. Tím lze snadno dosáhnout typů například osoby starší 18 let nebo automobily do 3,5 tuny.

Tato vlastnost jazyka přináší výhody, a to jak obecně, tak i za účelem tvorby konceptuálního modelu. Jen pomocí závislých typů lze popsat velkou část omezení, které se budou kontrolovat již během kompilace. Výhoda tedy spočívá jednak v detailnějším ověřování již v době kompilace (ne až v době běhu), ale i ve vyšší přehlednosti zdrojového kódu díky eliminaci potřeby kontrolovat validitu instancí v určitých kritických místech. [71, 72]

Závislé typy jsou doménou především čistě funkcionálních programovacích jazyků (například Agda, Idris, Cayenne a Matita), ale jejich podpora se vyskytuje i v některých multiparadigmatických jazycích jako F* nebo ATS (Applied Type System). Umožnění vytváření závislých typů výrazně zvyšuje expresivitu programovacího jazyka. Za závislými datovými typy stojí silné matematické základy vycházející již z typovaného lambda kalkulu [72].

4.2.5 Referenční transparence

Výraz v programu je referenčně transparentní, pokud jej lze nahradit jeho hodnotou a nedojde ke změně chování. Prakticky řečeno výraz nemá žádný vedlejší efekt než samotné vyhodnocení (nečte vstup z klávesnice, nemění stav programu pomocí globálních proměnných a podobně). Tato vlastnost umožňuje nejen zefektivnění výpočtů pomocí specializované optimalizace kódu, uchování výsledků a paralelizace, ale napomáhá ověřování správnosti programu díky předvídatelnému chování těchto výrazů. [67, 70]

4.2.6 Interní DSL

Jak již bylo zmíněno v úvodní kapitole, některé jazyky umožňují vytvářet za pomoci jejich konstrukcí DSL [45]. Takový jazyk může být použit přímo pro popis struktur a chování konkrétní domény. Jazyk může být ale také navržen na vyšší úrovni abstrakce tak, aby v něm bylo možné popsat libovolnou doménu a reprezentovat libovolný konceptuální model. Možnost tvorby DSL tak může napomoci při přípravě konstrukcí pro zápis modelů a současně vypovídá o expresivitě samotného programovacího jazyka.

4.2.7 Strategie vyhodnocování

Programovací jazyky se při vyhodnocování výrazů drží strategie, která je buď striktní nebo nestriktní a říká, kdy mají být argumenty vyhodnoceny a jaký druh hodnoty má být předán. Ve striktním vyhodnocení jsou argumenty vždy plně vyhodnoceny před jejich použitím. Předání argumentu probíhá buď hodnotou (call by value), kdy je hodnota zkopírována, nebo referencí (call by reference), kdy se předá odkaz na proměnnou s předávanou hodnotu. Dále je definováno i předání sdílením (call by sharing), které se od předání reference liší ve způsobu viditelnosti pro volajícího. [67]

Nestriktní vyhodnocení naopak vyhodnocuje argumenty, až když je nutné znát jejich hodnotu. Volání jménem argumentu (call by name) znamená, že jsou vyhodnoceny pokaždé, když je užito jejich jméno v těle funkce. Zlepšení přináší volání podle potřeby (call by need), které provede vyhodnocení, až je to nutné a jen jednou, to ale může způsobit potíže, pokud má takové vyhodnocení nějaký vedlejší efekt. V takovém případě je potřeba tyto situace vhodně ošetřit například užitím monád. Tento způsob vyhodnocování bývá označován jako

líný, neboť nevyhodnocuje nic, než je to opravdu potřeba, a nic nedělá víckrát, než je třeba, což je nespornou výhodou. [67]

4.2.8 Reflexe

Vlastnost programovacího jazyka, která umožňuje programu za běhu zjišťovat, procházet a případně i měnit svou vlastní strukturu a chování, se nazývá reflexe. Za běhu lze v rámci programu získat informace o libovolném typu, jeho druh, atributy a funkce včetně jejich názvů a další konstrukce. Reflexe tak najde využití například pro specializované kontroly i vizualizaci konceptuálního modelu zachyceného ve zdrojovém kódu. [73]

Pokud jazyk reflexi neumožňuje lze to buď řešit pomocí specializovaných knihoven, existují-li, nebo přímo prací se samotným zdrojovým kódem jako vstupním souborem a vytvořením si modelu uvnitř pomocné aplikace. Některé kompilátory a interpretry poskytují rozhraní pro práci se zdrojovým kódem a není nutné provádět lexikální, syntaktickou a sémantickou analýzu znovu.

4.3 Existující řešení

V oblasti zápisu konceptuálních modelů přímo do zdrojového kódu neexistují v současné době žádná široce známá a užívaná řešení. Při převodu modelu do kódu se sice využívají hojně různé frameworky a knihovny podporující princip DRY, slabou vazbu a oddělení zájmů (SoC), ale to nelze považovat přímo za prostředek pro konceptuální modelování a provázání s kódem. [26, 28]

V této části jsou detailněji popsány významné přístupy zmíněné v úvodní kapitole, které pracují s myšlenou provázání konceptuálního modelu s implementací ve formě software. Jejich analýza poslouží jako inspirace při návrhu vlastního řešení v jazyce Haskell. Přestože se v některých případech nejedná o přímé spojení zdrojového kódu s modelem, lze pozorovat souvislosti na převodu modelu na kód s využitím různých návrhových vzorů a vlastností cílového jazyka. Neustálý vývoj těchto postupů také ukazuje na potřebu usnadnění a zkvalitnění tvorby informačních systémů.

4.3.1 Teorie normalizovaných systémů

Teorie normalizovaných systémů říká, jak správně navrhovat a vybudovat informační systémy vzhledem k jejich evolvabilitě. Stojí na základní myšlence, že s postupným rozvíjením informačního systému v čase ve formě nových funkcionalit či jiných úprav se zákonitě zvyšuje jeho složitost, což se odráží ve zhoršující se struktuře systému a použitelnosti, pokud není vynaložena práce na údržbu nebo snížení komplexity. Zmíněná zhoršující se struktura a použitelnost systému zvyšuje náklady na další rozvoj a v určitém okamžiku už může být výhodnější vytvořit systém nový. [43]

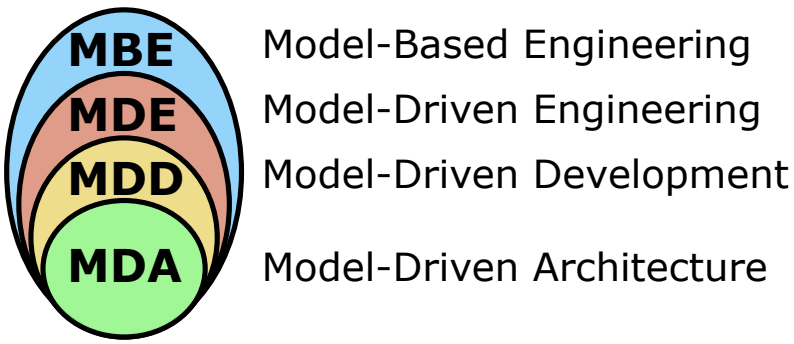
Tento problém se objevuje i při použití modulární architektury, neboť se zpravidla jedná o statickou modularitu a samotná architektura není dobře měnitelná v čase se zachováním ostatních implementovaných funkcí. Cílem normalizovaných systémů je dosáhnout těsného propojení změny na organizační úrovni a její promítnutí v architektuře systému a implementační úrovni. Teorie normalizovaných systémů využívá běžných principů ze softwarového inženýrství jako jsou návrhové vzory, oddělení zájmů (SoC), modularita, zapouzdření a transparence verzí dat i akcí. [26, 43]

Teorie pochází z Antverpské univerzity, kde rovněž vyvíjí nástroj pro generování evolvabilních normalizovaných systémů v jazyce Java na základě modelu zapsaného jednoduchou specifikací. Generování systémů je zatím jen úzce zaměřeno na Java EE CRUD systémy. [43]

Nevýhodou výsledného systému je však jeho nízká přehlednost v rámci zdrojového kódu, což znesnadňuje manuální úpravy. Dále při změně na organizační úrovni je nutné adekvátně změnit specifikaci a systém přegenerovat, tím však může vzniknout kolize právě v důsledku manuálních úprav nezachycených ve specifikaci. Řešením by bylo umožnění zachycení drobných úprav přímo v modelu a tím omezit zasahování do vygenerovaného systému, které je zjevně z pohledu konzistence nežádoucí. [26, 43]

4.3.2 MDE, MDD a MDA

Pojmenováním oblasti, která se zabývá vývojem informačních systémů na základě modelů, je MBE neboli inženýrství založené na modelu. Do té následně spadá modelem řízené inženýrství, do kterého patří modelem řízený vývoj jako paradigma vývoje. V některých zdrojích se MDE a MDD staví na stejnou úroveň a pojmy se libovolně zaměňují. Přístupem OMG jak pracovat s MDD je MDA, která využívá právě OMG standardů. [26, 74]



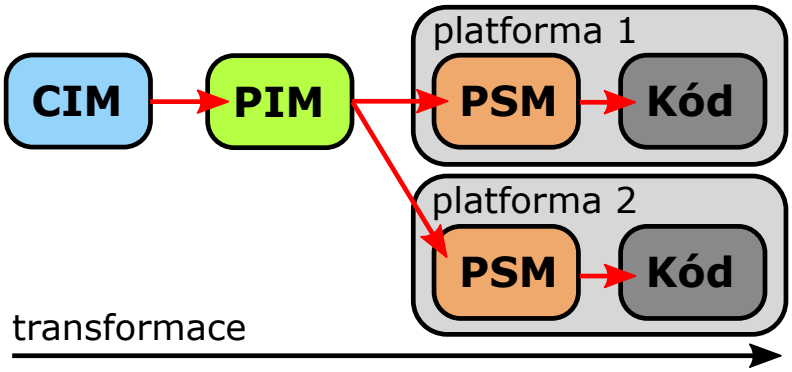
Obrázek 4.1: Souvislost MBE, MDE, MDD a MDA (podle [26, 74])

MDA bylo zahájeno již v roce 2001 a později bylo revidováno. Hlavní myšlenkou je umožnit modelovat v rámci průmyslových standardů OMG, takové

modely jsou poté označovány jako *MDA Models*. Modely se dělí podle MDA hledisek na:

1. výpočetně nezávislý model (CIM),
2. platformě nezávislý model (PIM) a
3. platformě specifický model (PSM) [28].

V uvedeném pořadí se modely přibližují implementaci, v CIM je zachycena problémová doména, PIM přidává informaci o realizaci ale jen do té míry, aby byl použitelný pro jakoukoliv cílovou platformu. Na úrovni PSM jsou definovány již detaily o implementaci a takový model je převoditelný na zdrojový kód nebo alespoň jeho základ, což je v dnešní době běžnější. Pokud PSM neobsahuje všechny nutné podrobnosti potřebné k implementaci, označuje se jako abstraktní. [28, 44]



Obrázek 4.2: Přechod od MDA modelů ke kódu (podle [26, 28])

Změny v modelech výše v hierarchii vedou ke změnám modelů níže a to může způsobovat nekonzistenci, vše závisí na možnostech nástrojů transformovat modely do nižších vrstev až do samotného kódu. Mimo nástrojů pro transformaci modelů MDA definuje také nástroje pro analýzu, vytváření a kompozici modelů, testování, verifikaci, validaci, simulaci, práci s metadaty a reverzní inženýrství. [26, 28]

4.3.3 Programování konceptuálních modelů

Článek *Conceptual Model Programming: A Manifesto* [47], již zmíněný v teoretické části této práce, poukazuje na potřebu programování pomocí konceptuálních modelů. V tomto směru by neměli programátoři psát tradičně řádky zdrojového kódu, ale místo toho pouze vytvářet konceptuální modely. To však vyžaduje prostředek pro zápis holistického konceptuálního modelu včetně uživatelského rozhraní, databáze a případně i napojení na externí prostředky jako

například práce s API jiných aplikací. Otázkou je, zda takový přístup je vůbec vhodný. Většinou je v konceptuálním modelu znát snaha oddělení esence problémové domény od nějaké konkrétní implementace.

Dle manifestu konceptuální model nahrazuje zdrojový kód aplikace a je sám o sobě přímo kompilovatelný, tím zaniká problém s nekonzistencí mezi modelem a konkrétní implementací. Další výhodou spustitelného konceptuálního modelu je fakt, že odstraňuje problémy s evolvabilitou řešené v rámci teorie normalizovaných systémů různými složitými návrhovými vzory a postupy generujícími složité zdrojové kódy, do kterých programátoři musí složitě doplňovat nízkoúrovňové úpravy.

CMP se staví na úroveň MDE, kdyby navíc zahrnovalo i plnou automazizaci procesu, tedy MDE bez nutnosti tradičních úprav vygenerovaného kódu, což je v současné době možné jen u velmi jednoduchých experimentálních systémů bez jejich dalšího postupného rozvoje. Myšlenky CMP se ukazují jako vhodné nejen v akademické sféře, ale i mezi velkými hráči v komerčním prostředí, kde se vyvíjí různé kompilátory konceptuálních modelů, jako například IBM Rational Rhapsody nebo WebRatio [47].

Kapitola 5

Analýza požadavků na reprezentaci modelů

V této kapitole jsou shrnuty požadavky na reprezentaci konceptuálních modelů na základě provedené rešerše v předchozích kapitolách. Tyto požadavky poslouží jako podklad pro návrh systému kódování konceptuálních modelů v jazyce Haskell a kontrolu výstupů. Jinak řečeno kapitola shrnuje, co bude nutné nějakým způsobem v navrhovaném systému reprezentovat.

Jak je zřejmé z provedené rešerše, metody formální specifikace staví na matematických základech a hojně využívají abstrakcí z teorie množin a logiky. Matematika je v tomto směru univerzální a dovoluje modelovat i případy označované jako konceptuálně extravagantní, ale svým aparátem pro přesné vyjadřování a dokazování správnosti je vhodným nástrojem pro popis modelů. Současně výhodou je i to, že základní znalosti těchto oblastí matematiky by měl mít každý analytik a programátor. [6, 49]

Na matematických abstrakcích a zápisu staví například i logický programovací jazyk Prolog, ve kterém se program skládá z predikátů popisujících jednotlivé typy, jejich vlastnosti a vztahy. Pokud by bylo možné pouze definicí datových typů popsat jakékoliv vnitřní i vnější podmínky omezující prvky množiny jejich hodnot, pak by stačila definice těchto typů pro reprezentaci konceptů a současně jejich vztahů. Jazyk Haskell stejně jako další programovací jazyky takto detailní specifikaci typů neumožňují a je nutné pravidla specifikovat zvlášť. [49, 68]

5.1 Entity

Základem každého konceptuálního modelu jsou entity reprezentující koncepty. Koncepty mají své vlastnosti zachycené pomocí atributů (například datum narození, název, adresa, barva a další). Některý atribut, sada atributů i externí vlastnosti mohou instanci entity identifikovat, pokud je napříč všemi

instancemi unikátní a žádná další instance tudíž nemůže mít tuto hodnotu shodnout. Systém musí umožnit zachycení konceptů jako typů pro specifikaci instancí včetně vlastností. [3, 8, 75]

V rámci entit je vhodné umožnit vytvářet hierarchii a taxonomii pomocí speciálních relací generalizace a specializace i mírně odlišných pojmů nadtyp a podtyp. V takové hierarchii je nutné odlišit entity, které jsou čistě abstraktní a nemohou mít vlastní instance. Abstraktní entity složí pouze jako pojmenování skupiny entit, například dopravní prostředek, savec nebo hořlavina. Entita může logicky patřit do více takových skupin zároveň. Systém musí umožnit tvorbu hierarchických typových struktur pro zachycení úrovní abstrakce v konceptuálním modelu. [18, 75]

V matematickém zápisu lze entitu reprezentovat množinou všech možných instancí. Díky podmnožinám, průnikům a sjednocením lze vytvářet libovolné hierarchie, jak je vidět na následujícím jednoduchém příkladu. V příkladu tvoří množinu osob O dvě disjunktní podmnožiny fyzických O_F a právnických O_P osob, fyzické osoby mají pro jednoduchost atributy jméno j, příjmení p a město m z příslušných množin. Příkladem instance je Marek Suchánek z Prahy, který je fyzickou osobou.

$$O_F = \{(j,p,m)|j \in J, p \in P, m \in M\} \qquad O = O_F \cup O_P \qquad O_F \cap O_P = \emptyset$$

$$o_1 = (\text{"Marek"}, \text{"Suchánek"}, \text{"Praha"}) \in O_F \subseteq O$$

5.2 Vztahy

Mezi entitami lze definovat vztahy, jedná se matematicky řečeno o relace libovolné arity neboli podmnožinu kartézského součinu množin představujících participující entity. Relaci je možné popsat predikátem tak, že v relaci jsou elementy, pro které je predikát pravdivý. Vztahy mezi entitami lze zachytit různými způsoby, někdy dokonce vztah degraduje na pouhý atribut jiné entity. V systému kódování konceptuálních modelů je nutné umožnit jak použití entit jako atribut entity, tak i tvorbu samostatných obecně n-árních relací podobně jako je tomu v ER nebo ORM. Následující příklad ukazuje unární predikát Z říkající, zda osoba žije, a binární predikát S, pro reprezentaci vztahu, že osoba studuje na univerzitě. [3, 8, 18, 49]

$$Z(o_1) \qquad o_1 \in \{o|o \in O_F \wedge Z(o)\}$$
$$u_1 = (\text{ČVUT}, \text{FIT}, \dots) \qquad U = \{(n,f,\dots)|n \in N, f \in F, \dots\}$$
$$S(o_1,u_1) \qquad (o_1,u_1) \in \{(o,u)|o \in O_F \wedge u \in U \wedge S(o,u)\} \subseteq O_F \times U$$

Vztah může mít vlastní atributy, ale ty lze modelovat již jako další entitu participující v relaci. Příkladem takového vztahu může být záznam o studiu na

univerzitě obsahující identifikaci studenta, studijní obor, datum přijetí a další. Taková instance se bude podílet na relaci a tvořit tzv. *truthmaker* neboli pečetidlo vztahu. [3, 76]

$$S'(o_1, u_1, (\text{"suchama4"}, \text{"MI-WSI-ISM"}, (1, 10, 2015), \dots))$$

Pro zajištění možnosti tvorby aplikací přímo nad zapsaným konceptuálním modelem musí být umožněno se dotazovat nad jednotlivými instancemi entit, ve kterých relacích participují a s jakými instancemi. Tím bude současně umožněno procházení všech instancí, které na sebe navzájem mohou mít vliv právě skrze relace.

5.3 Omezení

V konceptuálním modelu stejně jako v reálném světě nejsou logicky možné všechny možnosti a kombinace instancí entit a vztahů. Z tohoto důvodu je nutné vytvořit omezení, které zpřesňují model a tím zmenšují prostor možných instancí modelu. Pokud se nějaká dvě libovolná omezení navzájem vylučují nebo některé omezení samo o sobě není nikdy splnitelné, pak neexistuje logicky žádná instance modelu, což indikuje pravděpodobnou chybu modelu. [8, 18, 49]

5.3.1 Omezení atributů entit

Pro samostatné entity lze definovat omezení přes jejich atributy. Mimo požadavku na unikátní hodnotu atributu či sady atributů je nutné umožnit specifikovat pravidla, jakých hodnot mohou atributy nabývat a za jakých podmínek. Omezení mohou být jednoduchá nad jedním atributem nebo i složitější nad kombinací atributů v rámci entity.

5.3.2 Omezení vztahů

Stejně jako pro entity, je nutné umožnit zachycení omezení vztahů. Tato omezení lze definovat jako vlastnosti participace entity ve vztahu, mezi které se řadí především kardinalita, povinnost, jedinečnost a závislost na hodnotách atributů. Další skupinou vlastností vztahů stanovující určitá omezení jsou matematické vlastnosti binárních relací (například symetrie, reflexivita, tranzitivita), které mají přesně specifikované a očekávané důsledky v modelu.

5.3.3 Omezení modelu

Mimo omezení entit a vztahů mohou existovat ještě globální omezení celého modelu. To však vyžaduje detailní dotazování nad modelem a jeho instancemi. Systém musí umožnit volnost v zachycení modelu a tvorby omezení na touto reprezentací. Mechanismus dotazování nad kompletním modelem musí být ponechán na vývojářích aplikace za účelem dosažení maximální flexibility.

