VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

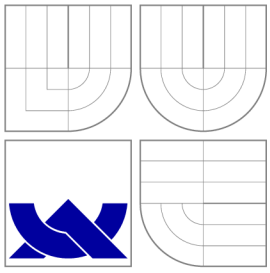
NÁSTROJ PRO NÁVRH ČIPU V UML

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

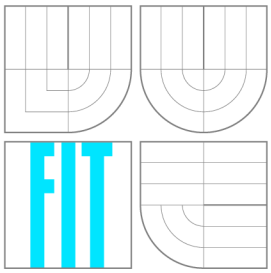
AUTOR PRÁCE
AUTHOR

Bc. PAVOL SRNA

BRNO 2010



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

NÁSTROJ PRO NÁVRH ČIPU V UML

TOOL FOR CHIP DESIGN IN UML

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. PAVOL SRNA

VEDOUCÍ PRÁCE
SUPERVISOR

Prof. Ing. TOMÁŠ HRUŠKA, CSc.

BRNO 2010

Abstrakt

Táto práca sa zaoberá vytvorením nástroja pre návrh čipu v UML. Cieľom je predstaviť novinky v jazyku UML verzie 2.0, ktoré je možné použiť pre modelovanie vstavaných systémov. Následne rozoberá možnosti a spôsob modelovania v prostredí Eclipse a sústredí sa na Eclipse Modeling Framework. Práca vysvetľuje princíp vytvárania grafických editorov s využitím frameworku GMF, pretože ho vyvíjaný nástroj plne využíva. Nakoniec je diskutované zvolené riešenie.

Abstract

This paper deals with the creation of the tool for chip design in UML. The intention of this work is to present the news in the UML language version 2.0, that can be possibly used for modeling of embedded systems. Furthermore, it deals with the possibility and method of modeling in the Eclipse environment and it focuses on the Eclipse Modeling Framework. This work explains the principle of developing of graphical editors based on GMF used fully by developing tool. Finally, it discusses the chosen solution.

Kľúčové slová

Zásuvný modul do eclipse, editor, UML, Eclipse Modeling Framework (EMF), Graphical Modeling Framework (GMF).

Keywords

Eclipse plug-in, editor, UML, Eclipse Modeling Framework (EMF), Graphical Modeling Framework (GMF).

Citácia

Pavol Srna: Nástroj pro návrh čipu v UML, diplomová práce, Brno, FIT VUT v Brně, 2010

Nástroj pro návrh čipu v UML

Prehlásenie

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením Prof. Ing. Tomáša Hrušky CSc.

.....
Pavol Srna
25. mája 2010

Poďakovanie

Ďakujem Ing. Karlovi Masaříkovi, Ph.D. za ochotu a čas pri odborných konzultáciách a za jeho cenné rady a pripomienky pri písaní tejto práce.

Obsah

Kapitola 1

Úvod

Vstavané systémy nájdeme takmer v každom aspekte nášho života: nachádzajú sa vo väčšine domácností, v automobiloch, v rôznych elektronických prístrojoch, priemyselných zariadeniach a nástrojoch. Produkcia vstavaných systémov je oproti produkcii osobných počítačov niekoľko tisíckrát vyššia. Nároky kladené na vstavané systémy sú zákonite striktne vymedzené. Jedná sa hlavne o nízku cenu, malú veľkosť a nízku spotrebu. Na výrobu vstavaných systémov sa v súčasnosti najčastejšie používa technológia na čipe. Táto technológia umožňuje integráciu niekoľkých aplikačne špecifických inštrukčných procesorov (ASIP) a pamätí do jedného čipu. Procesory tvoria jadrá vstavaných systémov. Dôležitým faktorom je znižovanie času, ktorý je potrebný pre návrh a konštrukciu architektúry. Jazyky pre popis architektúry (ADL) dokážu popísať architektúru programovo a automaticky vygenerovať potrebné nástroje. Výrazne tak skracujú čas potrebný pre návrh a konštrukciu architektúry. Trendom súčasnosti je rychlé prototypovanie vstavaných systémov v jazyku UML, teda rýchly prechod od špecifikácie k realizácii.

Táto práca sa zaoberá vytvorením nástroja pre návrh čipu v UML, ktorý umožní zvýšiť produktivitu návrhu počítačovej architektúry. ISAC je jazyk, ktorý sa používa v rámci projektu Lissom [1] na popis architektúry vstavaného systému a následné vygenerovanie softwarových nástrojov. Cieľom práce je predstaviť koncept grafického popisného jazyka založeného na jazyku UML a navhrnúť transformáciu z grafickej reprezentácie do vhodnej textovej reprezentácie. Jadro práce sa sústredí na vývoj nástroja, ktorý na báze navrhnutého grafického popisného jazyka modeluje počítačovú architektúru, pričom odpovedajúca textová reprezentácia musí umožniť transformáciu na zdrojové súbory jazyka ISAC.

Práca sa člení na niekoľko kapitol. Úvodné kapitoly vysvetľujú základné novinky jazyka UML 2.0, ktoré umožňujú modelovať vstavané systémy. Kapitola 2.4 predstavuje grafický popisný jazyk. Kapitoly 3 a 4 sa zaoberajú kľúčovými vlastnosťami Eclipse frameworkov, na ktorých je vyvíjaný nástroj postavený. Návrh riešenia je popísaný v kapitole 5. Spôsob implementácie opisuje kapitola 6. V záverečných kapitolách je predstavený výsledný nástroj a sú zhodnotené dosiahnuté výsledky práce.

[1] Stránky projektu Lissom: http://merlin.fit.vutbr.cz/Lissom/

Kapitola 2

SoC a UML

Posledných niekoľko rokov pozorujeme výrazný pokrok v dvoch oblastiach súvisiacich s návrhom hardware a software pre elektronické zariadenia. Prvým z nich je rýchly rast návrhu komplexného zariadenia na čipe (SoC), druhým je pokrok v pridávaní nových možností do jazyka UML pre lepšiu podporu návrhu vstavaných systémov a systémov pracujúcich v reálnom čase. Väčšina systémov na čipe je určených pre vstavané zariadenia. Trendom súčasnosti je nárast komplexnosti systémov na čipe, ktoré mnohokrát obsahujú viacero procesorov. V takomto prípade sa hovorí o multiprocesorových systémoch na čipe (MPSoC) [10]. S nárastom komplexnosti systémov úzko súvisí zložitosť návrhu, ktorá prirodzene tiež narastá. Preto je snaha vyvíjať grafické nástroje, ktoré návrhárom umožnia rýchly prechod od špecifikácie k realizácií. Jazyk UML vo verzii 2.0 prináša oproti verzii 1.5 niekoľko podstatných noviniek, ktoré výrazne zlepšujú modelovanie systémov na čipe.

2.1 UML 2.0

Unifikovaný modelovací jazyk (UML) 2.0 je štandard asociácie Object Managment Group, ktorý sa podľa [10] rozdeľuje na 4 základné časti:

- UML SuperStructure - popis UML z pohľadu užívateľa,
- UML InfraStructure - metamodel jazyka UML,
- UML Object Constraint Language (OCL) - jazyk pre špecifikáciu vstupných/výstupných podmienok a invariantov v jednotlivých diagramoch,
- UML Diagram Interchange - popis XML štruktúr na výmenu dát medzi jednotlivými modelmi.

Verzia UML 2.0 bola navrhnutá tak, aby umožnila postupné zavedenie modelom riadených metód[1]. Jednou z najvýznamnejších štrukturálnych zmien v UML superstructure je definícia komponentov a ich portov. Komponenty a porty reprezentujú stavebné bloky, ktoré zapúzdrujú štruktúru a správanie. Podstatným vylepšením v definovaní správania predstavujú aktivity a akcie [10].

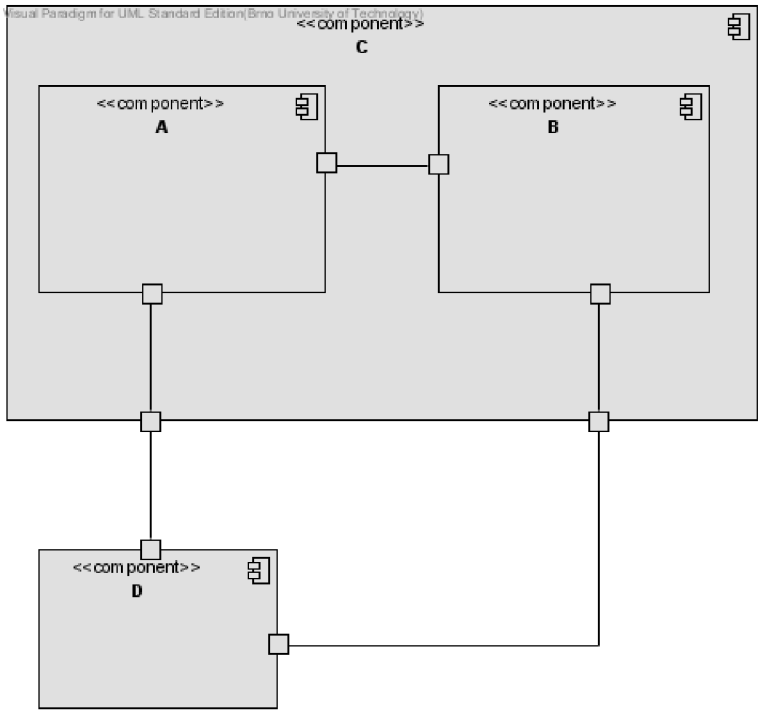
Nasledujúce dve kapitoly poukazujú iba na základné zmeny v štruktúre a vlastnostiach v správaní jazyka UML verzie 2.0 oproti verziám 1.5 a nižším. Komponenty, porty, konektory a stereotypy predstavujú základné stavebné kamene jazyka UMLISAC, ktorému sa venuje

[1] MDD - z anglického Model Driven Development, je princíp vývoja založenom na modeli.

kapitola 2.4. Podrobnejšie informácie o novinkách v jazyku UML verzie 2.0 je možné nájsť v [13].