5.4 Chování

Systém musí být navržen dostatečně obecný, aby bylo možné modelovat koncepty událostí a případně i jejich vlivů na jiné entity. Samotné chování v rámci zápisu procesů musí být opět flexibilní pro zvýšení použitelnosti a přehlednosti. Zakódovaný konceptuální model může obsahovat i zápis chování popisující změny instancí modelu, výpočty hodnot atributů a další ve formě zdrojového kódu využitelného napříč celou aplikací.

Chování lze matematicky zapisovat jako funkce, jejichž definičním oborem či alespoň jejich částí případně i oborem hodnot jsou množiny reprezentující entity a vztahy. Omezení jsou pak specializací takových funkcí říkajících o instanci, zda je validní. [49]

$$f : O \to O \qquad g : O \to \mathbb{N} \qquad h : S \to U \qquad c_i : O \to \mathbb{B}$$

5.5 Nefunkční požadavky

Mimo specifických požadavků na to, jaké prvky konceptuálních modelů musí být možné v rámci navrhovaného systému kódování zachycovat, vyvstávají další požadavky a cíle, které je nutné splnit, aby byl systém reálně použitelný. Tato skupina požadavků je označena jako nefunkční ve stejném smyslu jako je tomu u vývoje SW aplikací.

5.5.1 Univerzálnost

Navržený systém musí být dostatečně univerzální ve smyslu své expresivity, musí umožňovat zachycení různých úrovní abstrakce a libovolných entit, vztahů, omezení i chování. Tím, že se jedná o systém kódování konceptuálních modelů přímo ve zdrojovém kódu aplikace, a tudíž spojených s implementací, je nutné, aby model mohl zachytit maximální množství informací o modelované doméně potřebné v implementaci. Konstrukce programovacích jazyků umožňují zachytit v určité podobě rozličné struktury i chování, což by mělo být vyzdviženou výhodou přístupu zvoleného v navrhovaném systému.

5.5.2 Jednoduchost použití

Má-li být systém pro reprezentaci konceptuálních modelů úspěšný a široce užívaný, nesmí být použitelný jen experty na programování a matematiku, ale i pro méně zkušené programátory a analytiky. V tomto smyslu by měl být systém dostatečně intuitivní, využívat známé konvence a pojmenování. Chybové výpisy, vizualizace a dokumentace musí být vždy srozumitelné i bez detailní znalosti programování a matematických přístupů.

Nicméně pokročilý uživatel by neměl být limitován v použití složitějších konstrukcí za účelem dosažení vyšší efektivity nebo jiných vlastních cílů. Kupří-

kladu nelze od běžného uživatele očekávat znalost teorie kategorií a souvisejících konkrétních typových tříd, ale zkušenější uživatel, který je zná a běžně s nimi pracuje, má mít možnost je použít.

5.5.3 Rozšiřitelnost

Systém pro reprezentaci konceptuálních modelů by měl umožnit jednoduché rozšíření a úpravu pro specifické užití. Příkladem této vlastnosti jsou například profily UML, kdy je možné vzít podmnožinu jazyka UML, upravit ji a případně i přidat nové prvky. Touto kustomizací je možné zahrnout do jednoduššího metamodelu relativně složitější přístupy z vyšší či doménové ontologie nebo přímo z praxe.

V případě použití existující platformy pro reprezentaci by nemělo být znemožněno aplikování běžných postupů a rozšíření na této platformě. Tato práce se zabývá podporou konceptuálního modelování v jazyce Haskell, a tak by mělo být možné s takto vytvořeným konceptuálním modelem ve zdrojovém kódu dále pracovat a využít jej přímo v implementaci, a to včetně užití různých knihoven třetích stran.

5.5.4 Multiplatformní řešení

Systém musí být použitelný na libovolné běžné platformě pro osobní počítače (Linux, Mac a Windows). Omezení se na jednu určitou platformu by vedlo k výraznému limitování uživatelů, které je nepřípustné. Drobné odlišnosti v některých postupech při užití na různých platformách (instalace, výpisy, vizualizace a podobně) jsou možné, ale samotné konceptuální modely musí být volně přenositelné a platformě nezávislé.

5.6 Verifikace a validace

Implementace systému musí umožnit provádět verifikaci a validaci. Tím, že se jedná o systém kódování konceptuálních modelů přímo v programovacím jazyce a model tak bude přímo spustitelný je nutné pouze kontrolovat správnost zápisu modelu. U validace je potřeba připravit řešení pro generování instancí zakódovaného modelu a instance vhodně vypsat i vizualizovat. Uživatel pak bude mít možnost kontrolovat, které instance model připouští a zda to odpovídá modelované realitě.

Generování instancí však musí být jen doplňková funkcionalita a uživatel musí mít možnost manuálně upravovat generované instance i vytvářet zcela nové. Způsob generování instancí musí být vysoce konfigurovatelný tak, aby uživatel mohl generování přizpůsobit modelované problémové doméně i svým vlastním požadavkům.

5.7 Vizualizace

Pro účely konzultace nad modelem s netechnicky zaměřenými osobami je vhodné využít grafickou podobu modelu. Z tohoto důvodu musí pro navrhovaný systém být vytvořen vizualizační nástroj. Grafická podoba modelů slouží jak ke kontrole a diskuzi, tak i jako součást dokumentace, proto je požadavkem umožnění exportu do libovolných grafických rastrových i vektorových formátů.

Vzhledem k požadavkům na možnosti zápisu modelů, kterým jako poklad posloužily existující metody reprezentace konceptuálního modelování, je žádoucí se v grafické notaci opět inspirovat existujícími přístupy. Mimo vizualizace samotného zakódovaného konceptuálního modelu musí být možné vizualizovat i jeho instance a to jak generované v rámci validace, tak i vytvářené ručně a při provozu implementace modelu.

Vizualizace je formou transformace modelu z kódu v jazyce Haskell do jiného formátu. Způsob převodu musí být dobře zdokumentována a umožňovat tvorbu rozšíření do budoucna například pro překlad do jiných notací konceptuálního modelování. Cílem je tedy vytvořit jednoduchý ale snadno rozšiřitelný a znovupoužitelný nástroj.

Kapitola 6

Analýza jazyka Haskell z pohledu konceptuálního modelování

Kapitola se zabývá analýzou programovacího jazyka Haskell v rámci jeho využití jako prostředku pro tvorbu konceptuálních modelů, zjišťuje úroveň expresivity jazyka a hledá možné přístupy k naplnění stanovených požadavků na reprezentaci konceptuálních modelů včetně jejich následné verifikace, validace a vizualizace. Tato analýza je poté v další kapitole použita při návrhu vlastního systému kódování konceptuálních modelů jazyce Haskell včetně příslušné metodiky a implementaci prototypu.

6.1 Vývoj a vlastnosti jazyka Haskell

Haskell je čistě funkcionální programovací jazyk s nestriktním, nebo také líným, vyhodnocováním. Jednou z hlavních předností jazyka je silný statický typový systém založený na Hindley–Milnerově typovém rozšíření lambda kalkulu, který je postaven na typových třídách a parametrickém polymorfismu. Jako samotné funkcionální paradigma, tak i základy tohoto jazyka, vychází z matematického modelu známého jako lambda kalkul. [77, 78]

Haskell vzešel z akademického prostředí jako volně dostupný jazyk použitelný pro výuku, výzkum i vývoj komplexních systémů, který sjednocuje a rozvíjí myšlenky ostatních funkcionálních programovacích jazyků té doby (konec 80. let 20. století). V rámci vývoje jsou vydávány specifikace jazyka *Haskell Report*, kde mezi klíčové patří standardy 98 [77] a 2010 [78]. Postupem doby vznikla řada implementací jazyka v podobě kompilátorů, které často jazyk obohacují o vlastní rozšíření usnadňující programátorům vývoj aplikací. Nejpoužívanějším kompilátorem je v dnešní době GHC pocházející ze skotské University of Glasgow [79]. [80]

Neboť se funkcionální programování i samotný Haskell těší velké oblibě v akademické sféře, existuje množství výzkumných projektů experimentujících s jazykem a funkcionálním paradigmatem v různých podobách. Z oblasti konceptuálního modelování stojí v souvislosti s touto diplomovou prací za zmínku například projekt pro reprezentaci ontologií pomocí funkcionálních jazyků od J. S. Posthuma [81] a zápis ontologie pozorování v prostoru a čase přímo do jazyka Haskell od Wernera Kuhna [82].

6.2 Běžné konstrukce jazyka Haskell

V této části kapitoly jsou stručně popsány základní konstrukce jazyka Haskell ve standardu Haskell 2010 [78] a v jeho hlavní implementaci GHC, které mají vliv na expresivitu z pohledu konceptuálního modelování a je možné jich využít při návrhu vlastního systému pro kódování konceptuálních modelů.

6.2.1 Datové typy

Haskell disponuje mimo běžných datových typů známých z jiných jazyků také algebraickými datovými typy popsanými již v kapitole 4. Typ se deklaruje pomocí `data`, typového konstruktoru a souvisejících datových konstruktorů, které mohou mít libovolný počet parametrů. Jazyk Haskell vyžaduje, aby tyto konstruktory začínaly velkým písmenem, naopak typová proměnná je tvořena zpravidla jedním malým písmenem v případě generických typů. [70, 78]

GHC dovoluje vytvářet záznamové typy, jedná se o tzv. *syntactic sugar* pro součinový typ s definovanými funkcemi pro přístup k jednotlivým atributům, které slouží zároveň jako jejich pojmenování. To výrazně usnadňuje práci a zpřehledňuje zdrojový kód, ale přináší také několik omezení, která se řeší pomocí různých rozšíření, viz ukázka kódu 6.7. [70]

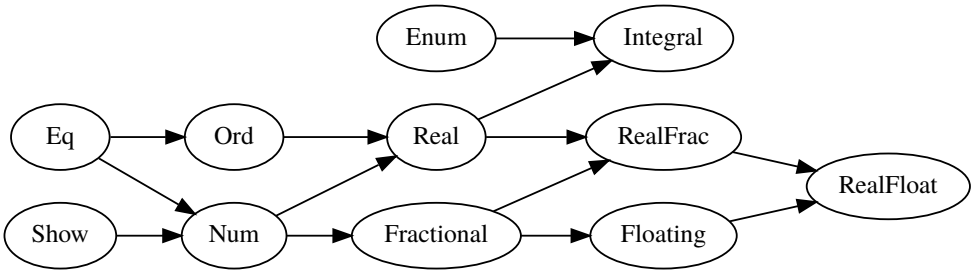
Konstrukce `newtype` funguje shodně jako `data`, ale umožňuje pouze jeden datový unární konstruktor a tím slouží k pouhému zapouzdření jiných datových typů, zpřehlednění kódu a tvorbě více instancí stejné typové třídy de facto nad zapouzdřeným typem. Použitím klíčového slova `type` je možné definovat typová synonyma bez přidaného datového konstruktoru, ale například s určením typových proměnných. Pomocí algebraických datových typů lze vytvářet celou hierarchii typů, a to včetně rekurzivních typů, viz `FamilyTree` nebo `List` pro seznam položek libovolného typu `a` v příkladu 6.1. [78]

6.2.2 Typové třídy

Typové třídy se často přirovnávají k rozhraním podobným těm z jazyku Java, protože definují, jak bude s typem, který je instancí typové třídy, možné pracovat. Typová třída slouží k polymorfismu ve formě, kdy funkce nebo i jiná konstrukce včetně typové třídy vyžaduje, aby parametr byl instancí nějaké typové třídy namísto konkrétního typu [78].

Kód 6.1: Příklad deklarace různých datových typů

```haskell
data Gender = Male | Female

newtype Age = Age Int  -- type renaming with new constructor

data Person = Person { name   :: String
                     , age    :: Age
                     , gender :: Gender
                     }
            | Unknown

data List a = Cons a (List a) | Nil  -- generic List type

type Descendants = List FamilyTree   -- type synonym

-- not record type ->    mother,father,their children
data FamilyTree = Couple Person Person Descendants
                | Single Person
```



Obrázek 6.1: Hierarchie typových tříd pro číselné typy (podle [70])

V Haskellu je předdefinována řada typových tříd a základní datové typy jsou instancemi některých z nich. Mezi důležité typové třídy patří mimo číselných tříd (`Num`, `Real`, `Fractional`, ...) a tříd pro různé struktury (`Monoid`, `Functor`, `Monad`, `Arrow`, ...) také třídy `Show` a `Read` deklarující funkce pro konverzi do a z řetězce symbolů typu `String`. Ve výchozí implementaci funkce `show` a `read` pracují s takovým řetězcem, který je shodný se zápisem instance ve zdrojovém kódu jazyka Haskell. [70, 78]

V rámci typových tříd lze vytvářet hierarchii tím, že lze vyžadovat, aby instance třídy byla instancí také některé jiné třídy nebo i několika jiných tříd, jak je znázorněno na obrázku 6.1. Podtřída pak může logicky využívat funkcí deklarovaných v nadtřídě, neboť instance podtřídy vždy musí být zároveň instancí dané nadtřídy. [78]

Kód 6.2: Příklad typové třídy a její instanciace (podle [70])

```haskell
class Quantifiable a where
  quantity :: a -> Int

data List a = Cons a (List a) | Nil

instance Quantifiable (List a) where
  quantity (Cons h t) = 1 + quantity t
  quantity Nil        = 0

data Picture = Picture { width  :: Int
                       , height :: Int
                       } deriving (Show, Read)

instance Quantifiable Picture where
  quantity pic = width pic * height pic

better     :: (Quantifiable a, Quantifiable b) => a -> b -> Bool
better x y = quantity x > quantity y
```

6.2.3 Čisté funkce a operátory

Haskell je funkcionální jazyk, a tudíž je funkce tzv. *first-class* objektem. Funkce vyššího řádu jsou funkce, které přijímají jednu či více funkcí jako parametr nebo funkci vrací jako výsledek. Pokud v rámci vykonání funkce není způsobován žádný vedlejší efekt jako třeba zápis do souboru, síťová komunikace, změna vnějšího kontextu, čtení vstupu a podobně, pak se funkce označuje jako čistá. Čisté funkce logicky při volání se stejnými parametry vždy vrací stejný výsledek, čímž je dosaženo referenční transparence a kompilátoru je umožněno s nimi provádět řadu optimalizací včetně paralelizace a cachování. [70, 78]

V jazyce Haskell je možné volání funkcí provádět prefixově i infixově. Operátory jsou infixové funkce definované s asociativitou a prioritou, v prefixu je možné je použít pomocí kulatých závorek (například: `(+) 5 7`). Naopak běžné binární funkce lze zapsat infixově s využitím zpětných jednoduchých uvozovek (například: `5 ‘add‘ 7`). [70, 78]

Základní funkce pro práci s ostatními funkcemi jsou základem Haskellu ve standardním modulu `Prelude`. Díky tomu je možné funkce skládat, invertovat, pozdržet vyhodnocení nebo například opakovat do splnění podmínky. Funkce lze částečně aplikovat a tím vytvářet nové funkce v rámci principu známého jako *currying*. Dále obsahuje i řadu funkcí pro práci s kolekcemi a dalšími datovými typy. Jelikož se jedná o čistě funkcionální jazyk neexistují imperativní konstrukce pro cykly a ty musí být zapsány pomocí rekurze. [70]

Kód 6.3: Příklad curryingu, složené funkce a operátoru

```
add3       :: Int -> Int -> Int -> Int
add3 a b c = a + b + c

add2 :: Int -> Int -> Int
add2 = add3 0

doubleLength :: [a] -> Int
doubleLength = (2*) . length

(<#>)      :: Int -> Int -> Int
(<#>) a b = (a * b) - (a + b)
infixr 5 <#>
```

6.2.4 Řídící konstrukce

Jako ve většině ostatních vyšších programovacích jazyků, tak i v Haskellu je možné provádět větvení pomocí konstrukce `if-else` a `case`. V jazyce Haskell však konstruují výraz a ten má vždy určený svůj výsledný typ. Navíc podobně jako v ostatních funkcionálních jazycích Haskell nabízí *pattern matching*, kterým lze efektivně a přehledně větvit kód na jednotlivé případy na základě předaných parametrů funkcí. [78]

Kód 6.4: Příklad větvení na různých zápisech výpočtu faktoriálu

```
fact1   :: Integer -> Integer
fact1 n = if n == 0 then 1 else n * fact1 (n - 1)

fact2   :: Integer -> Integer
fact2 0 = 1
fact2 n = n * fact2 (n - 1)

fact3   :: Integer -> Integer
fact3 n = case n of 0 -> 1
                    _ -> n * fact3 (n - 1)

fact4   :: Integer -> Integer
fact4 n
    | n == 0    = 1
    | otherwise = n * fact4 (n - 1)
```

6.2.5 Strukturování kódu

Zdrojový kód projektu napsaného v jazyce Haskell je vhodné členit do jednotlivých modulů, jejichž struktura kopíruje, opět podobně jako v jiných programovacích jazycích, hierarchii souborů a složek. Modul může definovat své veřejné funkce, třídy, typy a další konstrukce, které lze poté využít v rámci jiných modulů, skriptů nebo například v REPL. Při importu v rámci jiného modulu jazyk dovoluje explicitně stanovit, které části se mají importovat, nebo naopak, které se mají skrýt. Rovněž je umožněna tvorba aliasů a vynucení plně kvalifikovaných jmen. [70, 78]

Vstupním bodem programu je standardně `main` typu `IO ()`. Definovat funkce a výrazy je možné jak na úrovni celého modulu, tak je i zanořovat pomocí samovysvětlujících klíčových slov `where` a `let in`. Tím opět dochází k zpřehlednění zdrojového kódu a vytvoření principu lokality funkcí a výrazů. Inteligentní IDE pak umožňují s těmito konstrukcemi efektivně pracovat a zpříjemňují vývoj aplikací. [70, 78]

Kód 6.5: Příklad jednoduchého modulu

```haskell
module People (Person, mature) where

import qualified MyLib.MyModuleA (Age)
import qualified MyLib.MyModuleB hiding (Age) as MyB

data Person = Person { name :: MyB.Name
                     , age  :: MyLib.MyModuleA.Age
                     } deriving (Show, Read, Eq)

mature   :: Person -> Person
mature p = let isMature = age p > maturity in
             if isMature then p else p { age = maturity }
             where maturity = 18
```

6.2.6 DSL

Haskell je svou velmi přehlednou syntaxí vhodný pro tvorbu DSL, a to dokonce i pouze za použití základních konstrukcí jazyka, viz příklad 6.6. Mezi tyto využitelné konstrukce patří i funkce v infixovém zápisu, které při vhodném pojmenování umožňuje číst program jako jednoduché věty z přirozeného jazyka. Interní DSL lze dále snadno nadefinovat pomocí typových tříd, algebraických datových typů s využitím vlastností typových a datových konstruktorů. Poté už stačí jen definovat funkce převádějící zápis DSL na požadované akce.

Současně je možné vytvářet i externí DSL a pro vlastní jazyk relativně snadno v jazyku Haskell implementovat překladač. Tento způsob umožňuje větší míru volnosti při tvorbě syntaxe DSL, avšak již není tak jednoduše upravitelný a spravovatelný.

Kód 6.6: Užití vlastního DSL pro recepty na vaření

```
recipe :: Recipe
recipe = cooking Pancake $ do
  prepare Dough $ do
     add Milk 400 Milliliter
     add Sugar 1 Spoon
     add Salt 1 Pinch
     add Flour 200 Gram
     mixUp

  wait 30 Minute

  prepare Pancake $ do
    heatUp
    add Oil
    fillWith Dough
    bakeUntil Golden BothSides

  serve Pancake
```

6.3 Rozšíření a knihovny jazyka Haskell

Kolem jazyka Haskell se utvořila početná, a hlavně aktivní komunita, která vyvíjí různé knihovny a rozšíření jazyka. Obliba Haskellu v akademických kruzích se projevuje i v kvalitě a tématech některých rozšířeních, které by pravděpodobně v komerční sféře nikdy nevznikly. V této části kapitoly jsou zmíněna významná rozšíření a knihovny, které je možné v rámci konceptuálního modelování v jazyce Haskell využít.

Rozšíření implementace jazyka GHC nabízí řadu velmi pokročilých konstrukcí a většina je pro běžné uživatele složitá na pochopení i použití. Podobně je tomu tak i pro knihovny, na kterých lze pozorovat dvě skupiny vývojářů v Haskell komunitě. První skupinu tvoří čistí teoretici hledající vždy co nejvíce abstraktní a matematické řešení i za cenu nepřehlednosti a nepoužitelnosti pro méně zkušené uživatele. Na druhé straně jsou vývojáři s více praktickým přístupem, kteří spíše dbají právě na snadnou použitelnost i za cenu delšího a jednoduššího kódu. [70, 79]

6.3.1 Template Haskell

Template Haskell je rozšíření, které přidává Haskellu možnost metaprogramování. Široké možnosti Template Haskell dovolují programátorovi vytvářet nové konstrukce jazyka Haskell, které se v době překladu převedou na klasický Haskell s použitím abstraktního syntaktického stromu. V době kompilace je možné s konstrukty různě pracovat a transformovat je. Využití tohoto rozšíření však vyžaduje pokročilé znalosti jazyka a s použitím se váže řada omezení a možných chyb. [79]

6.3.2 Duplicate record fields

Od GHC verze 8 je možné psát duplicitní názvy v záznamových typech, což v předchozích verzích nebylo možné, neboť pojmenováním se vytvoří funkce pro získání atributu z instance typu. Tímto rozšířením sice dochází k zpřehlednění jmen atributů, ale již nenutí programátora k zamyšlený, zda stejný atribut neznačí společnou vlastnost různých typů, která by měla být řešena společně například v rámci kompozice nebo typových tříd. [79]

Kód 6.7: Příklad užití rozšíření pro záznamové typy

```
{-# LANGUAGE  DuplicateRecordFields #-}
{-# LANGUAGE  NamedFieldPuns #-}
{-# LANGUAGE  RecordWildCards #-}

data Address = Address { street :: String, no :: Int,
                         city :: String, country :: String }
data Pet     = Pet     { name :: String, age :: Int }
data Person = Person { name :: String, age :: Int,
                       home :: Address }

isMature :: Person -> Bool
-- isMature Person { age = age } = age >= 18
isMature Person { age } = age >= 18

addressBox :: Person -> String
-- addressBox Person { home = Address { street, ... } }
addressBox Person { home = Address { .. } } =
  street ++ " " ++ show no ++ nl ++ city ++ nl ++ country
  where nl = "\n"
```

6.3.3 Named field puns a Record wildcards

Opět jako předchozí i tato dvě rozšíření zpřehledňují práci s názvy atributů záznamových datových typů, a to při jejich použití ve výrazech. *Named field puns* dovoluje vynechávat deklarování lokálních proměnných ve funkcích a výrazech se shodným názvem atributu. *Record wildcards* umožňuje automatické přiřazení, neboli *binding*, atributů na lokální proměnné pomocí užití dvou teček, viz příklad 6.7. [79]

6.3.4 Generalised algebraic datatype

Generalizované algebraické datové typy jsou zobecněním parametrických algebraických datových typů, které dovoluje definovat datové typy pomocí deklarování typu jednotlivých datových konstruktorů. Obecněji řečeno se jedná o algebraické datové typy s jiným než standardním datovým konstruktorem. Využití tato možnost nalezne například při vytváření vlastních interních DSL v jazyce Haskell. [79]

6.3.5 LiquidHaskell

LiquidHaskell umožňuje pomocí anotací definovat predikáty, které se kontrolují v době překladu. Kontroluje se především, že funkce jsou úplné, výpočet není nekonečný, omezení množin typů a zajištění zachování specifikovaných vlastností. Dále je podporováno psaní formálních důkazů přímo v jazyce Haskell a spousta dalších pokročilých anotací, které jsou pro běžného uživatele již poměrně těžko uchopitelné a je vyžadována vyšší znalost matematiky, Haskellu i samotného LiquidHaskell. [83]

Kód 6.8: Příklad základních anotací LiquidHaskell

```
{-@ type Age = {y:Int | y > 0 && y < 120} @-}

{-@ validAge :: [Age] @-}
validAge = 18

{-@ invalidAge :: [Age] @-}
invalidAge = -5

{-@ increaseAge :: age:Age -> Age @-}
increaseAge age = if age < 100 then age + 50
                               else age + 10
```

6.3.6 Language.Haskell

Knihovna *haskell-src* [84] přináší modul `Language.Haskell` pro práci se samotným zdrojovým kódem v jazyce Haskell, umožňuje vybudovat nad jazykem vlastní jednoduchý kompilátor či interpret. Hlavní částí knihovny je popis syntaxe Haskellu a parser pro převod zdrojového kódu ve formě textu na abstraktní syntaktický strom. Se zdrojovým kódem v Haskellu převedeným do definovaných datových struktur lze poté libovolně manipulovat a opět převést zpět na nový zdrojový kód nebo vytvořit zcela jinou interpretaci.

6.4 Frameworky pro testování

Ve většině programovacích jazyků tvoří specifickou skupinu knihovny zaměřené na testování aplikací a není tomu jinak ani u jazyku Haskell. V rámci konceptuálního modelování je možné využít některého z těchto frameworků k testování vlastností modelu, generování jeho instancí a posléze i validaci modelu. Pro analýzu byly vybrány tři nejpoužívanější knihovny dle Haskell komunity. [70]

6.4.1 HUnit

Jak název napovídá HUnit je framework pro tzv. jednotkové testování inspirovaný nástrojem JUnit pro jazyk Java. Ve frameworku lze vytvářet jednotlivé testovací případy a z nich tvořit sady testů, kde základním kamenem testu je aserce. Výsledkem provedení testové procedury je pak čtveřice s počty testovacích případů, provedených testů, neočekávaných chyb a selhání testů. Práce s frameworkem je jednoduchá, ale nabízí i úpravu chování spouštění jednotlivých testů pomocí přetěžování, tvorby speciálních operátorů a dalších. Bohužel však nenabízí generování testovacích dat. [70]

Kód 6.9: Ukázka jednotkového testu pomocí HUnit

```
maturityTest1 = TestCase $
  assertEqual "mature 15" maturePerson (mature immaturePerson)
    where maturePerson   = Person { name = "Marek", age = 18 }
          immaturePerson = Person { name = "Marek", age = 15 }

maturityTest2 = TestCase $
  assertEqual "mature 24" olderPerson (mature olderPerson)
   where olderPerson     = Person { name = "Marek", age = 24 }

main :: IO ()
main = runTestTT $ TestList [maturityTest1, maturityTest2]
```

6.4.2 Hspec

Hspec je framework inspirovaný RSpec pro jazyk Ruby a slouží pro testování pomocí definice očekávaného chování v připraveném DSL, které připomíná přirozený jazyk a tím je čitelné i pro laiky. Navíc nabízí integraci s HUnit i QuickCheck, rychlejší provedení testů pomocí paralelizace a automatizované vyhledávání testů. Test se skládá z popisu chování určitého modulu či funkce, jak má pracovat s určitými vstupy, případně jak se má chovat v rámci posloupnosti výrazů. [70]

Kód 6.10: Ukázka Hspec specifikace pro testování

```
spec :: Spec
spec = do
  describe "Person" $ do
    it "mature person shouldn't change" $ do
      mature olderPerson `shouldBe` olderPerson

    it "immature person should mature" $ do
      mature immaturePerson `shouldBe` maturePerson
      where
        olderPerson    = Person { name = "Marek", age = 24 }
        maturePerson   = Person { name = "Marek", age = 18 }
        immaturePerson = Person { name = "Marek", age = 15 }

  describe "addition" $ do
  it "integer addition is comutative" $ property $
    \x, y = add x y == add y x

main :: IO ()
main = hspec spec
```

6.4.3 QuickCheck

QuickCheck testuje mírně odlišným způsobem než předešlé dvě knihovny, protože sám generuje testovací vstupy pro ověření definované vlastnosti a tím se snaží najít protipříklad. Test se tedy skládá pouze z definice očekávané vlastnosti (například asociativita funkce) a poté se automaticky ověřuje nastaveným počtem generovaných hodnot, přičemž se snaží nejprve testovat v jistém smyslu mezní hodnoty. Generátory hodnot je možné vytvářet vlastní a tím si vhodně upravit chování testů povaze testovaných funkcí a datových struktur. Samotné funkce pro generování instancí lze využít i mimo testování. [70]

Kód 6.11: Ukázka QuickCheck testu

```haskell
instance Arbitrary Person where
   arbitrary = do
     nameLength <- choose (1, 30)
     randomName <- replicateM nameLength (elements ['a'..'z'])
     randomAge  <- choose (0, 100)
     return Person { name = randomName
                   , age  = randomAge
                   }

prop_Mature       :: Person -> Bool
prop_Mature p
    | age p < 18  = mature p == p { age = 18 }
    | otherwise   = mature p == p

main :: IO ()
main = quickCheck prop_Mature
```

6.5 Zhodnocení expresivity a použitelnosti

Jak je vidno v této kapitole na různých příkladech, syntaxe, sémantika i filozofie jazyka Haskell se velice blíží některým jazykům formální specifikace popsaným v kapitole 2. Haskell nabízí díky silným matematickým kořenům i řadě rozšíření a knihoven výbornou úroveň expresivity nejen pro konceptuální modelování. Dále početná komunita a velké množství podpůrných nástrojů pro vývoj a další práci s jazykem usnadňuje jeho použití v rámci projektů s různými účely a parametry.

Kapitola 7

Systém pro kódování konceptuálních modelů

Tato část práce popisuje vlastní řešení systému pro kódování konceptuálních modelů v jazyce Haskell vzhledem k vlastnostem jazyka a definovaných požadavků v předchozích kapitolách. Specifikuje základní přístup systému a jeho obecné vlastnosti, současně jsou rozvedena naplnění jednotlivých požadavků na kódování prvků, ze kterých se konceptuální model skládá, i postupů verifikace, validace a vizualizace. Tato specifikace systému je v souladu s jeho implementovaným prototypem. Detailní popis všech modulů, typových tříd, datových typů, funkcí a dalších částí prototypu systému je detailně popsán v jeho dokumentaci.

V popisu systému se vyskytuje užití označení „uživatel“ a „analytik“, která jsou v tomto případě zaměnitelná. Uživatelem systému je primárně analytik, který model vytváří, získává z něj informace a dále s ním pracuje.

7.1 Reprezentace konceptuálních modelů

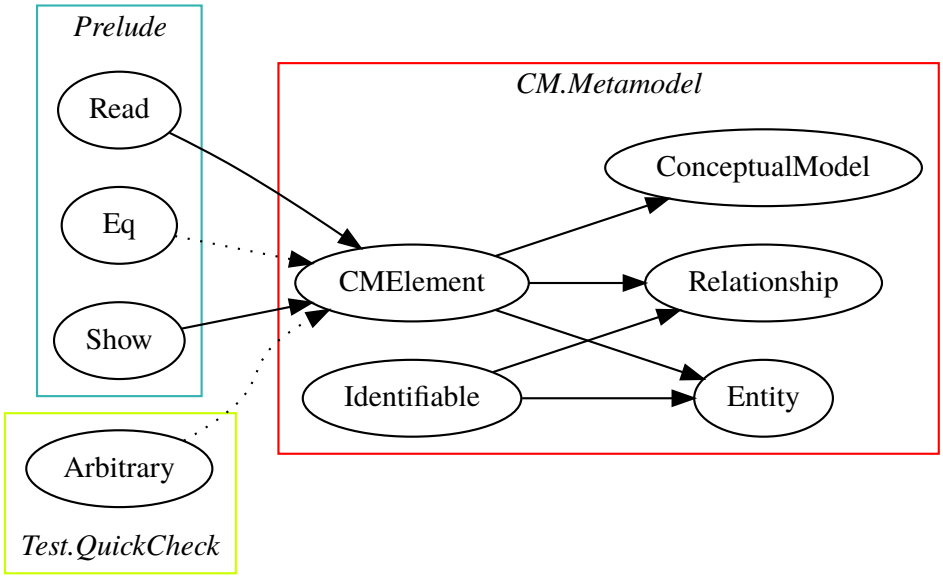
Systém je navržen jako univerzální knihovna *hCM* jazyka Haskell, tudíž ji lze libovolně využívat jak v desktopových, tak i webových aplikacích. Dále staví pouze na konstrukcích jazyka Haskell tak, aby nebyly při zápisu konceptuálních modelů vyžadovány další znalosti. V tomto smyslu se systém kódování snaží být co nejjednodušší a tím i nejpoužitelnější, současně ale také volně rozšířitelný a umožňující použití dalších knihoven pro detailnější specifikaci (například pomocí LiquidHaskell), pokud to uživatel potřebuje.

7.1.1 Typové třídy

Základem systému je soustava typových tříd specifikujících očekávané vlastnosti jednotlivých prvků, ze kterých jsou konceptuální modely z pohledu systému složeny. Tyto třídy vychází z analýzy požadavků založených na rešerši

základů konceptuálního modelování a existujících přístupů. Některé vlastnosti jsou dodány již s výchozí implementací, aby došlo ke zjednodušení tvorby konceptuální modelu. Zároveň je však umožněno přepsání této výchozí implementace v případě potřeby.

V systému jsou využity i základní typové třídy Haskellu `Show` a `Read` pro usnadnění práce s konceptuálním modelem. Výchozí implementace funkcí `show` a `read` umožňuje vypisovat a načítat instance v přehledné formě zdrojového kódu, tím lze instance částí i celého konceptuálního modelu serializovat a ukládat pro pozdější použití. Je doporučeno použití typové třídy `Eq` a pro účely validace i třídy `Arbitrary` frameworku QuickCheck, viz diagram 7.1.



Obrázek 7.1: Hierarchie typových tříd prototypu systému *hCM*

Problematika identity je řešena za pomoci typové třídy `Identifiable` s funkcí `identifier`. Díky tomu lze specifikovat, za jakých podmínek jsou dvě instance entity shodné (`identic`), například dvě osoby na základě rodného čísla. Tím je možné současně využívat i základní třídu `Eq` s operací `==`, jejíž význam je z konceptuálního hlediska v systému mírně odlišný a slouží k porovnání na základě všech atributů, stejně tomu je ve výchozí implementaci. Dvě instance osoby mohou představovat identickou osobu, ale například v rámci času se nacházet v odlišném stavu.

Předepsané rozhraní jednotlivých typových tříd lze nalézt v dokumentaci prototypu systému, ale i za pomoci REPL GHCi, který navíc také umožňuje snadné zjištění, které datové typy jsou instancemi vybrané typové třídy.

7.1.2 Konceptuální model

V navrženém systému je konceptuální model trojicí (E, R, C), kde

- E je množina entit s vlastními omezeními,
- R je množina vztahů mezi entitami s vlastními omezeními,
- C je množina funkcí definujících omezení pro celý model.

Přesně takto je konceptuální model také deklarován pomocí typové třídy `ConceptualModel` a samotná implementace je přenechána na uživateli systému. Díky tomu má naprostou volnost, jak zvolí, aby byl model konstruován. Může být složen z více pomocných datových typů nebo přímo sdružovat potřebné množiny v sobě. Dále je možné vlastním typem modelu obsáhnout další množiny nad rámec požadavků typové třídy nebo naopak provést zjednodušení, pokud nejsou žádná omezení celého modelu.

7.1.3 Element

Základní typovou třídou je `CMElement` pro reprezentaci očekávaného rozhraní každého prvku konceptuálního modelu zakódovaného pomocí systému. Každý prvek musí mít svá vlastní omezení, která ve výchozím stavu představují prázdnou množinu. Rozhraní umožňuje snadné vyhodnocování omezení za účelem zjištění přípustnosti daného prvku v modelu. Dále prvek nese své jméno, kterým se identifikuje jako v modelu, a funkci pro převod sebe sama do vyšší úrovně abstrakce, viz metamodel dále.

7.1.4 Entity

Prvním typem prvku konceptuálního modelu je entita, zachycena pomocí třídy `Entity`, kterou zakódované entity musí instanciovat. Entita je vždy zároveň identifikovatelná a uživatel musí specifikovat její identifikátor. Dále předepisuje, že entita má atributy, které mají svůj název, typ a hodnotu. Uživatel systému má volnost v přejmenování atributů a typů, jejich vyčlenění z modelu, definici nadtypů i podtypů a podobně, musí jen dodržet předepsané rozhraní, což však hlídá kompilátor.

Pro zakódování entit konceptuálních modelů je doporučena definice záznamového datového typu, což v jazyce Haskell umožňuje díky zmíněným vlastnostem algebraických datových typů efektivně zachycovat i složitou hierarchii a taxonomii. Tento přístup přímo naplňuje požadavky, jedinou nevýhodou je nemožnost stejného pojmenování atributů entit, ale to je řešeno pomocí GHC rozšíření od verze 8. Implementace funkce pro získání informací o atributech je poté triviální. Navíc je možné si atributy libovolně přejmenovat i včetně, pro Haskell nepovolených, identifikátorů obsahujících mezery, speciální symboly nebo začínající velkým písmenem.

Kód 7.1: Ukázka instanciace tříd pro entitu

```
data Student = Student { stUsername :: String
                       , stFirstname :: String
                       , stLastname :: String
                       }
             deriving (Show, Read)

instance Identifiable Student where
  identifier = stUsername

instance CMElement Student where
  toMeta = toMetaEntity

instance Entity Student where
  entityAttributes Student {..} =
    map tupleToAttribute
      [ ("username", "String", stUsername)
      , ("firstname", "String", stFirstname)
      , ("lastname", "String", stLastname)
      , ("email", "String", stUsername ++ "@fit.cvut.cz")
      ]
```

Při kódování systému se neklade žádné omezení na použití existujících typů v jazyce Haskell. Může být využito jak různých kolekcí, tak i například typů `Maybe` a `Either` pro zachycení volitelnosti a exkluzivity. Dále lze rovněž volně využít vlastních či knihovních typových tříd, například lze ze své vlastní entity učinit instanci třídy `Monoid`.

Entita může být realizována i složitější strukturou s pomocnými atributy, které se do konceptuálního modelu nepromítnou, díky zvolenému rozhraní. To může být velmi výhodné pro práci s různými knihovnami pro persistenci dat, webovými frameworky a dalšími. Samozřejmě tato volnost je vykoupena nutností specifikace detailů, které jsou při běžné implementaci záznamovými typy již přímo v kódu.

7.1.5 Vztahy

Podobně jako entita jsou zakódovány v systému i vztahy mezi nimi. Vztah je opět reprezentován libovolným datovým typem, který je instancí typové třídy `Relationship`. Jediným rozdílem je nahrazení atributů participacemi entit ve vztahu, což je vlastně speciální atribut vztahu. Shodně jako v celém systému je umožněna při modelování absolutní volnost a lze vytvářet vztahy libovolné

arity. Identita instancí relací se definuje ve výchozí implementaci jako identita všech jejich participantů.

Pokud by entity obsahovaly relaci jako atribut, pak by logicky vznikaly v instanci modelu smyčky, což je pro účely generování a výpisu nežádoucí. V rámci práce se samotnou entitou je však někdy nutné mít k dispozici vztahy, kterých se účastní, například za účelem svého chování nebo tvorby omezení. To je však již specializací toho, že entita si je vědomá okolního modelu. Při práci s entitou tak může být požadováno předání kontextu ve formě celé instance modelu, díky které je mimo jiné možné zjistit, v jakých vztazích a s jakými ostatními entitami participuje.

Kód 7.2: Ukázka instanciace tříd pro vztah

```
data IsEnrolled = IsEnrolled { ieWho :: Student
                             , ieWhat :: Course
                             , ieTruth :: Enrollment
                             }
                deriving (Show, Read)

instance Identifiable IsEnrolled

instance CMElement IsEnrolled where
 toMeta = toMetaRelationship

instance Relationship IsEnrolled where
  relationshipName _ = "enrolled"
  relationshipParticipations IsEnrolled {..} =
    map tupleToParticipation
      [ ("who", "Student", identifier ieWho,
         Optional Unlimited)
      , ("what", "Course", identifier ieWhat,
         Optional Unlimited)
      , ("truthmaker", "Enrollment", identifier ieTruth,
         Mandatory Unique)
      ]
```

Instance vztahu musí mít vždy všechny své participanty. Potřeba volitelných participantů značí, že se v modelu pravděpodobně slučují případy, které jsou z konceptuálního hlediska neslučitelné. Nicméně toto omezení lze obejít pomocí definice participantů s kolekcemi nebo typu `Maybe`. Kardinalita vztahu z pohledu jednotlivých participantů se definuje pomocí předpřipraveného typu `ParticipationType`, kde lze definovat libovolný typ vztahu (povinný unikátní, volitelný, zcela libovolný m:n).