2.2 Štrukturálne vlastnosti UML

Medzi základne štrukturálne jednotky UML patria triedy s atribútmi, operácie a rozhrania. Jednotlivé komponenty znázorňuje diagram komponentov. Komponenty predstavujú samostatné jednotky s rozhraniami, ktoré je možné pomocou portov prepojiť s inými komponentmi. Komponenty majú vnútornú štruktúru pozostávajúcu z prvkov, ktoré môžu byť komponenty alebo triedy prepojené konektormi väčšinou cez porty. Porty sú komunikačné body, ktoré majú množinu požadovaných a poskytovaných rozhraní a umožňujú skryť detaily implementácie. Tento spôsob dekompozície celkového návrhu systému, alebo štruktúry aplikácie je prirodzenejší pre návrh väčšiny elektronických zariadení [10]. Ukážka diagramu komponentov je na obrázku 2.1. Porty sú vyznačené malými štvorcami a konektory predstavujú čiary medzi jednotlivými portami. Z obrázku je zrejmé, že porty umožňujú komponentom skryť detaily implementácie - tzv. black box modeling. **Stereotypy** sú jedným



Obr. 2.1: Ukážka diagramu komponentov.

z mechanizmov rozšíriteľnosti jazyka UML. Dovoľujú návrhárovi vytvoriť z existujúcich UML prvkov prvky nové, ktoré majú špecifické vlastnosti. Návrhár môže vytvoriť sadu rozličných stereotypov, ktorými označí jednotlivé triedy alebo atribúty. Takýmto neinvazívnym spôsobom dokáže rozlišovať medzi jednotlivými triedami a riešit radu špecifických problémov. Stereotypy nachádzajú uplatnenie pri modelom riadenom vývoji (MDD), pretože pre každý stereotyp je možné generovať špecifický výstup.

2.3 Vlastnosti správania UML

Základný koncept správania v UML kombinuje akcie, aktivity a stavové diagramy. Koncept akcií vychádza z predchádzajúcej práce konsorcia [2] o sémantike akcií a podliehal niekoľkoročným skúsenostiam s generovaním spustiteľného kódu z UML. Vo verzii 2.0 jazyka UML je akcia definovaná ako základná jednotka správania. Akcie môžu používať primitívne funkcie, vyvolávať správanie, posielať a prijímať signály, a dokonca modifikovať štruktúry [10].

Aktivity sú reprezentované pomocou diagramov aktivít. Aktivita je orientovaný graf, ktorý opisuje sled akcií v rámci správania. Akcia je atomická jednotka správania. Aktivita môže obsahovať okrem akcií aj iné typy uzlov. Riadiace uzly (control nodes) sa používajú, ak v aktivite existuje niekoľko možných ciest. Riadiace uzly reprezentujú rozhodovacie a spojovacie body, čím zabezpečujú vetvenie a spájanie súbežných tokov. Hrany v rámci aktivity prenášajú objekty alebo hodnoty medzi jednotlivými uzlami. Je teda možné povedať, že základný model aktivity sa sústredí na objekty, alebo hodnoty tečúce cez uzly po hranách, ktoré sú modifikované alebo transformované v akciách a smerované v riadiacich uzloch. Ďalším typom uzlov sú tzv. objektové uzly (objects nodes), ktoré označujú inštanciu konkrétneho klasifikátora v konkrétnom stave. Obsahujú hornú medzu, ktorá udáva kapacitu ich vyrovnávacej pamäte. Existujú 2 špeciálne typy objektových uzlov: piny a uzly parametrov aktivít. Piny sa používajú na prepojenie vstupných a výstupných tokov objektov s akciami. Uzly parametrov reprezentujú parametre, ktoré sa predávajú do/z aktivity [10].

Sémantika vykonávania aktivity je definovaná na základe tokenov tečúcich cez aktivitu. Inými slovami, tokeny prenášajú objekty alebo hodnoty po hranách medzi uzlami. Pohyb tokenov môže predstavovať riadiaci tok systému, podobne ako to je v Petriho sieťach. Akcia sa vykoná len vtedy, ak sú všetky tokeny prítomné na všetkých vstupoch akcie [10].

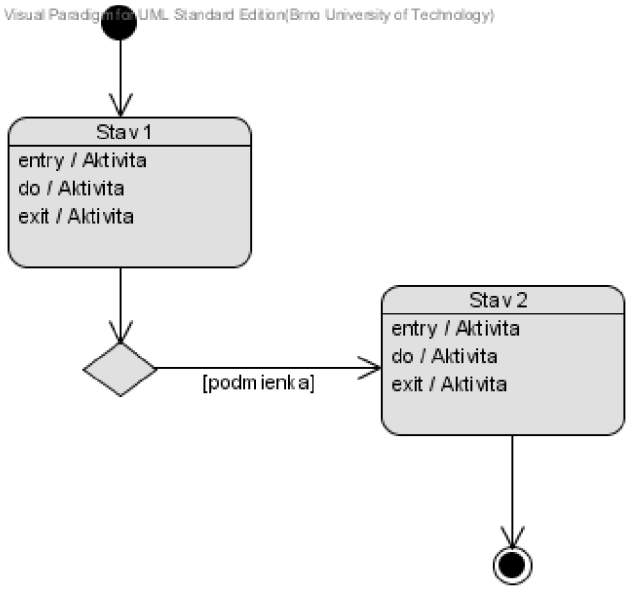
Diagram aktivít modeluje správanie na základe toku dát. Oproti tomu stavový diagram je stavovo orientovaný. Stavové diagramy opisujú správanie tried, zobrazujú stavy entít a ich reakcie na rozličné udalosti zmenou z jedného stavu do stavu druhého. Na obrázku 2.2 je znázornený stavový diagram. Prechody medzi stavmi sú znázornené šípkou. Podmienka je zapísaná pomocou hranatých zátvoriek. Stavové automaty vyvolávajú aktivity pri spracovávaní stavov alebo prechodov. Existuje niekoľko druhov aktivít napríklad: *Entry*, *Do*, *Exit*, *Effect*, ktoré sú rozdelené podľa toho, kedy sú vyvolané. Napríklad pred vstupom do určitého stavu sa vyvolá aktivita *Entry*, v prípade, že je definovaná, atp.

2.4 UMLISAC

Princíp nástroja, ktorého vývojom sa zaoberá táto práca, je umožniť návrhárovi graficky opísať architektúru mikroprocesoru a vygenerovať zdrojový kód v jazyku ISAC. ISAC je jazyk pre opis architektúry mikroprocesoru, ktorý sa používa v rámci projektu Lissom. Ing. Karel Masařík, Ph.D. opisuje vo svojej dizertačnej práci [11] grafickú reprezentáciu jazyka ISAC založenú na jazyku UML 2.0. Navrhnutý grafický popisný jazyk sa nazýva UMLISAC. Vyvíjaný nástroj bude využívať tento jazyk. Poznatky poskytnuté v tejto kapitole som prevzal z [11].

Jazyk UMLISAC definuje nové modelovacie elementy pomocou rozširujúcich mechanizmov UML a nemodifikuje samotný UML metamodel. UMLISAC využíva základný koncept UML, ktorý obsahuje pravidlá kompozície a prepojenia, abstrakcie, interakcie, zapúzdrenia a rozšíriteľnosti. Ak je tento koncept aplikovaný do domény popisu architektúry, tak potom

[2] Konsorciom je myslená asociácia Object Managment Group.



Obr. 2.2: Ukážka stavového diagramu.

samotná štruktúra mikroprocesoru je definovaná implicitne diagramom tried a chovanie je definované v stavovom diagrame [11].