7.1.6 Omezení

Omezení jsou zachycena jako čisté funkce použitelné pro filtrování struktur entit a vztahů. V systému se vyskytují však dva typy těchto funkcí. První a jednodušší pracuje pouze nad vlastním elementem konceptuálního modelu bez znalosti kontextu ve formě modelu, do kterého patří. Druhý a složitější navíc bere za parametr také instanci celého model obsahující daný element, kterou může vlastní omezení využívat pro konstrukci komplexnějších podmínek a závislostí.

Samozřejmě druhý způsob zobecňuje první, a proto je možné jednodušší omezení převést na složitější s tím, že se parametr ignoruje. Naopak v rámci curryingu lze složitější omezení dodáním prvního argumentu převést na jednodušší. Oba tyto typy popisuje rozhraní třídy `CMElement` a poskytuje funkce pro převod a vyhodnocování.

Kód 7.3: Ukázka dvou typů omezení

```
sinceBeforeUntil :: Enrollment -> Validity
sinceBeforeUntil Enrollment {enrlUntil = Nothing, ..} = Valid
sinceBeforeUntil Enrollment {enrlSince = a, enrlUntil = Just b} =
  newConstraint (a < b) "Since is not before Until"

instance CMElement Enrollment where
  toMeta = toMetaEntity
  simpleConstraints = [sinceBeforeUntil]

max2Enrollments :: (ConceptualModel m) =>
                   m -> IsEnrolled -> Validity
max2Enrollments model r =
  newConstraint (same <= 2) "Student can enroll course only twice"
  where same = length . filter areSame $ enrolledRelationships
        areSame e = (sameCourse e) && (sameStudent e)
        -- ...

instance CMElement IsEnrolled where
  toMeta = toMetaRelationship
  complexConstaints = [max2Enrollments]
```

Vytváření takových funkcí vyžaduje pouze základní znalosti programování a jako celý jazyk Haskell velmi připomíná matematický zápis. Pomocí typových tříd lze tyto funkce deklarovat genericky a jejich implementace i použití je plně na uživateli systému. Ve výchozí implementaci funkcí jsou entity i vztahy zcela bez omezení, tudíž pro jednodušší modely se s nimi analytik nemusí vůbec zabývat a explicitně stanovovat, že žádná omezení nejsou.

7.1.7 Chování

Chování lze v systému zachytit pomocí čistých funkcí, které nějakým způsobem operují nad entitami a jejich vztahy nebo i celým modelem. Každou funkci, jejíž typ parametru nebo návratový typ je instancí třídy `Entity`, `Relationship` anebo `ConceptualModel`, lze chápat jako chování v rámci konceptuálního modelu. Při modelování chování ve formě funkce či sady funkcí lze použít všech konstrukcí jazyka Haskell i dalších knihoven a rozšíření.

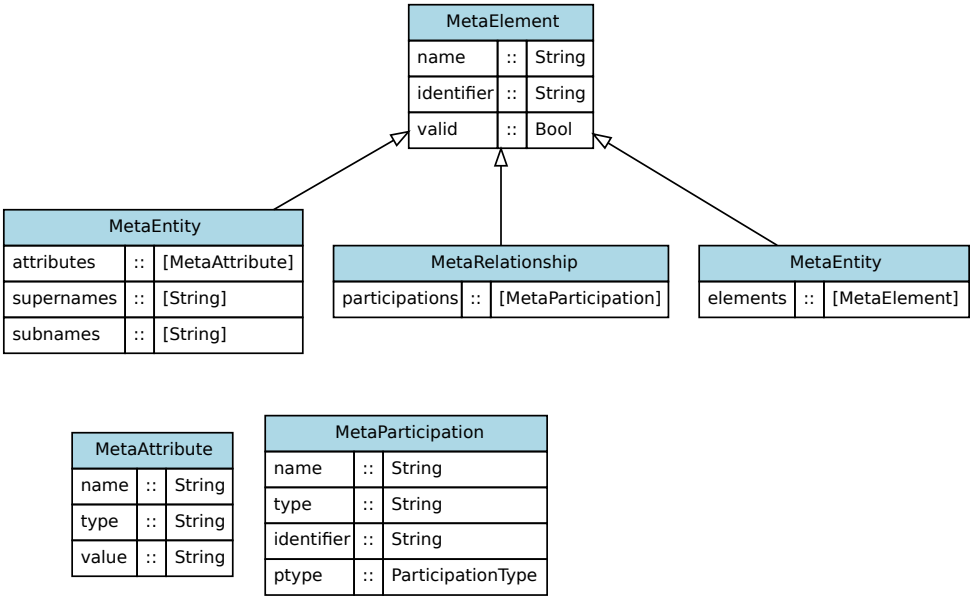
Na chování nejsou z principu kladeny vůbec žádné požadavky a jejich modelování není systémem ovlivňováno. Nicméně v jazyku Haskell lze chování z hlediska funkcionálního paradigmatu dobře modelovat. Volnost je zachována i tím, že chování se nijak nezačleňuje do instance třídy `ConceptualModel` a ani se nepromítá do datových typů metamodelu. Vliv namodelovaného chování lze pak jednoduše testovat a v rámci ověřování zjišťovat, zda instance zakódovaného konceptuálního modelu je validní i po provedení dané akce.

7.2 Metamodel

Pro snadnou práci se zapsaným konceptuálním modelem i lepší porozumění filozofie a fungování samotného systému slouží vlastní reprezentantce metamodelu ve formě datových typů. Tyto datové typy popisují vlastní strukturu částí systému: entit, vztahů a modelů. Samy datové typy jsou také entitami tvořící model, neboť metamodel je díky vysoké úrovni abstrakce typových tříd systému rovněž modelem. Každá entita, vztah a model by měla být pomocí chování specifikovaného typovými třídami převeditelná na svůj typ z metamodelu, jehož je de facto instancí skrze typové třídy.

Tento přístup umožňuje pracovat s modelem na různých úrovních a pohodlněji jej transformovat například v rámci vizualizace. Analytik má možnost při implementaci funkce `toMeta` využít dle doporučení předdefinovaných funkcí `toMetaRelationship`, `toMetaEntity` a `toMetaModel` z příslušných typových tříd, ale také volně měnit popis daných entit a vztahů. Příkladem jsou omezení Haskellu na názvy atributů a datových typů, při převodu do meta úrovně může být specifikován vlastní alias atributu jako libovolný textový řetězec. Stejně tak lze učinit i s názvy datových typů a výpisem hodnot atributů. Tím se může například vizualizace stát ještě čitelnější pro zákazníka.

Otázkou může být, proč nestačí k tomuto popisu pouze typové třídy a čím tento převod na datový typ přináší výhody. Jedním z argumentů je skutečnost, že v jazyce Haskell nelze vytvářet kolekce instancí typové třídy jako je tomu například u objektově orientovaných jazyků. Uživatel systému by musel tudíž implementovat mechanismy pro aplikování různých funkcí na všechny prvky modelu sám, ale takto stačí vytvořit převod, u kterého může navíc provést i zmíněná vylepšení popisu kódovaného konceptuálního modelu.



Obrázek 7.2: Zjednodušený diagram metamodelu prototypu systému *hCM*

7.3 Verifikace a validace

Systém používá pro verifikaci a validaci pouze kompilátor GHC a testovací framework QuickCheck. Dále je však možné použít jakékoliv další nástroje podle potřeby například specifikovat detailněji typy pomocí LiquidHaskell a ty poté kontrolovat, využít jiný testovací framework, použít LiquidHaskell jen pro základní kontrolu úplností funkcí a podobně.

7.3.1 Kompilátorem řízené modelování

Jednou z výhod zvoleného řešení kódování modelů a jeho částí je vedení uživatele systému procesem tvorby konceptuálního modelu pomocí kompilátoru. Po označení struktury za konceptuální model pomocí instanciace třídy `ConceptualModel` navede kompilátor uživatele k definování povinných deklarovaných funkcí ve třídě. Především se jedná o funkci pro získání elementů modelu, tím uživatel zjistí, že musí vytvořit tyto entity, resp. vztahy. Kompilátor zkontroluje, zda jsou tyto části instancemi potřebných tříd. Shodně po jejich označení za instance třídy `Entity`, resp. `Relationship`, kompilátor hlídá definici potřebných funkcí.

Celý proces modelování je tak řízen kompilátorem, uživateli stačí začít se třídou `ConceptualModel` a poté již dostává vždy hlášení, co musí být instancí té které třídy a jaké funkce musí definovat. Protože velká část funkcí tříd je již definována ve výchozí implementaci, kompilátor vyžaduje jen nutné minimum. Nicméně pokud uživatel zvolí vlastní definice, pak je kompilátor samozřejmě

zkontroluje a ověří správnost jejich zápisu. Tato užitečná vlastnost navrženého systému je po vzoru MDD v této práci označena jako *kompilátorem řízené modelování*, v angličtině zkráceně CDM.

Jak již bylo zmíněno, kompilátor GHC disponuje i prostředím REPL a debuggerem, které lze efektivně využívat pro procházení modelu i samotného systému, zjišťování informací o typových třídách, jejich funkcích a instancích, stejně jako vytvářet modely a dotazovat se nad nimi. To vše dokládá, jaký zásadní vliv kompilátor má v navrženém systému.

Kód 7.4: Chyby při kompilaci vynucující kódování pomocí tříd systému

```
/home/.../Example.hs:133:28: error:
    • No instance for (CMElement Course) arising from a use of
    ↪  ‘toMeta’

/home/.../Example.hs:45:10: warning: [-Wmissing-methods]
    • No explicit implementation for
        ‘toMeta’
    • In the instance declaration for ‘CMElement Course’

/home/.../Example.hs:46:10: warning: [-Wmissing-methods]
    • No explicit implementation for
        ‘entityAttributes’
    • In the instance declaration for ‘Entity Course’
```

7.3.2 Ověření správnosti modelu

Díky použití základních konstrukcí, které umožňuje jazyk Haskell, lze celý model v rámci verifikace a ověření správnosti kontrolovat pomocí kompilátoru. Zároveň není žádné oddělení implementace od konceptuálního modelu a tím je zajištěno, že se aplikace bude chovat přesně podle namodelovaných specifikací. Kompilátor obstarává kontrolu syntaxe, použití správných typů při definovaných typových omezeních na třídy, různých typů konfliktů a kolizí v názvech, implementaci rozhraní typových tříd a další. Chybové hlášení obsahuje přesné určení místa chyby s odůvodněním a ve většině případů je vypsána i možná příčina včetně návrhu možné opravy.

Haskell komunita vyvíjí různá rozšíření pro textové editory, některá z nich umožňují velkou část těchto kontrol provádět přímo při psaní zdrojového kódu. To je také jedním z důvodu pro využití ověřování pomocí rozšířeného kompilátoru GHC. Další testování vlastností entit, vztahů, chování a celého modelu může uživateli posloužit přímo testovací framework QuickCheck nebo Hspec, do kterých díky procesu validace jsou již připraveny generátory instancí.

Kód 7.5: Chyby formální správnost zápisu při kompilaci

```
/home/.../Example.hs:122:36: error: parse error on input ‘::’

/home/.../Example.hs:104:38: error:
    • Couldn't match type ‘Maybe [Char]’ with ‘[Char]’
      Expected type: String
        Actual type: Maybe [Char]
    • ...
```

7.3.3 Kontrola specifikovaných omezení

Vytvořená omezení pro entity, vztahy a celé instance modelu slouží ke kontrole přípustnosti instance dané konstrukce pomocí funkce `valid` z typové třídy `CMElement`. Rovněž je možné se dotázat na seznam chyb funkcí `violations`. Systém nic dalšího pro toto jednoduché fungování již nevyžaduje.

Kód 7.6: Ukázka vyhodnocování omezení v REPL

```
GHCi> evalConstraints Example.model Example.courseRRI
[Invalid "Negative credits"]
GHCi> valid Example.model Example.courseRRI
False
GHCi> violations Example.model Example.courseRRI
["Negative credits"]
GHCi> evalConstraints Example.model Example.courseDIP
[Valid]
GHCi> valid Example.model Example.courseDIP
True
GHCi> violations Example.model Example.courseDIP
[]
GHCi> validModel Example.model
False
```

7.3.4 Generování instancí modelu

Pro účely validace systém poskytuje možnost generování instancí entit, vztahů a tím i celého modelu. Základem je typová třída `Test.QuickCheck.Arbitrary` z testovacího frameworku *QuickCheck*. Pro každou entitu, vztah a celý model se vyžaduje za účelem automatizovaného generování instancí implementovat funkci `arbitrary`, která vytváří generátor pseudonáhodných instancí typu `Test.QuickCheck.Gen`.

Implementace takové funkce je v případě záznamových datových typů i s tvorbou hierarchie relativně jednoduchá. Umožňuje definovat rozsahy hodnot, kterých mohou jednotlivé atributy nabývat, nebo využít výchozích implementací funkce `arbitrary` na základních či již vlastních definovaných datových typech.

Kód 7.7: Ukázka předpisu pro generování instancí entity

```
companyNames = ["Duff", "Channel 5", "Kwik-E-Mart"]
firstNames =  ["Meg", "Marge", "Bart", "Lois", "Homer",
               "Peter", "Brian", "Chris", "Lisa"]
lastNames =  ["Simpson", "Griffin"]

instance Arbitrary Subject where
    arbitrary = do
        subjectType <- elements ["Company", "Person"]
        case subjectType of
            "Company" -> do
                randomName <- elements companyNames
                return $ Company { name = randomName }
            otherwise -> do
                randomFName <- elements firstNames
                randomLName <- elements lastNames
                return $ Person { firstname = randomFName
                                , lastname = randomLName }
```

Pro snazší použití jsou v následující případové studii připraveny generátory jmen, adres a dat, které zároveň slouží jako příklad, jak vytvářet vlastní generátory a jak je použít. Analytik připravující konceptuální model pro validaci se zákazníkem by měl při návrhu generování zohlednit řadu faktů, které nelze aplikovat v rámci obecného systému pro konceptuální modelování. Současně mu zůstává možnost vytvářet instance modelu manuálně či si je libovolně upravovat a ukládat díky typovým třídám `Show` a `Read`.

Další výhodou zvoleného přístupu pomocí frameworku QuickCheck je, že připravené generátory instancí se využijí pro testování případné implementace aplikace nad konceptuálním modelem. Jak bylo zmíněno v předchozí kapitole, tak pro takové testování je možné využít i framework Hspec, který integruje jak HUnit, tak QuickCheck. Testování je nedílnou a velmi užitečnou součástí vývoje aplikací a při tvorbě Haskell balíčků nástrojem `stack` je již připraveno v rámci šablon. [70]

Při použití rozšíření *DeriveGeneric* je možné třídu `Arbitrary` derivovat podobně jako například třídu `Show`, to však není doporučeno, neboť rozsahy hodnot nereflektují problémovou doménu a jsou zcela pseudonáhodné. Nicméně

pro pouhé testování libovolných hodnot takový postup může být přínosný, neboť takové instance zákazník ani analytik nemusí zkoumat, pokud jsou v modelu vhodně ošetřeny.

Kód 7.8: Ukázka generování instancí entity v REPL z příkladu 7.7

```
GHCi> subjectGen = arbitrary :: Gen Subject
GHCi> :type sample
sample :: Show a => Gen a -> IO ()
GHCi> sample subjectGen
Company {name = "Kwik-E-Mart"}
Person {firstname = "Bart", lastname = "Simpson"}
Person {firstname = "Chris", lastname = "Griffin"}
Company {name = "Channel 5"}
Company {name = "Duff"}
Person {firstname = "Lisa", lastname = "Simpson"}
Person {firstname = "Chris", lastname = "Griffin"}
...
GHCi> personGen = suchThat subjectGen isPerson
GHCi> :type personGen
personGen :: Gen Subject
GHCi> :type isPerson
isPerson :: Subject -> Bool
GHCi> sample personGen
Person {firstname = "Bart", lastname = "Griffin"}
Person {firstname = "Meg", lastname = "Griffin"}
Person {firstname = "Meg", lastname = "Simpson"}
...
```

Jak je dobře vidět na příkladu 7.8, s generátory instancí typu `Gen` lze pohodlně pracovat nejen v kódu, ale i v REPL. Framework QuickCheck poskytuje různé funkce pro generování instancí podle požadavků. Důležitou je, v příkladu demonstrovaná, funkce `suchThat`, která konstruuje ze stávajícího generátoru nový, který generuje instance splňující určitou podmínku. Takovou funkci je vhodné použít pro generování validních nebo naopak nevalidních instancí entit, vztahů a celých modelů s použitím funkce `valid`, která je definována v typové třídě `CMElement`.

7.4 Vizualizace

Jedním z požadavků i součástí zadání práce byla možnost vizualizace, která plní důležitou roli především při konzultaci a validaci konceptuálního modelu se zákazníkem, porovnávání modelů a jeho instancí i pro přiložení do tech-

nické dokumentace. Ačkoliv je zdrojový kód v jazyce Haskell sám o sobě velmi přehledný, pro zákazníka by tato forma nemusela být srozumitelná a grafická notace vede častěji k lepšímu porozumění [29].

7.4.1 Forma vizualizace

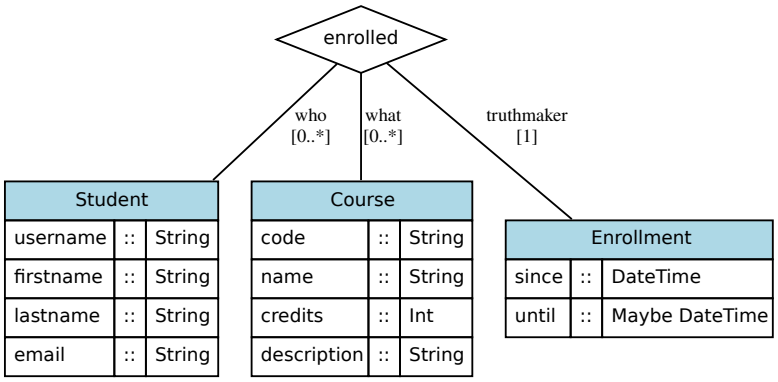
Pro vizualizaci konceptuálních modelů zakódovaných v jazyce Haskell bylo zvoleno využití jazyka DOT, se kterým mimo jiných pracuje i známý nástroj Graphviz [85]. Výhodou tohoto přístupu, oproti vlastní implementaci zobrazování ve formě aplikace s GUI, je jednoduché a známé použití, snadná rozšiřitelnost, přehlednost kódu knihovny i výsledku, export do různých grafických formátů a umožnění libovolných úprav na úrovni DOT i po exportu.

Podoba vizualizace byla zvolena tak, aby se co nejvíce podobala notaci ER/EER a UML, neboť se v systému shodně modelují entity a jejich vztahy. Atributy jsou zobrazeny přímo v rámci entit, podobně jako je tomu v UML, a nikoliv přes vztah *has*. Hierarchické relace *is a* jsou odlišeny speciální šipkou rovněž podle UML.

Přes své nesporné výhody má DOT do jisté míry omezené možnosti. Disponuje velkým počtem různých stylů uzlů a hran, voleb automatického uspořádání, seskupování a dalších nastavení [85]. Pro znázornění entit a jejich instancí byla využita forma HTML tabulek, u kterých však nelze aplikovat pokročilé stylování, a tak například musí mít všechna ohraničení. Nicméně všude, kde to bylo možné, byl dodržen styl podobný UML.

7.4.2 Vizualizace modelů

Konceptuální model je v systému zachycen na úrovni zdrojového kódu a jeho vizualizace je transformací tohoto z jazyka Haskell do jazyka DOT. Při transformaci jsou však z kódu vybrány jen definované konstrukce systémem pro kódování konceptuálních modelů a ostatní jsou ignorovány.



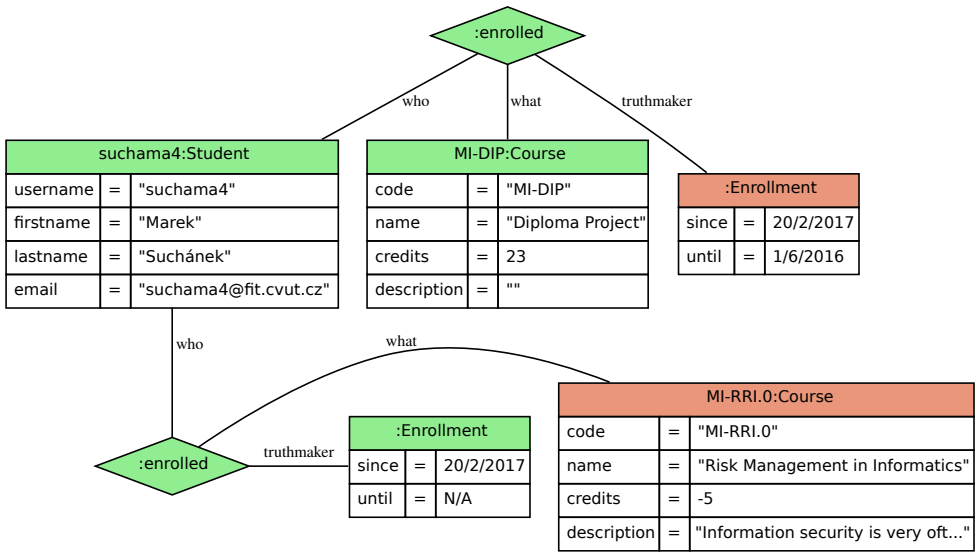
Obrázek 7.3: Ukázka vizualizace modelu

Vytváření vizualizace konceptuálního modelu probíhá na základě jeho instance. Systém totiž umožňuje jednoduché vytvoření prázdných instancí v metamodelu právě za účelem zkoumání struktury konceptuálního modelu. Tím lze vhodně upravit rozsah vizualizovaného modelu a vybrat jen jeho určitou část již v Haskellu. Při vytváření systému vizualizace byl zvážen i přístup získávání informací čtením zdrojového kódu a zpracování pomocí *haskell-src* nebo pomocné aplikace v jiném jazyce, ale ten by byl poměrně složitý, neefektivní a přinášející řadu omezení použitelnosti a flexibility samotného postupu kódování modelů.

7.4.3 Vizualizace instancí

Instance konceptuálního modelu se vytváří a existují v běhovém prostředí aplikace a jejich vizualizace je tak pouhou serializací do určitého tvaru. Nástroj nevizualizuje však pouze jednotlivé instance entit a vztahů, ale také skutečnost, zda jsou dané dílčí instance stejně jako a tím i instance celého konceptuálního modelu validní.

Podoba vizualizace instance odpovídá podobě vizualizace celého konceptuálního modelu, viz obrázek 7.4. Instance mohou být obrovské a jejich procházení komplikované. Prací analytika je vytvářet instance tak, aby bylo reálné je procházet, a přesto pozorovat různé závislosti. Díky zvolenému formátu je také velice jednoduché z vizualizace vybrat pouze část se zajímavým případem z pohledu validace a zbytek odstranit. Takové úpravy lze provádět na všech úrovních: ve zdrojovém kódu, ve vygenerovaném kódu DOT i ve výsledném souboru obrázku.



Obrázek 7.4: Ukázka vizualizace instance modelu z obrázku 7.3

7.5 Validace modelu se zákazníkem

Metodika validace se zákazníkem je díky podpoře vizualizace a generování instancí modelu velmi jednoduchá a skládá se z těchto kroků:

1. Analytik společně se zákazníkem vytvoří konceptuální model problémové domény zachycující potřebnou míru detailu.

2. Analytik vygeneruje podle specifikace sadu instancí modelu, kde jsou především v určitém smyslu mezní hodnoty, které model ještě připouští nebo naopak již nepřipouští. Velikost sady je úměrná složitosti modelu a musí být zvolena na základě zkušeností analytika. Případně může analytik namodelovat instance i ručně.

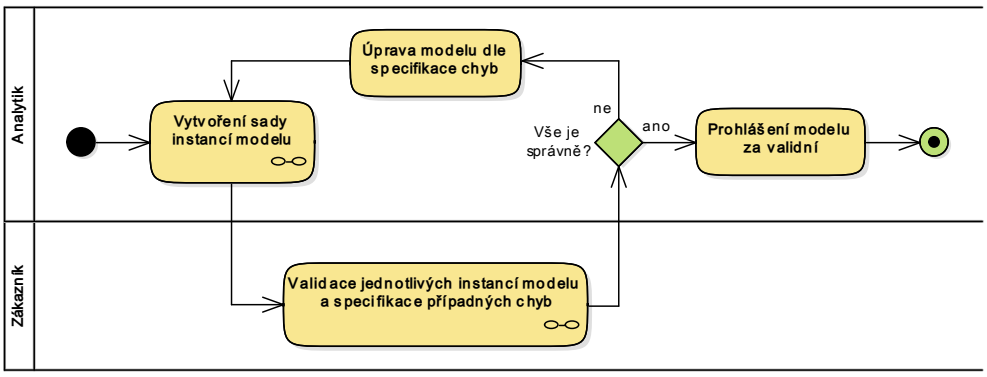
3. Zákazník společně se zákazníkem prochází vygenerované instance v grafické či případně textové podobě a rozhoduje, zda bylo rozhodnutí o jejich přípustnosti správné.

4. Pokud všechny instance modelu byly z pohledu zákazníka správné a odpovídají modelované doméně, pak je model prohlášen za validní a proces validace končí.

5. Při nalezení chyby zákazník analytikovi tuto chybu podrobně specifikuje.

6. Po identifikace všech chyb v sadě instancí analytik zapracuje na základě specifikace opravy a opakuje proces od kroku 2.

Po edukaci zákazníka je možné, aby krok 3 a 5 prováděl validaci sám zákazník, případně může s analytikem komunikovat vzdáleně prostřednictvím videohovoru se sdílením plochy. Při validaci je vhodné zapojit více doménových expertů za účelem odhalení více chyb.



Obrázek 7.5: Zjednodušený proces validace modelu se zákazníkem

7.6 Distribuce systému

Knihovna *hCM* včetně metodiky v rámci Haddock dokumentace je distribuována pod svobodnou licencí MIT přes GitHub a komunitní archiv Haskell balíčků zvaný Hackage, který funguje podobně jako PyPi pro jazyk Python, RubyGems pro jazyk Ruby nebo CTAN pro LaTeX. Díky Hackage je možné instalovat knihovnu snadno nástroji `cabal` a `stack`, přidávat je do dalších projektů přes specifikaci v souboru `.cabal`. Na Hackage stránce projektu je rovněž i aktuální odkaz na GitHub repozitář.

GitHub umožňuje efektivně projekt spravovat, vytvářet verze, vývojové větve, přispívat komunitě do projektu či ho upravovat ve vlastním repozitáři pomocí nástroje `git`. Zároveň má vlastní řešení pro hlášení chyb a nápadů na zlepšení, jednoduché projektové řízení pomocí Kanban, wiki platformu pro tvorbu dokumentace a komentování jednotlivých změn ve zdrojovém kódu.

Kapitola 8

Případová studie

K názorné demonstraci navrhovaného systému kódování konceptuálních modelů v jazyce Haskell slouží případová studie popsaná v této kapitole. Jedná se o modelování dostatečně komplexní problémové domény, na kterém lze dobře pozorovat vlastnosti prototypu systému. Nejprve je modelovaná doména popsána ve formě zadání podobnému tomu, co v reálné situaci vytváří IT analytik v rámci získávání znalostí o doméně. Poté je popsán a proveden postup zakódování konceptuální modelu do systému v jazyce Haskell, včetně částečného provedení verifikace, validace a vizualizace. V případové studii jsou tak simulovány běžné úkony prováděné při návrhu a vývoji SW.

Veškeré výstupy případové studie zahrnující zdrojový kód, potřebnou dokumentaci a vygenerované vizualizace jsou součástí obsahu přiloženého CD (viz příloha B) a jsou rovněž distribuovány přes GitHub jako příklad užití vytvořené knihovny pro veřejnost. Neboť se jedná o relativně velký konceptuální model, v textu práce jsou uvedeny pouze ukázky a některé vizualizace jsou velmi zmenšené za účelem demonstrace komplexity modelu. Čtenář elektronické verze může použít přiblížení, v případě tištěné verze je nutné pro detailní prozkoumání otevřít danou vizualizaci z CD nebo GitHub repozitáře.

8.1 Popis problémové domény

Jako problémová doména byla zvolena fiktivní půjčovna automobilů *KM Cars Rental*, dále uváděna jen jako *Společnost*. Tento příklad reprezentuje středně velkou společnost poskytující služby na několika svých pobočkách. Konceptuální modelování i popis domény jsou prováděny za účelem vývoje informačního systému podporující realizaci a evidenci výpůjček vozidel včetně výpočtu její ceny. Ostatní okolnosti problémové domény jako jsou mimo jiné účetnictví společnosti, mimořádné akce a slevy, údržba a provoz vozidel i poboček včetně jejich pojištění, evidence vybavení a další detaily domény nejsou modelovány (jsou tzv. *out of scope*).

8.1.1 Zákazníci

Zákazníkem je fyzická nebo právnická osoba, která je držitelem zákaznického účtu u společnosti. Zákaznický účet je nepřenosný a bez něj nelze realizovat výpůjčky vozidel. Zákaznický účet se řadí do věrnostní kategorie spojené s procentuální slevou na výpůjčku. Společnost u zákazníka musí evidovat kontaktní údaje, v případě fyzické osoby dále pak číslo občanského průkazu a rodné číslo, v případě právnické osoby IČ a údaje zaměstnanců oprávněných k výpůjčce na účet této právnické osoby. Zákazník je povinen půjčovně hlásit změny svých údajů evidovaných půjčovnou. Současně zákazník může kdykoliv svůj zákaznický účet zrušit.

8.1.2 Pobočky a zaměstnanci

Služby jsou poskytovány v rámci sítě poboček, kde za chod pobočky je zodpovědný vedoucí. Na pobočce pracují další řadoví zaměstnanci, kteří zprostředkovávají výpůjčky vozidel se zákazníky, poskytují potenciálním zákazníkům informace a přebírají zpět vozidla po výpůjčce. Zaměstnanci se v průběhu jejich působení ve firmě mohou mezi pobočkami přesouvat podle potřeby.

8.1.3 Vozový park

Společnost disponuje velkým množstvím různých motorových vozidel, které v rámci své činnosti půjčuje zákazníkům. Do vozového parku společnosti patří mimo běžných osobních automobilů také limuzíny, sportovní automobily, motocykly a dodávky. Společnost postupem času svůj vozový park obměňuje a rozšiřuje. U každého vozidla eviduje jeho základní údaje jako je výrobce, typ, SPZ, stáří, VIN kód a další. Vozidla se řadí do kategorií, podle které se určuje výpůjční cena.

8.1.4 Půjčení vozidla

Výpůjčka vozidla se realizuje na pobočce, kde fyzická osoba předloží svůj platný občanský a řidičský průkaz. V předstihu může zákazník provést rezervaci vozidla na budoucí vypůjčení, a to buď osobně, telefonicky nebo emailem. Před výpůjčkou je zkontrolován stav vozidla včetně stavu palivové nádrže a počtu najetých kilometrů. Zákazník může vrátit automobil na jakékoliv pobočce a při vrácení je opět zkontrolován jeho stav. Základ ceny výpůjčky je stanoven pomocí počtu dní (započatých) a skupině, do které vozidlo patří.

8.2 Příprava modelování

Ačkoliv je možné hned začít model zapisovat do souborů jako zdrojový kód, je vhodné si nejprve připravit prázdný projekt a rozvrhnout jeho strukturu. Jednoduché vytvoření nového projektu poskytuje nástroj `stack`. Stačí využít

příkaz `stack new` a podle základní šablony a nastavení se vytvoří všechny potřebné soubory a složky. Tento postup byl zvolen i při tvorbě případové studie. Do závislostí nově vytvořeného projektu byla přidána knihovna `hCM` pro podporu konceptuálního modelování v jazyce Haskell. Samotný projekt bude rozvržen do podmodulů `CM.CaseStudy` a odpovídající adresářové struktuře ve složce s názvem `Lib`.

8.3 Zakódování do systému

Jako první krok využití připraveného systému je nutné samotný konceptuální model zapsat ve formě předepsané navrženým systémem, aby bylo možné s ním dále pracovat a používat k vizualizaci, verifikaci, validaci a jako součást implementace požadovaného systému. Podkladem je textové zadání, ve kterém se nachází výchozí popis problémové domény. Kódování je prováděno v rámci navrženého kompilátorem řízeného modelování. Při modelování je průběžně prováděno kompilování projektu a zjišťována správnost zápisu i další nutný postup.

8.3.1 Konceptuální model

Po zakódování jednotlivých částí konceptuálního modelu přichází na řadu v souladu s navrženým systémem jejich finální seskupení do vlastního datového typu reprezentujícího celý konceptuální model, který je instancí typové třídy `ConceptualModel`. Ačkoliv tento typ již přímo nesouvisí s problémovou doménou, je nutný pro dobrou manipulaci s modelem jako celkem a jeho instancemi.

Pro implementaci tohoto typu byla zvolena varianta záznamového datového typu s atributy představujícími seznamy jednotlivých entit a vztahů. Příslušné funkce typové třídy `ConceptualModel` pak přímo pracují s těmito atributy. S postupným přidáváním entit a vztahů dle zpracování zadání jsou přidávány atributy datového typu a reflektovány v instanci typové třídy. Tato základní implementace je velmi jednoduchá, jak je zřejmé z příkladu 8.1. Volba seznamů typu `List` umožňuje snadnou práci pomocí dekompozice a základních funkcí `map`, `concat`, `length` a dalších.

8.3.2 Entity

Nejprve je nutné identifikovat v popisu problémové domény neboli zadání jednotlivé koncepty a jejich atributy. Důležité je rozhodování na základě požadavků a účelu modelu, co modelovat jako atribut a co jako samostatné entity. V rámci zadání byly identifikovány koncepty: subjekt se specializacemi fyzická osoba a právnická osoba, zákaznický účet, výpůjčka vozidel, věrnostní kategorie, pobočka, zaměstnanec jako speciální případ fyzické osoby, (motorové) vozidlo a kategorie vozidla.

Kód 8.1: Příklad zakódovaného modelu z případové studie

```
data CarRentalModel = CarRentalModel
                      { crmEPeople     :: [Person]
                      , crmECompanies  :: [Company]
                      -- ... more entities/relationships
                      , crmRAccOwner   :: [AccountOwnership]
                      } deriving (Show, Read, Eq)

instance CMElement CarRentalModel where
  toMeta = toMetaModel

instance ConceptualModel CarRentalModel where
  cmodelElements model = (map (toMeta model) $ crmEPeople model)
                      ++ (map (toMeta model) $ crmECompanies model)
                      -- ... more entities/relationships
                      ++ (map (toMeta model) $ crmRAccOwner model)
```

Tyto koncepty jsou postupně implementovány jako záznamové datové typy s příslušnými atributy primitivních či pomocných datových typů. Mezi pomocné datové typy patří například pohlaví, adresa, datum případně datum a čas. Po vytvoření datových typů následuje jejich přiřazení do konceptuálního modelu jako dalšího atributu typu seznamu instancí dané entity.

Po přiřazení entit ke konceptuálnímu modelu je možné využít kompilátorem řízeného modelování a nechat kompilátor vyhodnotit, co je nutné implementovat:

1. derivování typových tříd `Show` a `Read`,
2. instanciace typové třídy `Identifiable` s možností definice principu identity na základě atributů entity,
3. instanciace typové třídy `CMElement` s definováním povinných funkcí, kde je doporučeno využití funkce z typové třídy `Entity`,
4. instanciace typové třídy `Entity` s definováním povinných funkcí.

Po zakódování entit ve formě definice povinných funkcí tříd, je možné také upravit ty s výchozí definicí a například pomocí `CMElement` v souladu se systémem specifikovat omezení. Takových omezení je logicky v problémové doméně velké množství a velkou část z nich by odhalila až případná validace s doménovým expertem. V případové studii byla zakódována některá základní omezení entit jako například jednoduchá skutečnost, že datum narození osoby nemůže

být v budoucnosti. Alternativně by bylo možné toto i některá další omezení definovat také pomocí LiquidHaskell či jinými knihovnami pro Haskell.

Kód 8.2: Příklad zakódované entity z případové studie

```
data Person = Person { personalId      :: String
                     , personBirth     :: Date
                     , personFirstname :: String
                     , personLastname  :: String
                     , personHome      :: Address
                     , personGender    :: Gender
                     } deriving (Show, Read, Eq)

instance Identifiable Person where
  identifier = show . personalId

instance CMElement Person where
  toMeta = toMetaEntity

instance Entity Person where
  entitySuperNames _ = ["Subject"]
  entitySubNames _ = ["Employee"]
  entityAttributes Person {..} = map tupleToAttribute
    [ ("Personal ID", "String", personalId)
    , ("Birthdate", "Date", pretty personBirth)
    , ("Firstname", "String", personFirstname)
    , ("Lastname", "String", personLastname)
    , ("Home", "Address", pretty personHome)
    , ("Gender", "Gender", show personGender)
    ]
```

Kód 8.3: Ukázka omezení entity `Person`

```
personBirthInPast :: Person -> Validity
personBirthInPast Person {..} =
  newConstraint (dateInPast personBirth) "Person birth in future"
      where dateInPast = (< currentDateToday)
```

8.3.3 Vztahy

Poté, co jsou vytvořeny entity, přichází na řadu zachycení jejich vzájemných vztahů včetně skutečnosti, zda je pro entitu povinný a v kolika může participovat. Subjekt představovaný fyzickou či právnickou osobu může mít maximálně jeden zákaznický účet, ale také nemusí mít žádný. Zákaznický účet patří do žádné nebo jedné věrnostní kategorie, z druhé strany do věrnostní kategorie může patřit libovolný počet účtů, stejný typ vztahu je i mezi vozidly a jejich modely i kategoriemi.

Nejdůležitějším vztahem je výpůjčka vozidla, která spojuje zaměstnance, zákaznický účet, vozidlo, pobočku zahajující výpůjčku a pobočku ukončující výpůjčku. Navíc jako pečetidlo (*truthmaker*) slouží smlouva nesoucí potřebné údaje o samotné výpůjčce, která může participovat pouze v rámci jedné výpůjčky. Na druhou stranu pobočka, vozidlo, zákaznický účet i zaměstnanec mohou participovat v libovolném množství výpůjček. Omezením tohoto vztahu je například skutečnost, že vozidlo nemůže být v jeden čas vypůjčeno vícekrát.

Kód 8.4: Příklad zakódovaného vztahu z případové studie

```
data Rental = Rental { contract   :: RentalContract
                     , renter     :: CustomerAccount
                     , lessor     :: Employee
                     , rentedCar  :: Car
                     , rentedFrom :: BranchOffice
                     , rentedTo   :: BranchOffice
                     } deriving (Show, Read, Eq)

instance Identifiable Rental
  -- default identity principle

instance CMElement Rental where
  toMeta = toMetaRelationship

instance Relationship Rental where
  relationshipParticipations Rental {..} = map tupleToParticipation
    [ ("contract", "RentalContract", identifier contract, r11)
    , ("renter", "CustomerAccount", identifier renter, r0x)
    , ("lessor", "Employee", identifier lessor,  r0x)
    , ("rentedCar", "Car", identifier rentedCar,  r0x)
    , ("rentedFrom", "BranchOffice", identifier rentedFrom,  r0x)
    , ("rentedTo", "BranchOffice", identifier rentedTo,  r0x)
    ] where r11 = Mandatory Unique
            r0x = Optional Unlimited
```

Postup je velice podobný jako u entit, jednotlivé vztahy jsou zakódovány ve formě záznamových datových typů s příslušnými participanty jako atributy. Dále je možné již opět použít kompilátorem řízené modelování a nechat kompilátor zjistit, co je nutné specifikovat:

1. derivování typových tříd `Show` a `Read`,
2. instanciace typové tříd `Identifiable` se základním principem identity,
3. instanciace typové třídy `CMElement` s definováním povinných funkcí, kde je doporučeno využití funkce z typové třídy `Relationship`,
4. instanciace typové třídy `Relationship` s definováním povinných funkcí.