Architektúra mikroprocesoru sa skladá zo stavebných blokov. Jeden stavebný blok je napríklad aritmeticko-logická jednotka, pamäť atp. Triedy diagramu tried môžeme považovať za akési stavebné bloky. UMLISAC ponúka niekoľko preddefinovaných šablón, ktoré si návrhár vhodným spôsobom parametrizuje. Inštancie šablón sú neskôr použité vo väčších šablonách. Na prepojenie šablón sa využívajú porty a rozhrania. Rozhrania a konektory sú mapované na niektoré typy komunikačných kanálov, ako je napríklad zbernica. Každá trieda je označená stereotypom. Napríklad trieda súborového registra je označená stereotypom «ISACFILEREGISTER». Veľkosť súborového registra je určená atribútom **size**, ktorý je označený stereotypom «ISACSIZE». Súborový register «ISACFILEREGISTER» môže obsahovať len jeden atribút označený stereotypom «ISACSIZE». [11] Stereotypy umožňujú rozlišovať medzi jednotlivými šablónami a predstavujú dobrý predpoklad pre modelom riadený vývoj.

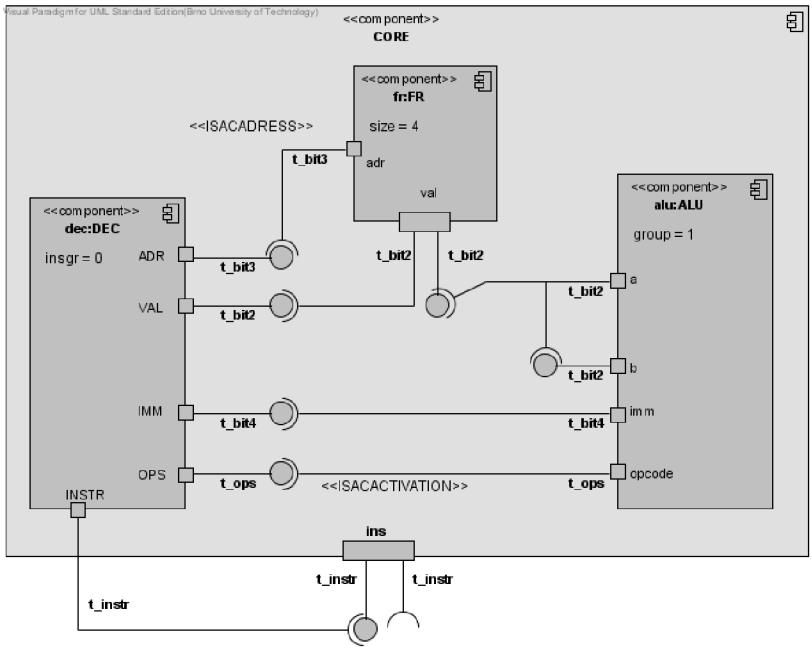
Na obrázku 2.3 je zobrazený príklad jadra mikroprocesoru v jazyku UMLISAC s využitím diagramu komponentov. Jednotlivé komponenty sú inštancie tried a sú navzájom prepojené pomocou portov a rozhraní. Rozhranie, ktoré poskytuje dáta, je označené kolieskom a rozhranie, ktoré vyžaduje dáta, je označené oblúkom. Zobrazené jadro sa skladá z inštrukčného dekodéru DEC, súborového registra FR a aritmeticko logickej jednotky ALU.

Pomocou priamej transformácie je možné previesť štrukturálny model, ktorý je zapísaný v jazyku UMLISAC na zdrojový text jazyka ISAC. Uml trieda označená stereotypom «ISACREGISTER» sa transformuje na zdrojový prvok REGISTER jazyka ISAC. Napríklad inštancia registra FR z obrázka 2.3 je v jazyku ISAC definovaná následovne:

```
REGISTER fr[4];
```

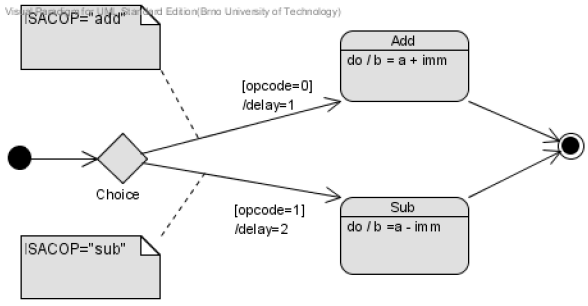
Obdobne by sa tramsformovali aj ostatné triedy jazyka UMLISAC. Trieda označená stereotypom «ISACMEMORY» sa transformuje na zdrojový prvok MEMORY jazyka ISAC a trieda označená stereotypom «ISACFILEREGISTER» sa transformuje na jednodimenzi-

onálne pole prvkov typu REGISTER. Atribút označený stereotypom «ISACSIZE» potom v prvom prípade udáva počet prvkov typu REGISTER v poli a v druhom prípade počet buniek pamäti.



Obr. 2.3: Príklad jadra mikroprocesoru v jazyku UMLISAC [11].

Na obrázku 2.4 je zobrazený stavový diagram, ktorý definuje správanie aritmeticko logickej jednotky (ALU). Správanie ALU rozlišuje dva základné stavy: *add* a *sub*. Prvý prechod definuje operačný kód inštrukcie sčítania, ktorej podoba je v jazyku assemblera **add** a hodnota v strojovom jazyku 0. Druhý prechod analogicky definuje operačný kód inštrukcie odčítania. Delay udáva veľkosť oneskorenia práve vykonávanej inštrukcie (požadovaný čas pre dokončenie každej inštrukcie vo funkčnej jednotke). Napríklad operácia **add** vyžaduje jeden takt pre dokončenie operácie sčítania.

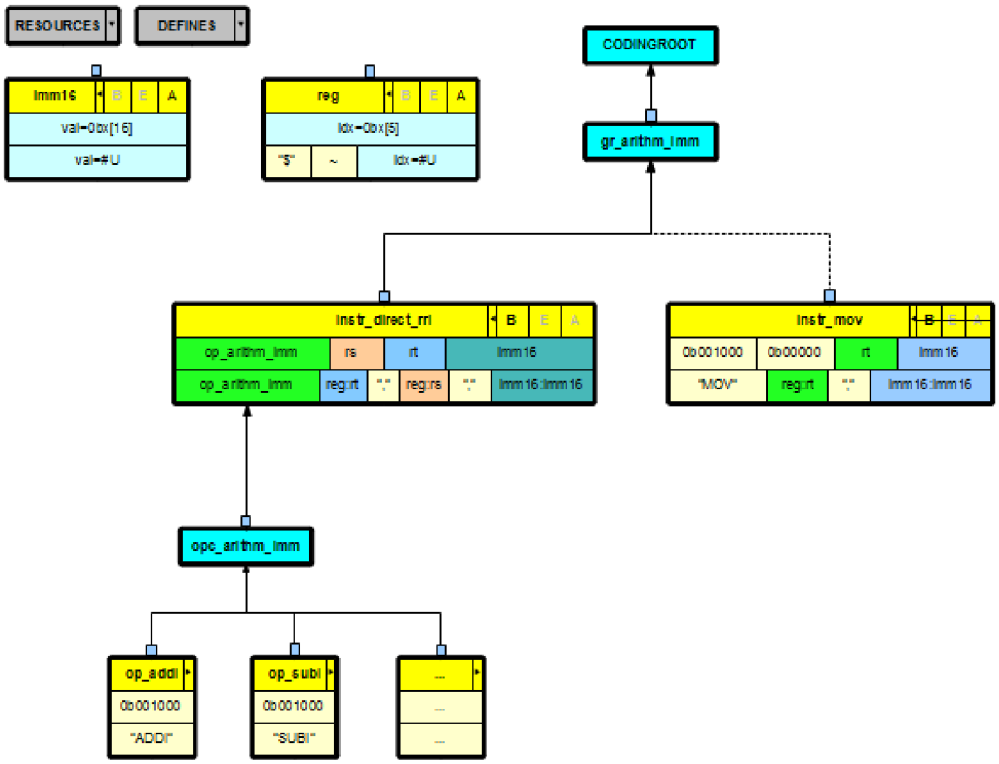


Obr. 2.4: Opis správania aritmeticko logickej jednotky [11].

Správanie jednotlivých komponentov namodelované pomocou diagramov aktivít a stavových diagramov nie je možné do jazyka ISAC transformovať priamo, ale sú potrebné všeobecne zložitejšie transformácie. Opis takýchto transformácií je publikovaný v [11]. Mo-

delovanie správania pomocou diagramov aktivít a stavových diagramov je vhodné hlavne pre jednoduchšie modely CPU. Pri zložitejších modeloch sa stávajú výsledné diagramy neprehľadné, pretože obsahujú veľký počet stavov. Z tohto dôvodu nie je vhodné takéto diagramy používať, pretože pre návrhára neposkytujú dostatočnú úroveň abstrakcie a neprinášajú dostatočné zefektívnenie návrhu vstavaného systému.

Na základe týchto skutočností bol navrhnutý Ing. Adamom Husárom nový diagram pre grafický popis inštrukčnej sady. Tento diagram už je možné používať aj pre zložitejšie modely procesorov, pretože poskytuje vyššiu mieru abstrakcie. Hlavnými stavebnými prvkami tohto diagramu sú operácie a skupiny, ktoré sú znázornené podobne ako triedy v štrukturálnom modeli, pomocou obdĺžnikov. Operácia obsahuje tri základné oddiely: hlavičku s názvom operácie, oddiel CODING a ASSEMBLER. Do oddielu ASSEMBLER a CODING je možné pridávať inštancie iných operácií a skupin, terminály a atribúty. Operácie je možné pridávať do skupin. V diagrame sa to značí pomocou šípky. Typ inštancie je špecifikovaný pomocou šípky vedúcej do tejto inštancie alebo alternatívne pomocou dvojbodky. Notácia s využitím dvojbodky je *ID ´:´ ID*, kde prvé ID je názov inštancie a druhé jej typ. Na obrázku 2.5 je zobrazený príklad diagramu inštrukčnej sady.



Obr. 2.5: Návrh diagramu inštrukčnej sady.

Diagram instrukčnej sady je možné oproti diagramom aktivít a stavovým diagramom transformovať priamo do jazyka ISAC. Ukážka transformácie diagramu z obrázku 2.5 je nasledujúca:

```
OPERATION op_addi {
    ASSEMBLER { "ADDI" };
    CODING { 0b001000 };
    EXPRESSION {OP_ADDI; };
}

OPERATION op_subi {
    ASSEMBLER { "SUBI" };
    CODING { 0b001001 };
    EXPRESSION {OP_SUBI; };
}

GROUP opc_arithm_imm = op_addi, op_subi, .. ;

OPERATION instr_direct_rri
{
    INSTANCE gpr ALIAS {rs, rt};
    INSTANCE op_arithm_imm ALIAS { op_arithm_imm };
    INSTANCE uimm16;

    ASSEMBLER{ op_arithm_imm rt "," rs "," uimm16 };
    CODING   { op_arithm_imm rs rt uimm16 };
    BEHAVIOR    {
                    ...
    };
}
```

2.5 Úvod do jazyka OCL

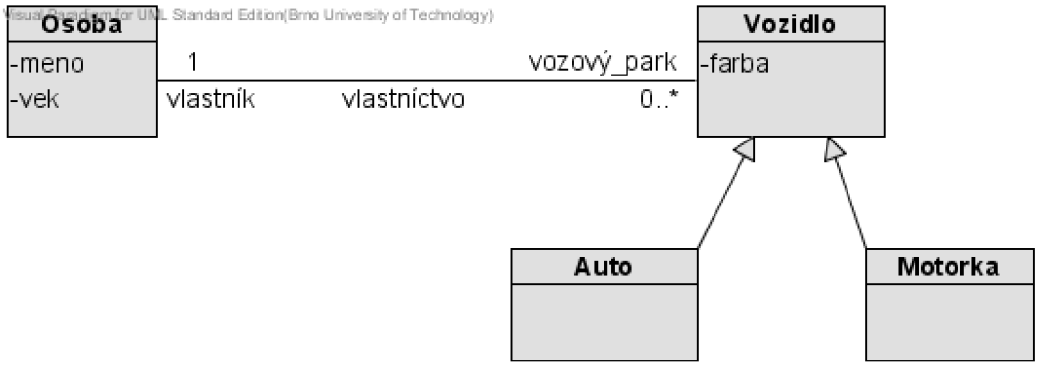
UML je unifikovaný modelovací jazyk, ktorý obsahuje sadu diagramov, pomocou ktorých je možné modelovať vyvíjaný systém. UML diagramy typicky nie sú v špecifikácií natoľko detailné, aby dokázali zachytiť relevantné aspekty modelovaného systému. Preto asociácia Object Management Group vytvorila jazyk OCL, pomocou ktorého je možné v spojení s jazykom UML vytvárať presnejšie modely. Object Constraint Language (OCL) je formálny modelovací jazyk, ktorý sa používa na popísanie výrazov vykonávaných nad UML modelmi. Takéto výrazy typicky špecifikuju nemenné podmienky, ktore musia v systéme, ktorý je modelovaný, neustále platiť a udržovať ho v konzistentnom stave. Pri výrazoch OCL je garantované, že neprichádza pri ich vyhodnocovaní k vedľajším účinkom. To znamená, že ak je daný výraz vyhodnotený - vráti hodnotu, ale nemení model. Object Constraint Language nie je programovací jazyk, preto ho nie je možné použiť na napísanie logiky programu alebo ako mechanizmus na sychronizáciu prenosu dát. Je však jazykom typovým, teda každý výraz v OCL má typ. Napríklad nie je možné porovnávať celočiselnú hodnotu s reťazcom. Každý klasifikátor v modeli UML predstavuje presný typ jazyka OCL. Zoznam preddefinovaných typov je možné nájsť v [1].

Jazyk OCL môže byť použitý na niekoľko rôznych účelov [1]:

- ako dotazovací jazyk,
- na definíciu invariantov tried a typov v diagrame tried,
- na určenie invariantov pre stereotypy,
- na popis *pre* a *post* podmienok operácií a metód,
- na určenie obmedzujúcich podmienok operácií,
- na určenie derivačných pravidiel pre atribúty ktorýchkoľvek výrazov v modeli UML.

2.6 Základy jazyka OCL

Jazyk OCL je úzko prepojený s modelovacím jazykom UML, to znamená, že každá trieda, rozhranie alebo typ v jazyku UML sa automaticky stáva typom v jazyku OCL. Základnými stavebnými prvkami výrazov v jazyku OCL sú objekty a vlastnosti objektov. Každý OCL výraz je napísaný v kontexte inštancie špecifického typu. Rezervované kľúčové slovo *self* slúži na získanie inštancie objektu v danom kontexte. Napríklad ak je kontext "Automobil", tak potom *self* odkazuje na inštanciu "Automobil". Na obrázku 2.6 je zobrazený triviálny diagram tried. V prípade, že je medzi triedami vytvorená asociácia a jednotlivé triedy majú priradené role (v obrázku 2.6 rola *vlastník* a *vozový_park*), tak potom OCL umožňuje jednoduchú navigáciu pomocou operátora "." medzi takýmito triedami.



Obr. 2.6: Jednoduchý diagram tried na demoštrovanie použitia jazyka OCL. [6].

Príklad z obrázka 2.6 zobrazuje jednoduchý prípad, v ktorom môžu osoby vlastniť vozidlá. Každá osoba má meno a vek a každé vozidlo je určitej farby. Jazyk UML nám však neumožňuje tento model bližšie špecifikovať. Napríklad nedovoľuje obmedziť model tak, aby vozidlo vlastnila iba osoba mladšia ako 18 rokov. Z podobných dôvodov bol navrhnutý jazyk OCL - aby vyplnil túto medzeru v UML. Príklad výrazu v OCL ošetrujúci minimálny vek vlastníka vozidla je nasledujúci:

```
context: Vozidlo
inv: self.vlastník.vek >= 18
```

Tento OCL výraz je definovaný v kontexte Vozidlo. V kontexte Osoba je nutné tento výraz upraviť na takýto tvar:

```
context: Osoba
inv: vek < 18 implies self.vozový_park->forAll(
                        v~| not v.oclIsKindOf(Auto))
```

Kľúčové slovo *implies* sa používa pre logickú operáciu implikácia. Častokrát je potrebné obmedzujúce pravidlo, ktoré platí pre všetky prvky v kolekcii. Operácia *forAll* umožňuje špecifikovať boolovský výraz, ktorý musí platiť pre všetky prvky v kolekcii. Operácia *oclIsKindOf* je vysvetlená neskôr v tejto kapitole.

Pre výrazy OCL musí vždy platiť, že:

- výraz musí mať typ (buď užívateľský definovaný, alebo preddefinovaný),
- má návratovú hodnotu, ktorá definuje typ výrazu.

Pre OCL obmedzujúce pravidla platí následovné:

- obmedzujúce pravidlo je platný OCL výraz typu *Boolean*,
- ak je obmedzujúce pravidlo splnené, tak vráti *true*, inak *false*.

V tabuľke 2.1 je uvedený zoznam základných preddefinovaných OCL operácií vrátane návratových hodnôt. Operácia *allInstances()* vráti množinu inštancií objektov zvoleného typu a všetkých jeho podtypov. Operácia *oclIsKindOf* vráti true, ak je typ objektu identický s typom objektu predanom v argumente alebo s podtypmi argumentu. Operácia *oclIsTypeOf* sa vyhodnotí ako pravdivá iba v prípade, že je typ objektu identický ako typ objektu v argumente. Operácia *oclAsType* vráti rovnaký objekt, ale typu *OclType*.

Operácia	Návratová hodnota
object.oclIsUndefined()	OclType
object.oclIsKindOf(type : OclType)	Boolean
object.oclIsTypeOf(type : OclType)	Boolean
object.oclIsNew()	Boolean
object.oclAsType(type : OclType)	Type
object.oclInState(str : StateName)	Boolean
type::allInstances()	Set(type)

Tabuľka 2.1: Preddefinované operácie OCL [4].

Ďalšie podrobné informácie k jazyku OCL sa nachádzajú v [1]. Cieľom tejto kapitoly bolo oboznámiť čitateľa s nevyhnutným základom jazyka OCL, ktorý sa používa v kapitolách venujúcich sa implementácii.