Podobně jako pro entity, byly v rámci případové studie namodelovány některá důležitá omezení pro vztahy. Jedním z nich je zmíněné omezení vypůjčení vozidla vícekrát v jeden čas, viz příklad 8.5. Modelování omezení vztahů je ve srovnání s jednoduchými omezeními entit složitější o nutnost dotazování nad modelem a zjišťování ostatních vztahů. To je však díky ponechání na uživateli systému relativně flexibilní a volně přizpůsobitelné.

Kód 8.5: Ukázka omezení vztahu `Rental`

```
carRentalSameTime :: (ConceptualModel m) =>
                                    m -> Rental -> Validity
carRentalSameTime model rental =
  newConstraint (sameTimeRent > 1) msg
  where msg = concat [ "Car rented more than once (exatly "
                     , show sameTimeRent
                     , " times) at the same time"
                     ]
        sameTimeRent = length . filter sameCarAndTime $ rentals
        sameCarAndTime r = sameCar r && timeOverlaps r
        -- ... sameCar & timeOverlaps with use of model & rental
```

8.3.4 Chování

Entity, vztahy a omezení popisují strukturu konceptuálního modelu, pro implementaci informačního systému je nutné však specifikovat i chování v problémové doméně. Doména modelovaná v této případové studii obsahuje velké množství procesů: založení, změnu či zrušení zákaznického účtu, zahájení a dokončení výpůjčky včetně výpočtu ceny, změna údajů vozidla či subjektu a mnoho dalších. Protože navrhovaný systém nijak zakódování chování neomezuje ani nekontroluje a není ani součástí implementace využívající toto chování, je v případové studii pro demonstraci implementováno chování jen částečně.

8.4 Verifikace

Pro ověření správnosti zápisu modelu do zdrojového kódu ve formátu předepsaným systémem byla prováděna verifikace kompilátorem GHC. Díky použití standardního Haskell projektu s využitím nástroje `stack`, stačilo použít pouze příkaz `stack build`.

Verifikace byla při zápisu modelu v prototypu systému prováděna průběžně za účelem využití kompilátorem řízeného modelování klasickým způsobem v příkazové řídce, ale i *on-the-fly* pomocí pluginu do textového editoru, což práci značně zpříjemňovalo. Mimo zobrazení chyb vedoucích procesem kódování konceptuálního modelu pomocí typových tříd kompilátor často upozornil na typické chyby způsobené primárně překlepy či zapomenutými konstrukcemi z kopírování.

Veškerá chybová hlášení, která se při práci objevila, byla vždy zcela srozumitelná včetně přesné lokace i zdůvodnění chyby, a tudíž každá chyba byla snadno odstraněna. Nikdy nenastal problém s nejasností či nejednoznačností chybového hlášení, které by vyústilo v nutnost složitého hledání řešení.

Kód 8.6: Ukázka výstupu `stack build`

```
.../hCM-CaseStudy$ stack build
hCM-0.1.0.0: unregistering (local file changes: ...)
hCM-CaseStudy-0.1.0.0: unregistering (missing dependencies: hCM)
hCM-0.1.0.0: configure (lib + exe)
hCM-0.1.0.0: build (lib + exe)
hCM-0.1.0.0: copy/register
hCM-CaseStudy-0.1.0.0: configure (lib + exe)
Configuring hCM-CaseStudy-0.1.0.0...
hCM-CaseStudy-0.1.0.0: build (lib + exe)
Preprocessing library hCM-CaseStudy-0.1.0.0...
[2 of 3] Compiling CM.CaseStudy.Model (...)
[3 of 3] Compiling CM.CaseStudy.Instances.Manual (...)
Preprocessing executable 'hCM_CS' for hCM-CaseStudy-0.1.0.0...
[1 of 1] Compiling Main (...)
Linking .stack-work/dist/.../build/hCM_CS/hCM_CS ...
hCM-CaseStudy-0.1.0.0: copy/register
Installing library in
.../lib/x86_64-linux-ghc-8.0.2/hCM-CaseStudy-0.1.0.0-...
Installing executable(s) in
.../bin
Registering hCM-CaseStudy-0.1.0.0...
```

8.5 Validace

Ačkoliv v této ukázkové případové studii nebylo možné reálně validovat model se skutečným zákazníkem, byla vyzkoušena jeho část, ve které se vytváří instance pro validování. Skutečný zákazník by poté v souladu s navrženou metodikou dané instance schválil nebo popsal jejich chyby. Ve druhém případě by následovala úprava modelu a zopakování vytvoření instancí s důrazem na původně chybnou oblast.

8.5.1 Ruční vytváření instancí modelu

Protože se jedná o relativně velký model ruční vytváření instancí není příliš pohodlné. Za účelem demonstrování tohoto postupu a vytvoření ukázkové instance pro účely validace a vizualizace byla napsána jedna instance modelu ručně. Tato instance modelu obsahuje od každé entity a vztahu alespoň jednu instanci. Pro zachování přehlednosti byla instance zapsána do modulu `CM.CaseStudy.Instances.Manual`. Tím je dosaženo přehledného oddělení nesouvisejících částí kódu a zároveň snadného použití instance modelu.

Kód 8.7: Ručně vytvořené instance entit a vztahu

```
carBMW1 = Car { carColor = "red"
              , carVIN = "STXUFBAK654375424"
              , carNumberPlate = "ABC 5487"
              , carYearManufactured = 2009
              }

modelBMW = CarModel { carModelType = "sedan"
                    , carModelName = "BMW 320 d"
                    , carModelMaxSpeed = 221
                    , carModelNumberOfDoors = 5
                    }


bmw1IsBMW = CarInstanceOfModel { carInstance = carBMW1
                               , itsModel = modelBMW
                               }

model = CarRentalModel { -- ...
                       , crmECars = [carBMW1, carBMW2, carCamaro]
                       -- ...
                       }
```

8.5.2 Generování instancí modelu

Pro jednotlivé entity, vztahy a samotný konceptuální model byla implementována funkce `arbitrary` typové třídy `Arbitrary` knihovny QuickCheck za účelem pseudonáhodného generování instancí modelu. Pro jednotlivé atributy entit byly vytvořeny seznamy hodnot reflektující situace z reálného světa, viz příklad 8.8. Princip použití generování je opět shodný s ukázkami v předchozích kapitolách a ukazuje, jak jednoduše lze s QuickCheck pracovat.

Kód 8.8: Ukázka instance třídy `Arbitrary` pro `Person`

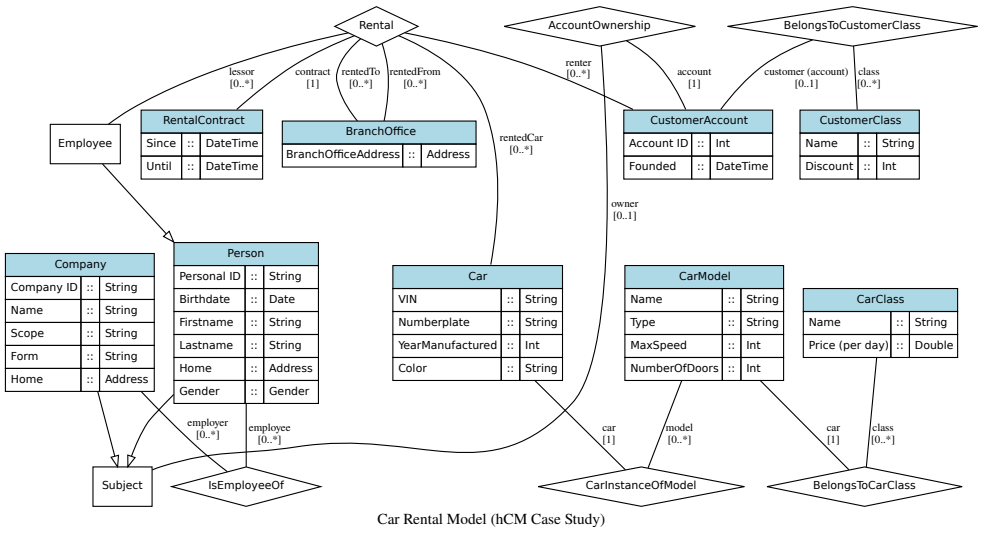
```
instance Arbitrary Gender where
  arbitrary = do
    isMale <- arbitrary
    return $ if isMale then Male else Female

instance Arbitrary Address where
  arbitrary = do
    street <- elements Values.streets
    streetNum <- choose (1, 2000) :: Gen Word
    city <- elements Values.cities
    postcode <- choose (10000, 90000) :: Gen Word
    country <- elements Values.countries
    return Address { addressStreet = street
                   -- ...
                   }

idchars = elements (['A'..'Z'] ++ ['0'..'9'])

instance Arbitrary Person where
  arbitrary = do
    pid <- replicateM 12 idchars
    gender <- arbitrary
    birthdate <- arbitrary
    home <- arbitrary
    firstname <- case gender of
      Male -> elements Values.firstnamesMale
      _    -> elements Values.firstnamesFemale
    lastname <- elements Values.lastnames
    return Person { personalId = pid
                  -- ...
                  }
```

8.6 Vizualizace

Průběžně v rámci konstrukce konceptuálního modelu v navrženém systému, při verifikaci i validaci byly generovány vizualizace modelu a jeho instancí. Vizuální kontrola odhalila řadu chyb a možností zlepšení modelu, které se z textového zápisu modelu identifikovaly hůře. Potvrdila se tak nutnost dobré vizualizace pro konceptuální modelování.

8.6.1 Vizualizace modelu

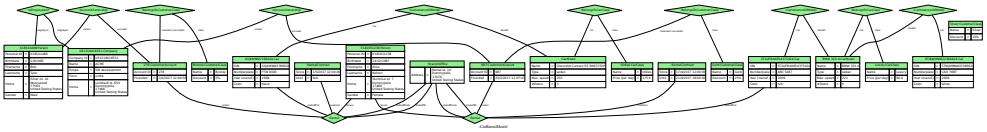
Pro vizualizaci modelu byl zvolen postup s vlastní definicí převodu metamodelu, neboť slouží pro veřejnou demonstraci a bylo vhodné upravit názvy atributů i jejich zobrazované názvy typů. Vizualizace na obrázku 8.1 byla upravena ve formátu DOT přidáním dvou jednoduchých řádků s příkazem `rank` pro změnu uspořádání, aby byla lépe umístitelná do textu práce, zbytek odpovídá základnímu generování vizualizací modelů z prototypu systému.



Obrázek 8.1: Vizualizace zakódovaného modelu

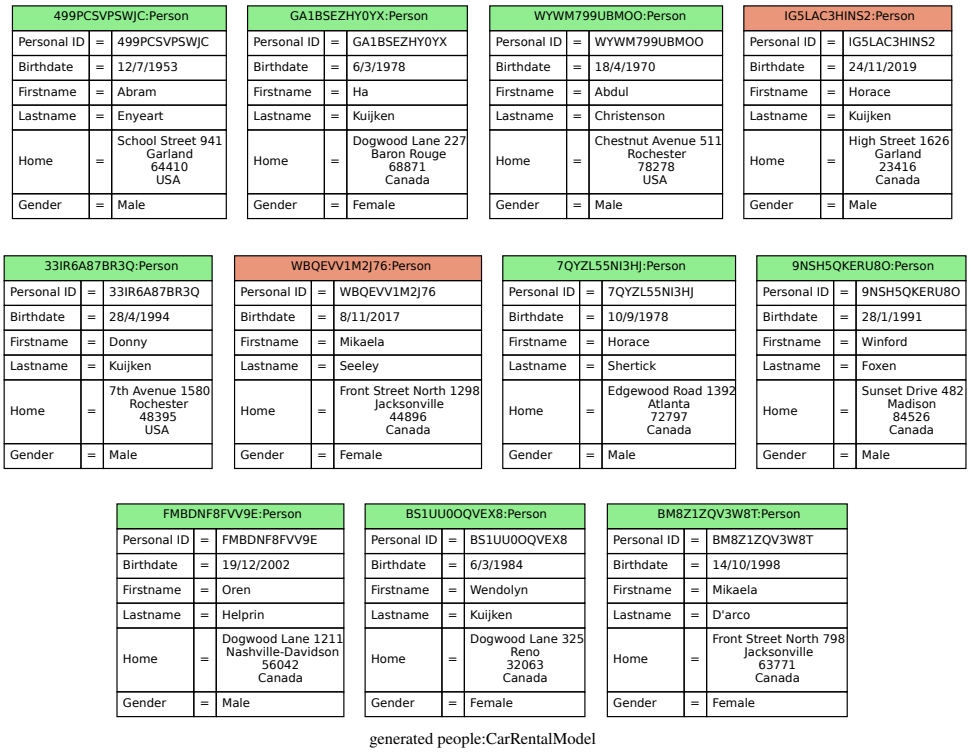
8.6.2 Vizualizace instance

Vizualizace instancí probíhá díky zvolené možnosti systému podobně jako vizualizace modelu, jen se mění volaná funkce nad stejnou instancí modelu. Jak bylo zmíněno, pro vizualizaci byla napsána instance modelu ručně, ta je vidět na obrázku 8.2. Vizualizace nebyla nijak upravena a je ve výchozím rozložení prvků. Ačkoliv je obrázek relativně velký, a proto zmenšený, na první pohled je vidět, že jsou všechny části modelu přípustné, díky barevnému odlišení.



Obrázek 8.2: Vizualizace instance zakódovaného modelu

Dále byly v rámci validace generovány instance modelů i jeho jednotlivých částí. Následující vizualizace na obrázku 8.3 ukazuje několik přípustných i nepřípustných instancí entity `Person`. Jednotlivé entity byly ve vizualizaci opět přeskupeny pomocí DOT konstrukce `rank` za účelem lepší čitelnosti v textu. Dále byla za účelem vyzkoušení ještě výsledná vizualizace ve vektorovém formátu SVG upravena pomocí grafického editoru a za zmínku stojí dobrá vlastnost Graphviz, který při převodu jednotlivé prvky dobře seskupuje a díky tomu lze obrázek lépe upravovat.



Obrázek 8.3: Vizualizace vygenerovaných instancí `Person`

8.7 Zakódovaný konceptuální model

Během kódování, verifikace, validace a vizualizace nenastaly žádné problémy. V programovacím jazyce Haskell je možné zapsat prakticky jakýkoliv model struktury a chování s využitím konstrukcí navrženého podpůrného systému. Konceptuální model ze zadání je zakódován pomocí prototypu, ale samozřejmě může být dále zpřesňován a rozvíjen, neboť konceptuální modelování je v tomto smyslu iterativní proces, ke kterému je však nutný doménový expert znalý všech souvislostí a potřeb.

V rámci zpřesňování by mohly být přidávány především další potřebné atributy jednotlivých entit, chování a pak také omezení, která jsou modelována jen za účelem jejich demonstrace v případové studii. Po konzultaci s doménovým expertem by bylo také možné zpřesnit generování instancí modelu pro validaci.

Zakódovaný konceptuální model pravděpodobně obsahuje řadu nedostatků z pohledu problémové domény, které by však v reálné situaci byly odhaleny právě pomocí iterativního procesu validace dle navrženého metodického postupu. Vytvoření přesného a správného konceptuálního modelu na první pokus je i při přímé spolupráci s doménovým expertem spíše výjimečné, tím je validace tak důležitá. Zpravidla je potřeba pro vytvoření kvalitního modelu provádět modelování a validaci s více než jedním doménovým expertem. [23]

Kapitola 9

Zhodnocení výsledného systému

V této kapitole jsou shrnuty klady a zápory navrženého systému pro kódování konceptuálních modelů v jazyce Haskell a implementovaného prototypu. Zhodnocení staví na výsledcích a poznatcích z případové studie popsané v předcházející kapitole, ale i na načerpaných znalostech možností existujících přístupů propojení konceptuálních modelů s implementací. Kapitola rovněž obsahuje návrh dalšího rozvoje systému a výzkumu v oblasti reprezentace konceptuálních modelů za pomoci programovacích jazyků.

9.1 Vlastnosti prototypu systému

Klady a zápory navrženého systému jsou v této části kapitoly vztaženy k jeho jednotlivým klíčovým částem a poté shrnuty. Zároveň popisuje, jak úspěšně byly naplněny všechny stanovené cíle a požadavky na systém, které vyplynuly z provedené rešerše a analýzy.

9.1.1 Kódování konceptuálních modelů

Jazyk Haskell poskytuje z pohledu této práce velmi dobrou expresivitu pro zachycení konceptuálních modelů přímo v rámci základní syntaxe bez použití dalších rozšíření jazyka ani složitějších konstrukcí, které by zhoršovaly použitelnost méně zkušenými programátory. Neméně důležitá je přehlednost syntaxe a možnost členění kódu do modulů se zapouzdřením některých privátních detailů. Ačkoliv jazyk Haskell postrádá přímé odrážení objektů reálného světa, jako je tomu u objektově orientovaných jazyků, díky silným matematickým kořenům poskytuje prostředky pro verifikaci a validaci na úrovni specializovaných metod formální specifikace.

Navržený systém těchto vlastností jazyka využívá a snaží se být v tomto ohledu co nejjednodušší a tím i flexibilní a univerzální. Pro tvorbu modelů a další práci se systémem je vyžadována základní znalost jazyka Haskell, na což lze pohlížet jako na negativní vlastnost, nicméně znalost programovacího

jazyka je použitelná i k dalším činnostem na rozdíl od znalosti specifikačních či modelovacích jazyků, které jsou v tomto smyslu jednoúčelové. Systém také vede následný vývoj aplikace k používání správných postupů ve formě rozdělení implementace do vrstev a použití typových tříd, čistých funkcí, skládání datových typů a polymorfismu.

9.1.2 Verifikace

Verifikace konceptuálního modelu má za úkol zajistit soulad implementace s modelem, a ten je zajištěn přímo tím, že konceptuální model je nedílnou součástí implementace neboli operačního modelu. Další výhodou je ověření správnosti zápisu modelu přímo kompilátorem, který zobrazuje přehledná chybová hlášení přesně specifikující danou chybu, a vedení samotným procesem ve smyslu kompilátorem řízeného modelování. Za účelem zpřesnění automatizovaných kontrol je možné využít libovolných dalších nástrojů a knihoven pro jazyk Haskell, jako je například LiquidHaskell, který umožňuje i tvorbu a vyhodnocení formálních důkazů a hledání protipříkladů [83].

Nevýhodou je nutnost vytvoření implementace v jazyce Haskell, který není tak rozšířený v komerčním prostředí jako například Java, C++ nebo C#. Nicméně jazyk Haskell je navržen i pro velké komerční projekty [80]. Současně lze očekávat, že konceptuální model již zakódovaný do programovacího jazyka bude snáze převoditelný než například z jiných, především grafických, notací.

9.1.3 Validace

V rámci návrhu a implementace systému byla navržena metodika validace konceptuálního modelu se zákazníkem, která předepisuje jasný a jednoduchý postup za účelem ověření souladu modelu s problémovou doménou. Analytikovi provádějícímu validaci je umožněno instance modelu generovat dle vlastních pravidel a případně je upravovat i vytvářet ručně. Vytvořené instance, které lze exportovat a importovat, mohou být přípustné i nepřípustné z hlediska modelu, což je snadno zjistitelné. Díky tomu se proces validace značně zefektivňuje, což vede ke kvalitnějšímu výsledku.

K vytváření generátorů instancí složitějších modelů je rovněž nutná větší znalost jazyka Haskell i frameworku QuickCheck. Tyto znalosti jsou však velmi vhodné při vývoji komplexních aplikací a požadavky na znalosti čistě pro účely konceptuálního modelování v navrženém systému jsou ve srovnání s jinými metodami minimální. Vytvořené generátory je poté možné použít při testování výsledné aplikace či dalších projektech s podobnými problémovými doménami.

9.1.4 Vizualizace

Systém umí konceptuální modely a jejich instance vizualizovat. Grafická notace čerpá z ER a UML, čímž by měla být pro většinu analytiků dostatečně

čitelná bez nutnosti studia příslušné dokumentace. Současně umožňuje v rámci instancí zobrazovat jejich přípustnost a tím je vhodná pro použití při validaci konceptuálního modelu.