Kapitola 3

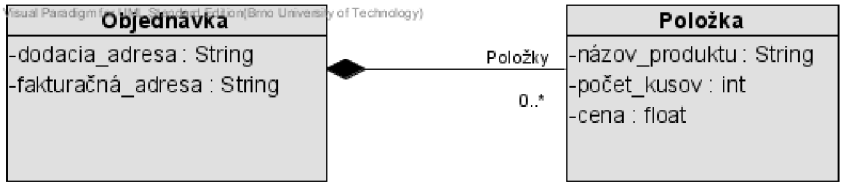
Eclipse Modeling Framework (EMF)

Vývoj tradične začína návrhom modelu. Eclipse Modeling Framework je navrhnutý na zjednodušenie návrhu a implementáciu štruktúrovaného modelu. JAVA platforma poskytuje prostriedky na generovanie kódu, vďaka ktorým sa môžeme sústrediť na model a nie na implementačné detaily. Medzi hlavné pojmy EMF patria: meta-model, generovanie kódu a serializácia. Object Management Group, ďalej len OMG, je asociácia, ktorá bola pôvodne zameraná na stanovenie štandardov pre distribuované objektovo-orientované systémy. V súčasnosti je zameraná na štandardy postavené na modeloch. Eclipse Modeling Framework pôvodne začal ako implementácia štandardu Meta-Object Facility (MOF) asociácie OMG a postupne sa vyvíjal. V súčasnosti vylepšuje model a štruktúru štandardu MOF verzie 2.0 tak, aby bola lepšie pochopiteľná a použiteľná pre koncového užívateľa. Eclipse Modeling Framework je súčasťou modelom riadenej architektúry (MDA). Myšlienkou MDA je schopnosť vyvíjať a riadiť celý životný cyklus produktu sústredením sa na model. Model sám o sebe je opísaný meta-modelom. Potom za použitia mapovania je tento model využívaný na generovanie softvérových prvkov, ktoré budú implementovať celý systém. Sú definované dva typy mapovania: Metadata Interchange, kde sú generované dokumenty ako XML, DTD, a XSD; a Metadata Interfaces, ktoré sa zameriavajú na Javu alebo iný jazyk a generujú IDL[1] kód. MDA je momentálne v procese štandardizácie [12].

Eclipse Modeling Framework poskytuje sadu nástrojov na vytváranie aplikácií alebo generovanie java kódu na základe štruktúrovaného modelu. EMF používa XMI (XML Metadata Interchange) ako kanonickú formu popisu modelu. Existuje niekoľko spôsobov ako vytvoriť takýto model. Použitím dostupných softvérových modelovacích nástrojov, napríklad Rational Rose, z ktorých sa vyexportuje vytvorený model. Alebo je môžné model vygenerovať na základe anotovanej javy, prípadne použiť XML schému. Poslednou z možností je vytvoriť model ručne. Vytvorený model sa použije ako základ pre generátor kódu.

Obrázok 3.1 zobrazuje jednoduchý diagram tried v jazyku UML. Trieda *Objednávka* obsahuje dva atribúty: *dodacia_adresa* a *fakturačná_adresa*. Trieda *Položka* obsahuje tri atribúty: *názov_produktu*, *počet_kusov* a *cena*. Medzi týmito dvoma triedami existuje asociácia (kompozícia). Tento príklad sa použije v nasledujúcich kapitolách na vysvetlenie Ecore modelu a XMI serializácie.

[1] Java IDL je technológia kompatibilná s technológiou CORBA pre distribuované objekty, ktorá umožňuje jednotlivým objektom komunikovať vzájomne medzi sebou bez ohľadu na to, či sú napísané v programovacom jazyku Java, alebo inom jazyku.



Obr. 3.1: UML diagram [15].

3.1 Model Ecore

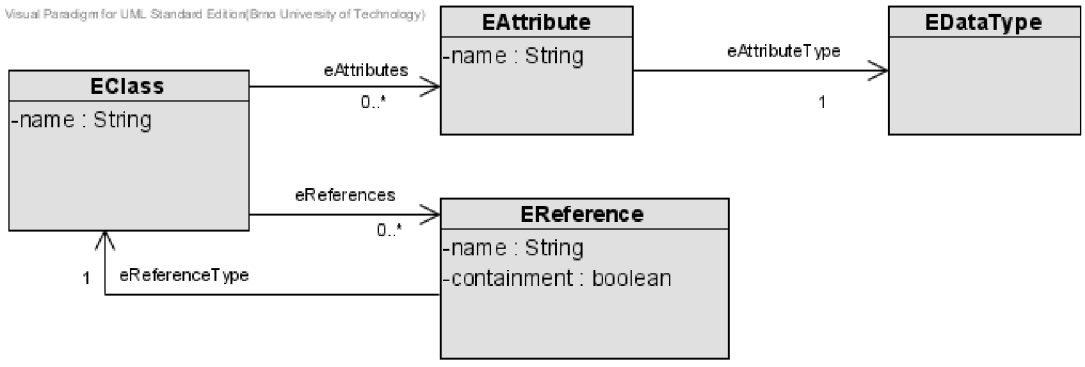
Na popis EMF modelov je potrebný model; metamodel. Model, ktorý reprezentuje modely v EMF, sa nazýva **Ecore**. Poznamenajme, že model nejakého modelu sa nazýva metamodel. Ecore je sám sebe EMF modelom, je teda vlastným meta-metamodelom [15]. Na obrázku 3.2 je zobrazená zjednodušená podmnožina metamodelu Ecore. Diagram zobrazuje iba časť modelu Ecore potrebnú na vysvetlenie príkladu 3.1 a nezobrazuje všetky základné triedy. Diagram tried na obrázku 3.2 obsahuje 4 základné Ecore triedy potrebné na reprezentáciu modela [2]:

1. **EClass** sa používa na reprezentáciu triedy, ktorú modelujeme. Obsahuje meno, žiadny alebo niekoľko atribútov a žiadny alebo niekoľko odkazov.

2. **EAttribute** sa používa na reprezentáciu atribútu, ktorý modelujeme. Atribúty majú meno a typ.

3. **EReference** sa používa na reprezentovanie jedného konca asociácie medzi dvomi triedami. Má meno a cieľ odkazu, ktorým je iná trieda. Ešte obsahuje príznak, ktorý udáva, či sa jedná o kompozíciu.

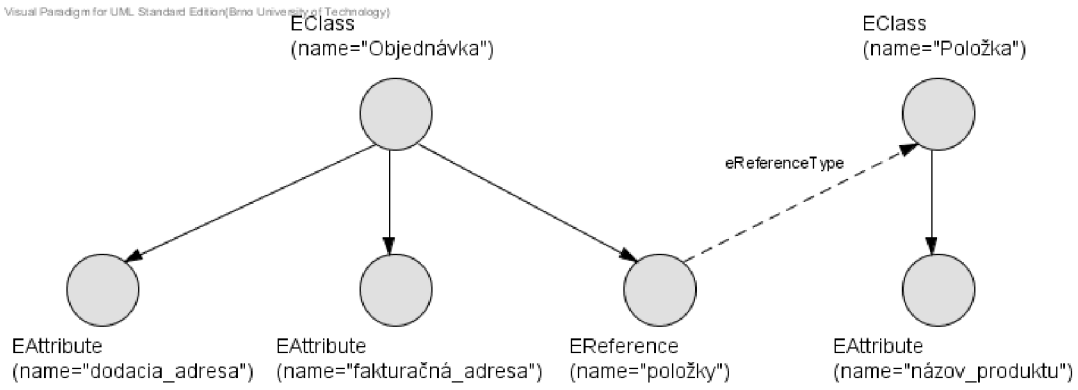
4. **EDataType** sa používa na reprezentovanie dátového typu atribútu, ktorý modelujeme. Datový typ môže byť primitívnym typom ako *int* a *float*, alebo objektovým (zloženým typom).



Obr. 3.2: Diagram tried zobrazuje zjednodušenú podmnožinu metamodelu Ecore [15].

[2]Modelom v tomto prípade rozumieme model, ktorý je zachytený na obrázku 3.1.

Na opis štruktúry tried modelu z príkladu 3.1 sa použijú inštancie tried definované v Ecore (obrázok 3.2). Napríklad sa trieda Objednávka opíše ako inštancia EClass pomenovaná *Objednávka*, ktorá obsahuje dva atribúty (inštancie typu EAttribute) pomenované *dodacia_adresa* a *fakturačná_adresa*, a jeden odkaz (inštancia typu EReference) pomenovaný *položky*, pre ktorého eReferenceType (typ cieľu odkazu) je zhodný s inou inštanciou EClass pomenovanou *Položka* [15]. Tieto inštancie sú zobrazené na obrázku 3.3. Vytvorením inštancií tried, ktoré sú definované v Ecore metamodele na určenie modela vyvíjanej aplikácie, sa vytvára Ecore model. Pojem Ecore model sa často používa v literatúre venovaj EMF.



Obr. 3.3: Ecore inštancie z príkladu 3.1 [15].

3.2 Serializácia modelu

Vytvorený Ecore model je potrebné nejakým spôsobom uchovať, uložiť. EMF používa štandard XML Metadata Interchange (XMI) pre serializáciu Ecore modelov. Príklad z obrázku 3.1 vyzerá serializovaný ako Ecore XMI súbor následovne:

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
    name="po"
    nsURI="http://www.example.com/SimplePO" nsPrefix="po">
  <eClassifiers xsi:type="ecore:EClass" name="Objednávka">
    <eStructuralFeatures xsi:type="ecore:EAttribute"
        name="dodacia_adresa"
        eType="ecore:EDataType
            http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
        name="fakturačná_adresa"
        eType="ecore:EDataType
            http://www.eclipse.org/emf/2002/Ecore#//EString"/>
```

```
    <eStructuralFeatures xsi:type="ecore:EReference"
        name="položky"
        upperBound="-1" eType="#//Item" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Item">
    <eStructuralFeatures xsi:type="ecore:EAttribute"
        name="názov_produktu"
        eType="ecore:EDataType
            http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
        name="počet_kusov"
        eType="ecore:EDataType
            http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
        name="cena"
        eType="ecore:EDataType
            http://www.eclipse.org/emf/2002/Ecore#//EFloat"/>
  </eClassifiers>
</ecore:EPackage>
```

Všimnime si, že XML prvky odpovedajú presne Ecore inštanciám v obrázku 3.3. Atribúty *dodacia_adresa*, *fakturačná_adresa*, *názov_produktu*, *počet_kusov* a *cena* sú deklarované ako zanorené prvky.

3.3 Validačný framework

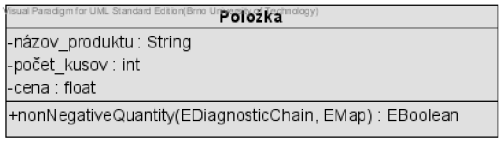
V predchádzajúcich kapitolách bol predstavený model Ecore, hlavný stavebný prvok frameworku EMF, ktorý umožňuje modelovať ľubovoľný systém. Doteraz bola pozornosť sústredená na vytvorenie modelu Ecore zachytávajúceho vzťahy modelovaného systému. Po vytvorení modelu je potrebné zabezpečiť, aby bol vytvorený model udržovaný po celú dobu v konzistentnom stave. Do určitej miery zabezpečujú konzistenciu väzby v modeli Ecore. Napríklad prostredníctvom asociácie je možné zaručiť, aby bolo možné prepojiť iba triedy určitého typu. Na obrázku 3.1 je znázornený jednoduchý príklad. Uvažujme nad situáciou, v ktorej je potrebné zabezpečiť, aby atribút počet_kusov nemohol obsahovať záporné hodnoty. V takomto prípade si už s modelom Ecore nevystačíme. Riešením je validačný framework, ktorý umožňuje definovať obmedzujúce podmienky a invarianty a zaručiť konzistenciu modelu.

Pojem obmedzujúce podmienky a invarianty vo frameworku EMF je inšpirovaný definíciou v UML. Obmedzujúca podmienka je tvrdenie, ktoré musí platiť v určitom okamihu. Oproti tomu invariant je tvrdenie, ktoré musí byť vždy pravdivé. Napríklad predchádzajúca podmienka metódy je rozhodne obmedzujúcim pravidlom, pretože je to obmedzenie, ktoré musí platiť v momente pred vykonaním metódy. Na druhej strane invariant môže zabezpečovať, aby napríklad atribút reprezentujúci počet kusov objednávky nemal nikdy zápornú hodnotu, nezávisle od momentu kedy je vyhodnotený. Vo frameworku EMF môže trieda alebo dátový typ obsahovať anotáciu [3] reprezentujúcu jedno alebo niekoľko obme-

[3] Anotácie predstavujú mechanizmus, prostredníctvom ktorého je možné pripojiť k modelu dodatočné informácie, ktoré nie sú natoľko podstatné, aby boli explicitne podporované metamodelom Ecore. Viac informácií o anotáciach sa nachádza v [15].

dzujúcich podmienok. Napriek tomu, že je mechanizmus anotácií vhodný pre definovanie obmedzujúcich podmienok, pre definovanie invariantov nie je dostatočný. Invariant je veľmi silné tvrdenie o objekte, pre ktorý je definovaný. Preto by mal byť invariant dostupný pre akýkoľvek kód, ktorý pracuje s objektom. S ohľadom na túto skutočnosť reprezentuje EMF invarianty ako operácie tried. Operácia invariantu musí spĺňať určité kritéria. Operácia musí mať dva parametre typu EDiagnosticChain a EMap a musí mať návratovú hodnotu typu EBoolean. Vo vygenerovanej metóde odpovedajú tieto tri typy typom DiagnosticChain, Map a boolean [15].

Na obrázku 3.4 je zobrazená trieda položky z príkladu 3.1 doplnená o operáciu invariantu. Operácia nonNegativeQuantity predstavuje invariant, ktorý zabezpečuje, aby atribút počet kusov neobsahoval záporné hodnoty.



Obr. 3.4: Trieda položky z príkladu 3.1 doplnená o operáciu invariantu.

Z modifikovaného modelu Ecore doplneného o operáciu invariantu sa vygeneruje zdrojový kód, ktorý obsahuje metódu implementujúcu invariant *nonNegativeQuantity* následovne:

```
public boolean nonNegativeQuantity(
                    DiagnosticChain diagnostics, Map context)
{
  // -> specify the condition that violates the invariant
  // -> verify the details of the diagnostic
  if (false) {
    if (diagnostics != null)
    {
      diagnostics.add(
        new BasicDiagnostic(
              Diagnostic.ERROR,
              InstructionSetValidator.DIAGNOSTIC_SOURCE,
              InstructionSetValidator.CR_OPERATION_HAS_NAME,
              EcorePlugin.INSTANCE.getString(
                "_UI_GenericInvariant_diagnostic",
                new Object[] {
                  "nonNegativeQuantity",
                   EObjectValidator.getObjectLabel(
                      this, (Map<Object, Object>) context) }),
              new Object [] { this }));
    }
    return false;
  }
  return true;
}
```

Vygenerovanú metódu je potrebné upraviť. Prvý príkaz *if* kontroluje podmienku invariantu, ktorá sa vyhodnotí a v prípade, že nie je splnená, sa vytvorí diagnostická správa. Vygenerovaná metóda ale obsahuje v prvom príkaze *if* hodnotu *false*, ktorú je potrebné nahradiť podmienkou porušujúcou invariant. EMF neposkytuje možnosť definovať podmienku invariantu v modeli Ecore. Preto je nutné definovať podmienku pre invariant nonNegativeQuantity v Java kóde [15]. Nasledujúci príklad ukazuje výraz, ktorým sa nahradí *false* z prvého if príkazu.

```
if(pocet_kusov < 0)
```

Niekedy je potrebné upraviť detaily o výsledku validácie. Tento výsledok sa nazýva *diagnostic* reprezentovaný triedou implementujúcou rozhranie *Diagnostic*. Ako napríklad inštancia triedy *BasicDiagnostic* vo vygenerovanom kóde. Argumenty používané pri vytváraní objektu sú vysvetlené v tabuľke 3.1. Vo väčšine prípadov je žiadané upraviť iba správu (message), ktorá zahŕňa meno invariantu a opis objektu, a vytvoriť špecifickejšiu a detailnejšiu správu. Volanie metódy *diagnostic.add(new BasicDiagnostic(...));* pridá vytvorenú diagnostiku do objektu *diagnostics*. Diagnostics je objekt implementujúci rozhranie *DiagnosticChain*. Umožňuje zoskupiť viaceré diagnostiky vytvorené počas validácie na rôznych obmedzujúcich podmienkach a invariantoch [15].

Argument	**Typ**	**Popis**
severity	int	Sumarizuje výsledky validácie. Argument môže byť jedným z následujúcich výčtov: Diagnostic.OK, Diagnostic.INFO, Diagnostic.WARNING, Diagnostic.ERROR, alebo Diagnostic.CANCEL. Ok značí, že validácia prebehla úspešne; INFO, WARNING, a ERROR sú použiteľné ak bola podmienka alebo tvrdenie porušené; a CANCEL sa používa na signalizáciu stavu, že bola validácia prerušená.
source	String	Identifikuje miesto, kde sú obmedzujúca podmienka alebo invariant definované. Zvyčajne sa používa konštanta DIAGNOSTIC_SOURCE triedy validátora, ktorú vo väčšine prípadov nie je potrebné zmeniť.
code	int	Identifikuje konkrétny dôvod neúspechu validácie.
message	String	Opisuje výsledok validácie. Predvolená implementácia generovaná prostredníctvom EMF používa externý reťazec na vytvorenie správy s meno obmedzujúcej podmienky alebo s meno invariantu a opisom objektu, ktorý sa overuje.
data	Object [ ]	Odkazy objektov, ktoré sú relevantné pre validáciu. Ako napríklad inštancia, voči ktorej sa vykonáva validácia.

Tabuľka 3.1: Opis argumentov Diagnostic koštruktora [15].

Okrem vygenerovaných metód, ktoré implementujú invarianty a obmedzujúce podmienky, vytvorí validačný framework ešte novú triedu v balíku *util*: validačnú triedu. Táto trieda poskytuje prostriedky na vykonanie validácie. Kontrolu platnosti objektov je možné vykonávať napríklad v dôležitých momentoch behu aplikácie, alebo na podnet užívateľa. EMF umožňuje rôzne spôsoby implementácie metód vykonávajúcich validáciu obmedzujú-

cich podmienok a invariantov. Podrobné informácie o tejto problematike je možné získať v publikácií [15]. Validačný framework EMF poskytuje pomocnú triedu nazývanú *Diagnostician*, ktorá je odporúčeným vstupným bodom validácie. S využitím tejto triedy je možné veľmi jednoducho implementovať metódu validateObject(), zobrazenú na príklade uvedenom nižšie, ktorá kontroluje platnosť objektov.