Ačkoliv je pro vizualizaci zvolen nástroj Graphviz namísto zcela vlastního řešení a je vyžadována jeho instalace a základní znalost použití pro převod kódu DOT do grafického formátu, přináší na druhou stranu i spoustu výhod. Výstupní grafický formát je téměř libovolný, výsledný soubor lze upravovat v grafickém editoru, stejně tak je možné provádět úpravy i v textovém formátu DOT. Dále existují i interaktivní nástroje s grafickým rozhraní pro práci s DOT a Graphviz. To vše dává poměrně široké možnosti, které by bylo velmi složité implementovat přímo v systému a ten by se stal zbytečně složitým.

9.1.5 Použitelnost systému

Vytvořený prototyp systému je díky zvoleným distribučním kanálům GitHub a Hackage jednoduše použitelný. Systém včetně svých závislostí je kompatibilní s běžnými platformami Linux, Windows i Mac. Prerekvizitou je pouze instalace GHC a nástroje `stack` pro instalaci balíčku včetně všech závislostí, kterých je však minimum. Výstupy ve formě serializovaných instancí i vizualizací jsou shodně jako samotný zakódovaný konceptuální model ve formě zdrojového kódu přenositelné a platformě nezávislé.

Kódování konceptuálního modelu v navrženém systému vyžaduje pouze základní znalosti jazyka Haskell a zápis částí modelu velmi často svou syntaxí připomíná DSL. Reprezentace pomocí standardních konstrukcí jazyka Haskell a implementace GHC umožňuje snadné použití definovaných typů, jejich vztahů, omezení a funkcí popisujících chování přímo v rámci konceptuální vrstvy či vrstvy business logiky výsledného SW produktu. Všechny tyto zmíněné vlastnosti vedou k velmi intuitivní, flexibilní a efektivní použitelnosti navrhovaného systému.

9.1.6 Shrnutí výhod a nevýhod systému

Výhody navrženého systému jsou zřetelné z uvedeného zhodnocení jednotlivých jeho částí a vlastností výše. Mimo udržení konzistence mezi konceptuálním modelem a implementací ve formě SW se jedná především o schopnosti jednoduché a přehledné verifikace, validace a vizualizace. Dále se jedná o snadnou použitelnost a rozšiřitelnost vlastního systému.

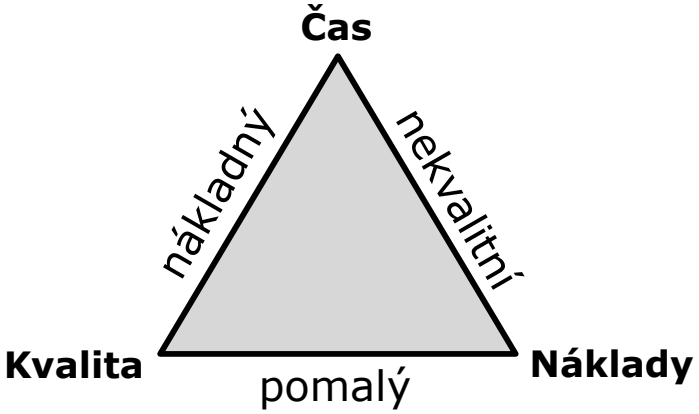
Velkou nevýhodou je logicky nulová rozšířenost a komunita, neboť se jedná o zcela nové řešení. Stejně tak je tomu i v případě nástrojové podpory, která sice díky povaze systému a možnosti užití pouze textových editorů nebo IDE není tak potřebná, ale může zlepšovat možnosti využití systému. Související slabou stránkou je i omezení se na jeden programovací jazyk, což může bránit širšímu použití. Ačkoliv je přístup ke konceptuálnímu modelování odlišný oproti klasickým metodám nese s sebou řadu výhod.

9.2 Ekonomicko-manažerské přínosy

Ekonomicko-manažerské přínosy navrženého systému a postupů pro tvorbu konceptuálních modelů v rámci vývoje software nelze nyní v rámci této práce přesně vyčíslit. A to i mimo jiné, protože je jisté, že tyto přínosy závisí na různých vlastnostech projektů, například na:

- velikosti projektu (rozsah díla a počet členů projektového týmu),
- typu a požadované kvalitě SW (zda je vyžadována verifikace a validace, záruka bezpečnosti a spolehlivosti),
- zvolené metodice vývoje (tradiční či agilní),
- plánech na další rozvoj SW,
- kvalifikaci členů projektového týmu,
- integraci s jinými aplikacemi,
- rozšiřitelnosti a škálovatelnosti SW.

Pro zjištění reálných výhod a vlivu uvedených vlastností by bylo nutné vytvářet shodné či alespoň velmi podobné projekty za stejných podmínek různými metodami. Je možné předpokládat, že svými vlastnostmi se systém kódování konceptuálních modelů přímo v programovacím jazyce hodí k užití především ve větších projektech s plány na budoucí rozvoj a garancí vysoké kvality. Hypotetické přínosy lze demonstrovat na vlivech vzhledem k jednotlivým částem projektového trojimperativu: náklady, čas, rozsah či kvalita [20]. Další skupinou přínosů jsou pak možné pozitivní dopady na vedení týmu a zefektivnění samotného procesu vývoje SW z manažerského hlediska.



Obrázek 9.1: Projektový trojimperativ (podle [20])

9.2.1 Snížení nákladů

Použití navrhovaného systému nevyžaduje žádný komerční CASE ani jiný nástroj a tím přímo snižuje náklady na potřebné licence k software. Systém je multiplatformní a shodně jako Haskell jej lze používat v čistě open-source prostředí, není to však nutnou podmínkou.

Dále systém vede i ke snížení nákladů na personální zdroje, protože není nutné mít v projektovém týmu odborníky na tvorbu modelů v různých notacích, ale stačí analytik znalý konceptuálního modelování a vývojáři v jazyku Haskell, kteří jsou však využíváni i při implementaci a jejich práce s modelem od počátku umožní snazší práci později. Vlivem toho je možné, aby byl projektový tým celkově menší.

Náklady na údržbu a rozvoj software vyvinutého pomocí navrhovaného systému se snižují díky jednoduššímu promítnutí změn problémové domény přímo do zdrojového kódu přes zakódovaný konceptuální model. Počet osob zpracovávajících změnu tak může být opět nižší.

9.2.2 Zvýšení kvality

Systém zjednodušuje procesy verifikace a validace, které jsou pro dosažení vysoké kvality klíčové, a tím zároveň podněcuje projektový tým, aby důkladně navrhl samotný konceptuální model. To umožňuje dosažení vyšší kvality výsledného software pomocí konzistentní implementace přímo nad validovaným a verifikovaným konceptuálním modelem problémové domény.

Současně systém předepisuje, jak vytvářet generátory instancí modelu, které jsou použitelné i v dalších fázích vývoje za účelem udržení souladu s modelem, tudíž i realitou. Testování výstupů projektu může být pak efektivnější a jednodušeji konzultovatelné se zákazníkem.

9.2.3 Zrychlení dodání výsledného SW

Dodání výsledného informačního systému či aplikace může být zrychleno, protože navrhovaný systém již v sobě zahrnuje části snadno využitelné v dalších fázích vývoje a usnadňuje tak jejich průběh. Logicky v rámci trojimperativu lze časové úspory kompenzovat nižšími náklady, pokud na projektu pracuje méně osob déle, nebo zvýšením kvality a rozsahu díla.

Čas na samotnou implementaci se zkrátí díky tomu, že již během konceptuálního modelování vznikne podstatná část aplikace, která je přímo použitelná. Rovněž se zkrátí i potřebný čas pro fázi testování, neboť během konceptuálního modelování je již prováděna verifikace a validace, což eliminuje větší chyby ve funkcionalitě aplikace, a rovněž jsou připraveny v rámci validace testovací data či jejich pseudonáhodné generování.

V neposlední řadě by SW vytvořený s navrhovaným systém měl umožňovat rychlejší a snazší změny i údržbu díky přímo zakomponovanému konceptuální modelu a vedení k vývoji v souladu s osvědčenými postupy. Změnové řízení

může vést totiž ke změně konceptuálního modelu a při správné implementaci je potřeba provádění dalších nutných změn již minimální či triviální.

9.2.4 Zefektivnění procesů a vedení týmu

Použití systému kódování konceptuálních modelů přímo v programovacím jazyce může mít navíc další výhody v souvislosti s procesy vývoje SW a vedení týmu. Tyto možné pozitivní dopady pak logicky vedou k výše uvedeným přínosům či s nimi nepřímo souvisí.

- Konceptuální model i jeho instance lze jednodušeji verzovat a porovnávat změny pomocí standardních nástrojů pro verzování zdrojových kódů.
- Není nutné využívat jiné nástroje pro týmový přístup ke konceptuálnímu modelu než se používají pro zdrojové kódy aplikace.
- Všichni členové týmu od analytiků přes programátory až po testery pracují se stejným konceptuálním modelem a informovanost celého týmu o změnách je jednodušší.
- Projektový manažer díky výše uvedeným výhodám může mít lepší přehled o práci všech členů týmu na projektu od sběru požadavků až po nasazení a údržbu.
- Vývojáři a testeři mají možnost získat větší povědomí o problémové doméně a tím vytvářet kvalitnější aplikace reflektující potřeby uživatelů.
- Analytici mají naopak možnost získat větší povědomí o aplikaci a efektivněji zpracovávat požadavky od zákazníka.
- Díky jednotnému přístupu k celému vývoji je jednodušší sledovat výkonové metriky jednotlivých členů týmu a na základě nich je spravedlivěji odměňovat a motivovat k ještě lepší práci.
- Všechny výše uvedené výhody mají podíl na lepší organizaci práce a její zpětné analýzy za účelem stanovení předběžných nákladů, finanční návratnosti a časové náročnosti budoucích projektů.

9.2.5 Metriky pro srovnání projektů

V případě dostupnosti dat o shodných projektech vytvářených za pomoci navrženého systému a jiné metody by bylo možné spravedlivé porovnání a přesná kvantifikace přínosů. Za účelem takového porovnávání je nutné stanovení metrik a dalších vypovídajících ukazatelů, pomocí nichž by se poté dalo rozhodovat, jakou metodiku pro vývoj dle okolností a požadavků zvolit.

Zároveň by došlo k odhalení závislosti výhod navrženého systému na různých parametrech projektů a potvrzení či vyvrácení zmíněných hypotetických

ekonomicko-manažerských přínosů. Další možností by mohlo být vyhodnocení experty na základě zkušeností, ale to samozřejmě neposkytuje takovou vypovídající hodnotu jako reálná data. Mezi vhodné metriky a statistiky k porovnání by mohly posloužit:

- celková doba trvání projektu a rozložení času na jednotlivé fáze projektu (analýza, návrh, implementace, testování, ...),
- vynaložené celkové náklady i náklady na vybrané kategorie (SW, HW, personalistika, ...) a fáze projektu,
- počet nutných změnových řízení, jejich závažnost a související náklady,
- vyžadovaná součinnost zákazníka v jednotlivých fázích projektu,
- životnost SW a kvalitativní metriky (dostupnost, odezva, časy průchodu, komplexita kódu, ...).

9.3 Další rozvoj systému

9.3.1 Rozšíření a vývoj nových nástrojů

Navržený systém včetně nástrojů pro vizualizaci a metodiky pro generování instancí konceptuálních modelů lze dále rozvíjet. Ačkoliv je prototyp systému pro kódování konceptuálních modelů v jazyce Haskell navržen na základě analýzy různých současných řešení, vlastností programovacích jazyků i samotného jazyka Haskell tak, aby byl snadno použitelný a zároveň dostatečně univerzální, jeho praktické využití v různých reálných projektech může odhalit místa ke zdokonalení.

Díky zvolené licenci a způsobu distribuce je možné, aby kdokoliv systém rozšiřoval podle svých potřeb. Současně lze systém obohacovat o další podpůrné nástroje. Mezi oblasti, které by mohly takové nástroje podpořit, patří například:

- částečná automatizace procesu validace modelu se zákazníkem,
- intuitivní CASE nástroj s GUI pracující nad zdrojovým kódem přímo s vizualizací konceptuálního modelu,
- vyčíslení nutného počtu validovaných instancí na základě komplexity i dalších vlastností modelu a problémové domény,
- generování kódu modelu a jeho instancí pomocí anotací nebo
- převod konceptuálních modelů v jazyce Haskell z a do jiných notací.

9.3.2 Ontologie

Implementovaný systém pro kódování konceptuálních modelů je dostatečně obecný a zároveň rozšiřitelný. Tím umožňuje snadné přidání vrstev s nižší úrovní abstrakce, kdy systém je použit jako metametamodel a je přidána vrstva metamodelu.

Příkladem takového rozšíření by mohlo být přidání konceptů a jejich vztahů z vyšší ontologie UFO. Stačilo by definovat nové typové podtřídy třídy `Entity`: `Kind`, `Subkind`, `Role`, `Phase` a další, stejně tak i vztahy a přidat omezení definovaná v ontologii. Rozšiřování systému je v podstatě analogické k využití UML stereotypů na třídách a asociacích.

9.3.3 Výzkum jiných jazyků

Jak se ukázalo, jazyk Haskell má dobrou úroveň expresivity a řadu vlastností, které jsou pro reprezentaci konceptuálních modelů velmi praktické. Jiné vlastnosti však chybí a je tak možné, že jiný programovací jazyk by na tom mohl být s reprezentací konceptuálních modelů ještě lépe. Další výzkum se může zaměřit na zkoumaní jiných programovacích jazyků z hlediska jejich expresivity a celkové použitelnosti jako prostředku pro kódování doménových konceptuálních modelů za účelem provázání s implementací.

Jedním z kandidátů je například multiparadigmatický jazyk Python, který disponuje dobrou podporou metaprogramování a reflexe, je relativně rozšířený a syntaxe stavící na odsazení dělá kód přehledným. Mezi další vhodné kandidáty svými vlastnostmi patří i čistě objektově orientované jazyky Smalltalk a Ruby, poměrně nový multiparadigmatický jazyk Rust nebo funkcionální jazyk s objektově orientovým přístupem Scala.

Závěr

Hlavním cílem této práce bylo navrhnout systém pro kódování konceptuálních modelů do jazyka Haskell a na jeho základě implementovat prototyp systému. Zvolený tradiční postup vývoje SW se osvědčil. V souladu se zadáním diplomové práce byla provedena rešerše oblasti konceptuálního modelování včetně jednotlivých metod a nástrojů, analýza relevantních vlastností programovacích jazyků vzhledem ke konceptuálnímu modelování, požadavků na systém a samotného jazyka Haskell.

Výsledkem analýzy i následného návrhu systému bylo zjištění, že vlastnosti jazyka Haskell jsou pro konceptuální modelování velmi dobré a samotný systém lze vytvořit relativně jednoduše s naplněním všech stanovených požadavků. Zároveň bylo docíleno umožnění snadného použití a zakomponování systému do rozličných projektů bez nutnosti hlubších znalostí specializovaných knihoven či pokročilých konstrukcí jazyka.

V rámci návrhu a implementace systému byla navržena metodika tvorby modelu a ověřování jeho správnosti označená jako *kompilátorem řízené modelování*. Dále byl stanoven postup pro validaci modelů se zákazníkem, generování instancí modelů a jejich vizualizaci. Prototyp systému je veřejně dostupný a obsahuje vlastní dokumentaci popisující detaily nad rámec popisu v samotném textu této práce.

Implementovaný prototyp systému byl následně dle zadání demonstrován na případové studii, podle které byl systém také vyhodnocen. V hodnocení prototypu systému byla identifikována pozitiva, která některé metody konceptuálního modelování a nástroje na jejich podporu zcela postrádají. Součástí zhodnocení byly zmíněny ekonomicko-manažerské přínosy, které systém může do projektů vývoje SW přinést, a to jak z pohledu výstupů ve formě výsledného software, tak i v rámci vedení samotného projektu.

Vytyčené cíle včetně samotného zadání diplomové práce byly úspěšně naplněny a byl navržen i další rozvoj systému pro kódování konceptuálních modelů v jazyce Haskell i možné další směry výzkumu v této oblasti. Výstupy práce jsou volně dostupné a připravené k použití a dalšímu rozvoji.

Literatura

[1] MYLOPOULOS, John: Conceptual Modelling and Telos. [online], 1992, University of Toronto. Dostupné z: http://www.cs.toronto.edu/~jm/2507S/Readings/CM+Telos.pdf

[2] KOTIADIS, Kathy a Stewart ROBINSON: Conceptual Modelling: Knowledge Acquisition and Model Abstraction. In *Proceedings of the 40th Conference on Winter Simulation*, WSC '08, Winter Simulation Conference, 2008, ISBN 978-1-4244-2708-6, s. 951–958.

[3] GUIZZARDI, Giancarlo: *Ontological Foundations for Structural Conceptual Models*. Enschede (The Netherlands): Centre for Telematics and Information Technology, Telematica Instituut, University of Twente, 2005, ISBN 978-9-07517-681-0. Dostupné z: http://doc.utwente.nl/50826/1/thesis_Guizzardi.pdf

[4] SMITH, Barry: Ontology. In *The Blackwell Guide to the Philosophy of Computing and Information*, Blackwell Publishing Ltd, 2008, ISBN 978-0-470-75701-7, s. 153–166, doi:10.1002/9780470757017.ch11.

[5] MARGOLIS, Eric a Stephen LAURENCE: Concepts. In *The Stanford Encyclopedia of Philosophy*, editace E. N. Zalta, Metaphysics Research Lab, Stanford University, 2014. Dostupné z: https://plato.stanford.edu/archives/spr2014/entries/concepts/

[6] JACKSON, Daniel: *Software Abstractions: Logic, Language, and Analysis*. Cambridge, MA, USA: MIT Press, 2006, ISBN 978-0-262-10114-1.

[7] ROBINSON, Stewart et al.: Conceptual modeling: Definition, purpose and benefits. In *2015 Winter Simulation Conference (WSC)*, Huntington Beach, CA, USA: IEEE, 2015, ISBN 978-1-4673-9743-8, ISSN 1558-4305, s. 2812–2826, doi:10.1109/WSC.2015.7408386.

[8] OLIVÉ, Antoni: *Conceptual Modeling of Information Systems*. Berlin: Springer, 2007, ISBN 978-3-540-39389-4, doi:10.1007/978-3-540-39390-0.

[9] GUARINO, Nicola: *Formal Ontology in Information Systems: Proceedings of the 1st International Conference June 6-8, 1998, Trento, Italy*. Amsterdam: IOS Press, 1998, ISBN 90-5199-399-4.

[10] SINCLAIR, H.: Conceptualization and Awareness in Piaget's Theory and Its Relevance to the Child's Conception of Language. In *The Child's Conception of Language*, Berlin: Springer, 1978, ISBN 978-3-642-67155-5, s. 191–200, doi:10.1007/978-3-642-67155-5_10.

[11] WATSON, John B. a Rosalie RAYNER: Conditioned Emotional Reaction. *Journal of Experimental Psychology*, , č. 3, 1920: s. 1–14. Dostupné z: http://psychclassics.yorku.ca/Watson/emotion.htm

[12] MADSEN, Ole Lehrmann: Open Issues in Object-oriented Programming – a Scandinavian Perspective. *Software – Practice & Experience*, ročník 25, č. S4, Prosinec 1995: s. 3–43, ISSN 0038-0644, doi:10.1002/spe.4380251303.

[13] MORRIS, Charles W.: *Foundations of the theory of signs*. International encyclopedia of unified science, Chicago: University of Chicago Press, 1938, ISBN 978-0-226-57577-3.

[14] ULLMANN, Stephen: *Semantics: An Introduction to the Science of Meaning*. New York: Barnes & Noble, páté vydání, Březen 1979, ISBN 978-0-06-497076-1.

[15] MACEK, Ondřej a Karel RICHTA: The BPM to UML activity diagram transformation using XSLT. In *Dateso*, ročník 9, 2009, s. 119–129.

[16] BRAGA, Bernardo F. B. et al.: Transforming OntoUML into Alloy: towards conceptual model validation using a lightweight formal method. *Innovations in Systems and Software Engineering*, ročník 6, č. 1, 2010: s. 55–63, ISSN 1614-5054, doi:10.1007/s11334-009-0120-5.