```
public static boolean validateObject(EObject eObj)
{
    Diagnostic diagnostic = Diagnostician.INSTANCE.validate(eObj);
    return diagnostic.getSeverity() == Diagnostic.OK;
}
```

EMF umožňuje prezentovať užívateľovi výsledky validácie prostredníctvom chybového dialógového okna. Lepším riešením je ale využiť samotnú platformu Eclipse a integrovať výsledky do pohľadu "Problems".

Kapitola 4

Graphical Modeling Framework (GMF)

Graphical Modeling Framework je generatívny prístup ako vytvárať grafické editory postavené nad EMF. GEF [1] je framework vyvíjaný v rámci Eclipse, ktorý umožňuje vytvárať obecné grafické editory. Takéto editory sú integrované priamo do platformy Eclipse a poskytujú nasledujúce základné vlastnosti:

- paletu nástrojov,
- pohľady,
- undo/redo,
- pravítka,
- sprievodcov a tutoriály,
- zarovnávanie na mriežku,
- atď.

Graphical Editing Framework využíva na vykresľovanie a usporiadanie grafických prvkov odľahčený toolkit Draw2D, ktorý je postavený nad SWT [2]. GMF je možné si predstaviť ako strechu nad EMF a GEF. Využitím GMF sa teda dokáže vytvoriť grafický editor, ktorý pomocou diagramov vykreslovaných prostredníctvom frameworku GEF zobrazuje modely EMF. Úsilie, ktoré bolo potrebné vynaložiť na vytvorenie vlastného editora diagramov bez využitia GMF, bolo značne vysoké a návrh vlastného vizuálneho zápisu symbolov sa tým pádom stával neekonomickým. V tomto smere je GMF revolúciou, pretože jeho koncept značne urýchľuje návrh grafického editoru a tým pádom aj jeho výslednú cenu. Koncept vývoja editora pomocou frameworku GMF je znázornený na obrázku 4.1. Z obrázku je zrejmé, že pred vytvorením projektu je potrebný Ecore model. Jedná sa o model EMF zachytávajúci vzťahy medzi prvkami, ktoré bude vygenerovaný editor vytvárať, prípadne modifikovať. Na vytvorenie kompletného editora je potrebných ešte niekoľko doplnkových modelov:

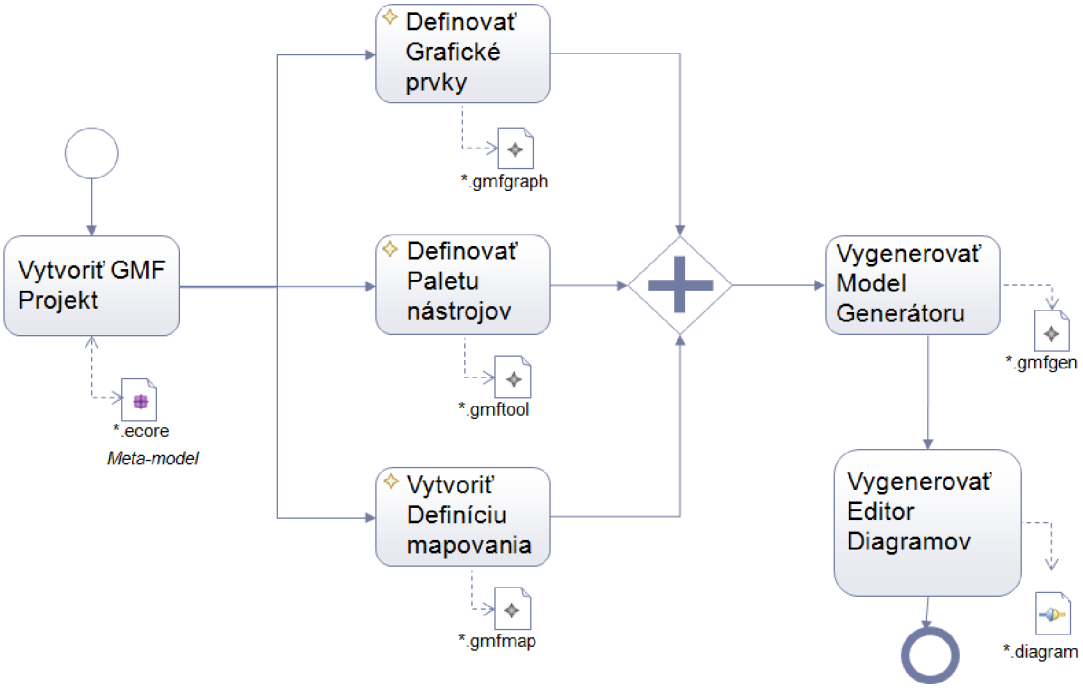
[1] GEF - Graphical Editing Framework
[2] SWT - Standard Widget Toolkit

1. **.gmfgraph** je model, ktorý udáva ako budú vyzerať jednotlivé tvary, ich uzly a prepojenia na plátne. Umožňuje dekoráciu tvarov, definovanie typov písma atď.

2. **.gmftool** je model, ktorý špecifikuje paletu nástrojov v editore, prípadne iné nástroje.

3. **.gmfmap** je model, ktorý určuje ako sa modely *gmfgraph* a *gmftool* medzi sebou namapujú. Je možné si to v jednoduchosti predstaviť tak, že sa v tomto modeli určí, ktorý nástroj (ikona) v palete nástrojov odpovedá entite na plátne. Ktorá entita sa zobrazí ako trieda, prípadne asociácia atp.

Zo všetkých týchto doplnkových modelov vytvorí GMF model **.gmfgen** automaticky. Jedná sa o nízko úrovňový model, ktorý slúži ako dekorátor, tzv. doplní potrebné údaje pre generovanie kódu, ako napríklad názvy JAVA balíčkov atp. Vstupom generátoru kódu je tento model, výstupom je zásuvný modul *.diagram*, ktorý obsahuje požadovaný grafický editor.



Obr. 4.1: Koncept vývoja editora pomocou GMF [8].

4.1 Definícia grafických prvkov

Model definície grafických prvkov reprezentovaný ako .gmfgraph sa používa na niekoľko vecí [16]:

1. Na reprezentovanie prvkov doménového modelu je možné definovať množinu tvarov. Štandardný editor obsahuje definíciu vlastností rozmerov a farieb, ako napríklad hrúbku čiar, farby pozadia a popredia, prípadne rôzne statické výplne. Typ tvaru

sa vytvorí pridaním dostupných možností ako potomkov (New Child) súčasného tvaru, ktorý predstavuje našu triedu.

2. Na definovanie uzlov grafu a spojení. Prvky doménového modelu, ktoré majú byť umiestnené na plátno editora sú definované ako uzol (Node). Doménové prvky, ktoré sú určené na prepojenie iných doménových prvkov, sa definujú ako spoj (Connection).

3. Na definovanie zložených oddielov (Compartments). Zložené oddiely sú úseky uzlov, ktoré obsahujú iné uzly, alebo skupinu prvkov.

4. Nakoniec na definovanie štítkov a zobrazenie textu priradeného k jednotlivým grafickým prvkom.

Framework GMF poskytuje niekoľko základných grafických primitív (ako napríklad úsečka, obdĺžnik, kruh a pod.), ktoré je možné medzi sebou kombinovať a definovať tak tvar uzlu. Grafická definícia tvaru v modeli gmfgraph je uložená v strome. Takýto strom opisujúci tvar uzlu sa v modeli gmfgraph nazýva **Figure Descriptor**. Figure Descriptor je uzavretá jednotka. Získať verejný prístup k jej položkám je možné iba pomocou definovania tzv. *"Child Access"* prvkov. V kontexte gmfgraph sa jedná o metódu, ktorej návratová hodnota je zvolená položka. Namapovať tvar na doménový prvok je možné len v tom prípade, že je Figure Descriptor potomkom prvku, ktorý určuje jeho typ. Prvky určujúce typ sú uzly, spoje, štítky a zložené oddiely. Tieto prvky sa používajú v modeli mapovania. Uzly, spoje a zložené oddiely sa väčšinou mapujú na triedy a štítky na atribúty tried.