[17] ALAGAR, V. S. a K. PERIYASAMY: *Specification of Software Systems*. London: Springer, druhé vydání, 2011, ISBN 978-0-85729-276-6.

[18] HENDERSON-SELLERS, Brian: *On the Mathematics of Modelling, Metamodelling, Ontologies and Modelling Languages*. Berlin: Springer, 2012, ISBN 978-3-642-29824-0.

[19] LEWIS, William E.: *Software Testing and Continuous Quality Improvement, Third Edition*. Boston, MA, USA: Auerbach Publications, druhé vydání, 2008, ISBN 978-1-4200-8073-5.

[20] ALAM, M. Daud a Uwe F. GÜHL: *Project Management in Practice*. Berlin: Springer, 2016, ISBN 978-3-662-52944-7, doi:10.1007/978-3-662-52944-7.

[21] DRAGAN Djurić: The Tao of Modeling Spaces. *Journal of Object Technology*, ročník 5, 2006: s. 125–147. Dostupné z: http://www.sfu.ca/~dgasevic/papers/The_Tao_of_Modeling_Spaces.pdf

[22] ROBINSON, Stewart: Simulation Model Verification and Validation: Increasing the Users' Confidence. In *Proceedings of the 29th Conference on Winter Simulation*, WSC '97, Washington, DC, USA: IEEE Computer Society, 1997, ISBN 0-7803-4278-X, s. 53–59, doi:10.1145/268437.268448.

[23] ROBINSON, Stewart: Simulation Verification, Validation and Confidence: A Tutorial. *Transactions of the Society for Computer Simulation International*, ročník 16, č. 2, Červen 1999: s. 63–69, ISSN 0740-6797.

[24] THACKER, Ben H. et al.: Concepts of Model Verification and Validation. Technická zpráva, Los Alamos National Laboratory, Los Alamos, 2004. Dostupné z: https://www.osti.gov/scitech/servlets/purl/835920/

[25] ZOU, Yang et al.: An Overview of Conceptual Model for Simulation. In *Theory, Methodology, Tools and Applications for Modeling and Simulation of Complex Systems: 16th Asia Simulation Conference and SCS Autumn Simulation Multi-Conference, AsiaSim/SCS AutumnSim 2016, Beijing, China, October 8-11, 2016, Proceedings, Part I*, Singapore: Springer, 2016, ISBN 978-981-10-2663-8, s. 96–100, doi:10.1007/978-981-10-2663-8_10.

[26] LIDDLE, Stephen W.: Model-Driven Software Development. In *Handbook of Conceptual Modeling: Theory, Practice, and Research Challenges*, Berlin: Springer, 2011, ISBN 978-3-642-15865-0, s. 17–56, doi:10.1007/978-3-642-15865-0_2.

[27] ELMASRI, Ramez a Shamkant NAVATHE: *Fundamentals of Database Systems*. Addison-Wesley, 6. vydání, 2010, ISBN 978-0-13-608620-8.

[28] TRUYEN, Frank: The Fast Guide to Model Driven Architecture - The basics of Model Driven Architecture. [online], 2006. Dostupné z: http://www.omg.org/mda/mda_files/Cephas_MDA_Fast_Guide.pdf

[29] ONGGO, Stephan: Methods for Conceptual Model Representation. In *Conceptual Modeling for Discrete-Event Simulation*, CRC Press, 2010, ISBN 978-1-4398-1037-8, s. 337—-354, doi:10.1201/9781439810385-c13.

[30] ZÄSCHKE, Tilmann et al.: Improving Conceptual Data Models Through Iterative Development. *Data & Knowledge Engineering*, ročník 98, č. C, Červenec 2015: s. 54–73, ISSN 0169-023X, doi:10.1016/j.datak.2015.07.005.

[31] MYLOPOULOS, John: Information Modeling in the Time of the Revolution. *Information Systems*, ročník 23, 1998: s. 127–155, ISSN 0306-4379.

[32] CHEN, Peter Pin-Shan: The Entity-relationship Model: Toward a Unified View of Data. *ACM Transactions on Database Systems (TODS)*, ročník 1, č. 1, Březen 1976: s. 9–36, ISSN 0362-5915, doi:10.1145/320434.320440.

[33] GOGOLLA, Martin: UML and OCL in Conceptual Modeling. In *Handbook of Conceptual Modeling: Theory, Practice, and Research Challenges*, Berlin: Springer, 2011, ISBN 978-3-642-15865-0, s. 85–122, doi:10.1007/978-3-642-15865-0_4.

[34] GUIZZARDI, Giancarlo, Gerd WAGNER a Marten VAN SINDEREN: A Formal Theory of Conceptual Modeling Universals. [online], 2004. Dostupné z: `http://doc.utwente.nl/49866/1/WPI.pdf`

[35] CARRARETTO Roberto: A Modeling Infrastructure for OntoUML. [online], Červenec 2010. Dostupné z: `https://nemo.inf.ufes.br/wp-content/papercite-data/pdf/a_modeling_infrastructure_for_ontouml_2010.pdf`

[36] JONES, Cliff B.: *Systematic Software Development Using VDM*. Upper Saddle River, NJ, USA: Prentice-Hall, druhé vydání, 1990, ISBN 978-0-13-880733-7.

[37] JACKY, Jonathan: *The Way of Z: Practical Programming with Formal Methods*. New York, NY, USA: Cambridge University Press, 1996, ISBN 978-0-521-55976-8.

[38] SCHNEIDER, Steve: *The B-method: An Introduction*. Cornerstones of Computing, London: Macmillan Education UK, Říjen 2001, ISBN 978-0-33379-284-1.

[39] DUMAS, Marlon et al.: *Fundamentals of Business Process Management*. Berlin: Springer, 2013, ISBN 978-3-642-33142-8.

[40] DIETZ, Jan L.G.: DEMO: Towards a discipline of organisation engineering. *European Journal of Operational Research*, ročník 128, 2001: s. 351–363, ISSN 0377-2217, doi:10.1016/S0377-2217(00)00077-1.

[41] MASCARDI, Viviana, Valentina CORDÌ a Paolo ROSSO: A comparison of upper ontologies. Technická zpráva, Università degli Studi di Genova, 2006, [online]. Dostupné z: `http://users.dsic.upv.es/~prosso/resources/MascardiEtAl_WOA07.pdf`

[42] BEISSWANGER, Elena et el.: BioTop: An Upper Domain Ontology for the Life Sciences. *Applied Ontology - Towards a Metaontology for the Biomedical Domain*, ročník 3, č. 4, 2008: s. 205–212, ISSN 1570-5838.

[43] MANNAERT, Herwig, Jan VERELST a Peter DE BRUYN: *Normalized Systems Theory: From Foundations for Evolvable Software Toward a General Theory for Evolvable Design*. Kermt (Belgie): Koppa, 2016, ISBN 978-90-77160-091.

[44] PASTOR, Oscar a Juan Carlos MOLINA: *Model-Driven Architecture in Practice: A Software Production Environment Based on Conceptual Modeling*. New York: Springer, 2007, ISBN 978-3-540-71867-3.

[45] FOWLER, Martin: *Domain Specific Languages*. Boston, MA, USA: Addison-Wesley Professional, 2010, ISBN 978-0-321-71294-3.

[46] GHOSH, Debasish: *DSLs in Action*. Greenwich, CT, USA: Manning Publications Co., 2010, ISBN 978-1-935182-45-0.

[47] EMBLEY, David W., Stephen W. LIDDLE a Oscar PASTOR: *Conceptual-Model Programming: A Manifesto*. Berlin: Springer, 2011, ISBN 978-3-642-15865-0, s. 3–16, doi:10.1007/978-3-642-15865-0_1.

[48] HALPIN, Terry: *Object-Role Modeling Fundamentals: A Practical Guide to Data Modeling with ORM*. Basking Ridge, NJ, USA: Technics Publications, LLC, 2015, ISBN 978-1-63462-074-1.

[49] O'REGAN, Gerard: *Mathematical Approaches to Software Quality*. London: Springer, 2011, ISBN 978-1-84996-564-4.

[50] OBJECT MANAGEMENT GROUP: Information technology - Object Management Group Unified Modeling Language (OMG UML), Infrastructure, v. 2.4.1. Technická zpráva, Duben 2012. Dostupné z: `http://www.omg.org/spec/UML/ISO/19505-1/PDF`

[51] OBJECT MANAGEMENT GROUP: OMG Unified Modeling Language, v. 2.5. Technická zpráva, Březen 2015. Dostupné z: `http://www.omg.org/spec/UML/2.5/PDF`

[52] STARRETT, Cortland: xtUML: Current and Next State of a Modeling Dialect. In *Proceedings of the 2nd International Workshop on Executable Modeling co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MODELS 2016), Saint-Malo, France, October 3, 2016.*, 2016, s. 33–37. Dostupné z: `http://ceur-ws.org/Vol-1760/paper5.pdf`

[53] GUERMEZI, Sahar et al.: Executable Modeling with fUML and Alf in Papyrus: Tooling and Experiments. In *Proceedings of the 1st International Workshop on Executable Modeling co-located with ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS 2015), Ottawa, Canada, September 27, 2015.*, 2015, s. 3–8. Dostupné z: http://ceur-ws.org/Vol-1560/paper1.pdf

[54] FRIEDENTHAL, Sanford, Alan MOORE a Rick STEINER: *A Practical Guide to SysML: The Systems Modeling Language*. San Francisco, CA, USA: Morgan Kaufmann, 2009, ISBN 978-0-12-378607-4.

[55] VAN RENSSEN, Andries: *Gellish: A Generic Extensible Ontological Language*. Delft: Delft University Press, 2005, ISBN 978-90-407-2597-5.

[56] BOULANGER, Jean-Louis: *Formal methods applied to complex systems: implementation of the B method*. London: Wiley-ISTE, 2014, ISBN 978-1-84821-709-6.

[57] OBJECT MANAGEMENT GROUP: Object Constraint Language, v. 2.4. Technická zpráva, Únor 2014. Dostupné z: http://www.omg.org/spec/OCL/2.4/PDF

[58] SPIVEY, J. Michael: *The Z notation: a reference manual*. Englewood Cliffs, NJ, USA: Prentice Hall, druhé vydání, 1992, ISBN 978-0-13978-529-0.

[59] MALIK Petra et al.: CZT: Community Z Tools. [online], [cit. 2017-02-28]. Dostupné z: http://czt.sourceforge.net

[60] OVERTURE CONTRIBUTORS et al.: Overture Tools, v. 2.4.6. [software], [cit. 2017-03-03]. Dostupné z: http://overturetool.org

[61] SPARX SYSTEMS: Enterprise Architect, v. 12. [software], [cit. 2017-03-05]. Dostupné z: http://www.sparxsystems.com

[62] SPARX SYSTEMS: Enterprise Architect User Guide. [online], [cit. 2017-03-06]. Dostupné z: http://www.sparxsystems.com.au/bin/EAUserGuide.pdf

[63] KHALEK, Shadi Abdul et al.: TestEra: A tool for testing Java programs using alloy specifications. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, Washington, DC, USA: IEEE Computer Society, 2011, ISBN 978-1-4577-1638-6, s. 608–611. Dostupné z: http://mir.cs.illinois.edu/marinov/publications/KhalekETAL11TestEraDemo.pdf

[64] MARINOV, Darko a Sarfraz KHURSHID: TestEra: A Novel Framework for Automated Testing of Java Programs. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE 2001)*, Washington, DC, USA: IEEE Computer Society, 2001, ISSN 1527-1366, s. 22–31.

[65] MENTHOR: Menthor Editor, v. 1.1.8. [software], [cit. 2017-03-11]. Dostupné z: `http://www.menthor.net/menthor-editor.html`

[66] BUETTNER, Fabian et al.: USE: UML-based Specification Environment, v. 4.2.0. [software], [cit. 2016-03-17]. Dostupné z: `http://useocl.sourceforge.net/w/`

[67] GABBRIELLI, Maurizio a Simone MARTINI: *Programming Languages: Principles and Paradigms*. London: Springer, 2010, ISBN 978-1-84882-913-8.

[68] PIERCE, Benjamin C.: *Types and Programming Languages*. Cambridge, MA, USA: MIT Press, 2002, ISBN 978-0-262-16209-8.

[69] MILNER, Robin: A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, ročník 17, 1978: s. 348–375.

[70] ALLEN, Christopher a Julie MORONUKI: *Haskell Programming: From First Principle*. 2016, [online]. Dostupné z: `http://haskellbook.com/`

[71] BRADY, Edwin: Programming and Reasoning with Algebraic Effects and Dependent Types. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, New York, NY, USA: ACM, 2013, ISBN 978-1-4503-2326-0, s. 133–144, doi: 10.1145/2500365.2500581.

[72] McKINNA, James: Why Dependent Types Matter. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, New York, NY, USA: ACM, 2006, ISBN 1-59593-027-2, s. 1–1, doi:10.1145/1111037.1111038.

[73] MIAO, Weiyu a Jeremy SIEK: Compile-time Reflection and Metaprogramming for Java. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*, PEPM '14, New York, NY, USA: ACM, 2014, ISBN 978-1-4503-2619-3, s. 27–37, doi: 10.1145/2543728.2543739.

[74] WYMORE, A. Wayne: *Model-Based Systems Engineering*. Boca Raton, FL, USA: CRC Press, 1993, ISBN 978-0-8493-8012-9.

[75] CARDELLI, Luca a Peter WEGNER: On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys (CSUR)*, ročník 17, č. 4, Prosinec 1985: s. 471–523, ISSN 0360-0300, doi:10.1145/6041.6042.

[76] GUARINO, Nicola a Giancarlo GUIZZARDI: "We Need to Discuss the Relationship": Revisiting Relationships as Modeling Constructs. In *Advanced Information Systems Engineering: 27th International Conference, CAiSE 2015, Stockholm, Sweden, June 8-12, 2015, Proceedings*, London: Springer, 2015, ISBN 978-3-319-19069-3, s. 279–294, doi:10.1007/978-3-319-19069-3_18.

[77] JONES, Simon Peyton et al.: Haskell 98 Language and Libraries - The Revised Report. 2002, [online], [cit 2017-04-15]. Dostupné z: https://www.haskell.org/definition/haskell98-report.pdf

[78] MARLOW, Simon et al.: Haskell 2010 - Language Report. 2010, [online], [cit 2017-04-15]. Dostupné z: https://www.haskell.org/definition/haskell2010.pdf

[79] GHC TEAM: Glasgow Haskell Compiler Users Guide. 2016, [online], [cit. 2017-04-18]. Dostupné z: https://downloads.haskell.org/~ghc/latest/docs/users_guide.pdf

[80] HUDAK, Paul et al.: A History of Haskell: Being Lazy With Class. 2007, [online], [cit 2017-04-17]. Dostupné z: http://research.microsoft.com/en-us/um/people/simonpj/papers/history-of-haskell/history.pdf

[81] POSTHUMA, J. S.: Expressing ontologies using a functional language. 2010, [online], [cit 2017-04-18]. Dostupné z: http://referaat.cs.utwente.nl/conference/12/paper/7000/expressing-ontologies-using-a-functional-language.pdf

[82] KUHN, Werner: *A Functional Ontology of Observation and Measurement*. Berlin: Springer, 2009, ISBN 978-3-642-10436-7, s. 26–43, doi:10.1007/978-3-642-10436-7_3.

[83] VAZOU, Niki: *Liquid Haskell: Haskell as a Theorem Prover*. 2016, [online], [cit. 2017-04-19]. Dostupné z: http://goto.ucsd.edu/~nvazou/thesis/main.pdf

[84] MARLOW, Simon, Sven PANNE a Noel WINSTANLEY: haskell-src: Support for manipulating Haskell source code. 2017, [software]. Dostupné z: http://hackage.haskell.org/package/haskell-src

[85] GANSNER, Emden, Eleftherios KOUTSOFIOS a Stephen NORTH: Drawing graphs with dot. 2006, [online], [cit. 2017-04-21]. Dostupné z: http://www.graphviz.org/Documentation/dotguide.pdf

Příloha **A**

Seznam použitých zkratek

ALF Action Language for Foundational UML. 24, 25

AMN Abstract Machine Notation. 28

API Application Programming Interface. 36, 51

ASCII American Standard Code for Information Interchange. 30, 37

BFO Basic Formal Ontology. 17

BPEL Business Process Execution Language. 16

BPMN Business Process Model and Notation. 9, 16, 35

CASE Computer-aided software engineering. 28, 39, 105, 107

CDM Compiler-Driven Modelling. 79

CIM Computation Independent Model. 50

CMP Conceptual-Model Programming. 19, 51

CRUD Create, Read, Update, Delete. 49

DEMO Design & Engineering Methodology for Organizations. 16, 23

DIMACS The Center for Discrete Mathematics and Theoretical Computer Science. 34

DOLCE Descriptive Ontology for Linguistic and Cognitive Engineering. 17

DRY Don't repeat yourself. 48

DSL Domain-specific language. 10, 18, 47, 64, 65, 67, 69, 103

E²R Enhanced entity–relationship (model). 15

EER Enhanced entity–relationship (model). 15, 22, 23, 83

ER Entity-relationship (model). 15, 22–24, 54, 83, 102

fUML Foundational UML. 24, 25

GFO General Formal Ontology. 17

GHC Glasgow Haskell Compiler. 59, 60, 65, 66, 72, 73, 78, 79, 94, 103

GUI Graphical User Interface. 39, 83, 107

HTML HyperText Markup Language. 83

IDE Integrated Development Environment. 34, 37, 64, 103

ISO International Organization for Standardization. 23, 30

IČ Identifikační číslo osoby. 88

MBE Model-Based Engineering. 49

MDA Model-Driven Architecture. 19, 24, 49, 50

MDD Model-Driven Development. 24, 35, 49, 79

MDE Model-Driven Engineering. 49, 51

MIT Massachusetts Institute of Technology. 16, 26

MOF Meta-Object Facility. 11

OCL Object Constraint Language. 15, 16, 21, 26, 28, 29, 35, 36, 38, 39

OLED OntoUML lightweight editor. 38

OMG Object Management Group. 23, 24, 28, 49

OOP Objektově orientované programování. 30, 46

ORM Object Role Modeling. 23, 54

OWL Web Ontology Language. 17, 35, 38

PDCA Plan, Do, Act, Check. 10

PIM Platform Independent Model. 50

PSM Platform Specific Model. 50

RDF Resource Description Framework. 17, 35, 38

REPL Read–Eval–Print Loop. 64, 72, 79, 80, 82

SAT (Boolean) Satisfiability. 34, 37

SBVR Semantics Of Business Vocabulary And Business Rules. 38

SoC Separation of concerns. 48, 49

SPZ Státní poznávací značka. 88

SUMO Suggested Upper Merged Ontology. 17

SVG Scalable Vector Graphics. 98

UFO Unified Foundational Ontology. 15, 17, 25, 108

UML Unified Modeling Language. 9, 11, 15, 16, 21, 23–25, 28, 35, 38, 39, 57, 83, 102, 108, 119–121

USE UML-based Specification Environment. 39

VDM Vienna Development Method. 29, 34, 35, 121

VDM-RT VDM Real-Time. 34

VDM-SL VDM Specification Language. 16, 30, 34

VIN Vehicle identification number. 88

XMI XML Metadata Interchange. 22, 35

XML Extensible Markup Language. 22, 34

xtUML Executable UML. 24

Příloha **B**

Obsah přiloženého CD

- `README.txt` ..... stručný popis obsahu CD
- `src/`
  - `hCM/` ..... zdrojové kódy systému (Haskell balíček)
  - `hCM_CaseStudy/` ..... případová studie (Haskell balíček)
  - `thesis/` ..... zdrojová forma práce ve formátu LaTeX
    - `pics/` ..... obrázky
    - `code/` ..... ukázky zdrojových kódů
    - `thesis.tex` ..... samotný text práce (LaTeX)
    - `glossary.tex` ..... zkratky (LaTeX)
    - `sources.bib` ..... literatura (BibTeX)
- `text/` ..... text práce ve formátu PDF
  - `thesis.pdf`
  - `thesis_links.pdf` ..... text se zvýrazněnými odkazy