Mnohokrát je potrebné zobraziť jeden doménový prvok v rámci tvaru iného doménového prvku. Agregačný doménový prvok sa namapuje na uzol, ktorý sprístupňuje svoju časť tvaru pomocou *"Child Access"* prvka. Vytvorí sa prvok zloženého oddielu, ktorý ukazuje na miesto v strome tvarov volaním "Child Access" metódy. Konstitučný doménový prvok sa namapuje na zložený oddiel (compartment).

4.2 Model nástrojov

Model nástrojov reprezentovaný ako .gmftool špecifikuje paletu nástrojov v editore. Paleta nástrojov je množina tlačítok na pravej strane editora, ktorá umožňuje pridávať prvky do inštancie doménového modela [16].

Každému nástroju v palete sa priradí ikona, ktorá má rozmery 16 x 16 pixelov. GMF umožňuje vytvárať v rámci palety skupiny nástrojov. Potom je možné začleniť každý nástroj do príslušnej skupiny [16]. Napríklad skupina pre združovanie nástrojov na vytváranie prvkov doménového modela alebo skupina nástrojov na vytváranie spojov medzi rôznymi prvkami.

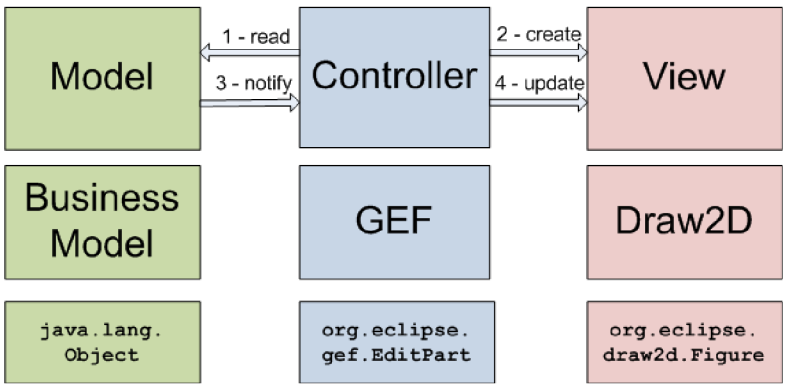
4.3 Model mapovania

Model mapovania je najklomplexnejším modelom. Prostredníctvom tohto modelu sa vykonáva mapovanie definície nástrojov a definície grafických prvkov na doménový metamodel [16]. Pre každý prvok doménového modelu, ktorý bude namapovaný priamo na povrch diagramu, sa definuje najprv `Top Node Reference`. V ďalšom kroku sa vytvorí mapovanie uzlov (`Node Mapping`). Mapovanie uzlov umožňuje zobraziť prvok doménového modelu na tvar, ktorý sa definoval v modeli `.gmfgraph` (Definícia grafických prvkov). Mapovanie štítkov (`Label Mapping`) dovoľuje zobraziť vlastnosť prvku doménového modelu

na štítok definovaný opäť v modele `.gmfgraph`. Pomocou mapovania zložených oddielov (`Compartment mapping`) je možné namapovať niekoľko inštancií prvkov doménového modelu do jedného tvaru definovaného v modeli .gmfgraph. Na obrázku 2.3 je zobrazený komponent dekóder. Šablóna takéhoto dekodéra neobsahuje presne definovaný počet portov. To znamená, že v doménovom modeli (Ecore) by sa tento dekóder popísal dvoma triedami a vzťahom kompozície. Jedna trieda by predstavovala port a druhá samotný dekóder. Práve mapovanie zložených oddielov umožňuje zobraziť dekóder presne tak, ako ho zobrazuje obrázok 2.3, kde sú porty súčasťou jedného tvaru. Nakoniec mapovanie spojov (`Link Mapping`) zobrazuje tvar spoja, ktorý je definovaný v modeli gmfgraph, medzi dvomi triedami z doménového modela.

4.4 Architektúra MVC

V aplikáciách, ktoré nerozlišujú medzi sémantickou a prezentačnou vrstvou, môže nastať niekoľko problémov. Takéto aplikácie sú náročné na údržbu, pretože vzájomná závislosť medzi všetkými komponentmi spôsobuje silne domínový efekt pri zmene v ktoromkoľvek mieste kódu. Vysoká väzba medzi triedami zapríčiňuje, že je veľmi zložité, skoro až nemožné, triedy znovu použiť, pretože závisia na mnohých ďalších triedach. Pridávanie nových datový pohľadov vyžaduje preimplementovať logiku aplikácie, čoho následkom je údržba na mnohých miestach programu. Spomínané problémy rieši návrhový vzor Model View Controller (MVC) odstránením väzby medzi prezentáciou dát, interakciou s užívateľom a logikou aplikácie. Základný koncept je znázornený na obrázku 4.2 v kontexte frameworku GEF. Kontrolér riadi prístup k sémantickým dátam, vytvára, obnovuje a ruší pohľady na dáta na základe interakcie s užívateľom [5].

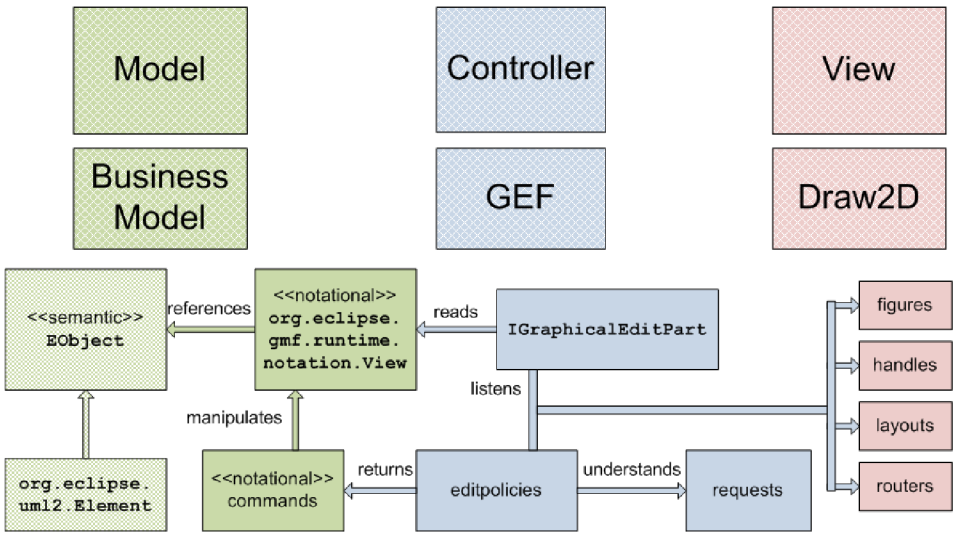


Obr. 4.2: GEF Model View Controller [5].

Sémantické dáta môžu byť zobrazené v rôznych kontextoch, editoroch a ako také by nemali ukladať informácie o tom, ako môžu byť zobrazené v editore. Taktiež sa sémantické dáta potenciálne zobrazujú niekoľko krát v rovnakom diagrame, čo znamená, že informácia o pohľade musí byť ukladaná takisto niekoľkokrát. S cieľom uľahčiť toto je potrebné, aby informácia o pohľade bola uložená v inom modeli, ktorý odkazuje na sémantický model. GMF to rieši tým, že poskytuje model presistencie pre GEF prostredníctvom diagramovej vrstvy. Definuje "notation-meta-model" v EMF, ktorý je univerzálny a logicky oddelený od sémantického modelu persistencie. To umožňuje viacerým klientom GMF mať kooperujúce diagramy prostredníctvom kompatibilného a konzistentného formátu. Oddelenie pohľadu

od sémantiky umožňuje klientom definovať niekoľko tvarov pre jeden sémantický prvok. Okrem toho ďovoľuje zobraziť sémantický prvok rozdielne v závislosti od kontextu [5].

Na obrázku 4.3 je zobrazený Model View Controller frameworku GMF. Sémantická vrstva je oproti MVC framworku GEF rozšírená o podporu modelu "notation-meta-model". Kontrolér implementuje rozhranie IGraphicalEditPart a s použitím politík na základe požiadavkov modifikuje sémantický model prostredníctvom príkazov.

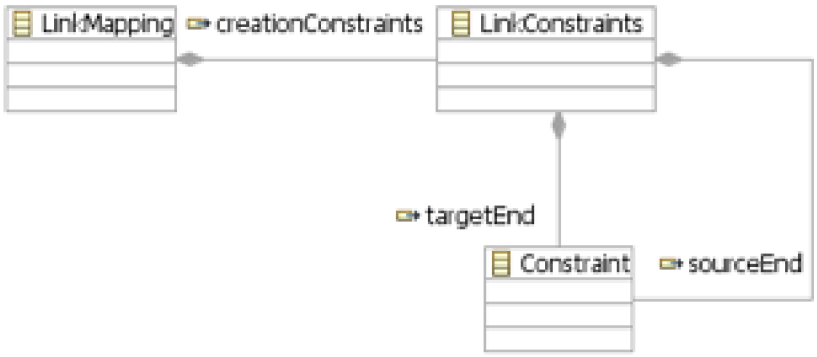


Obr. 4.3: GMF Model View Controller [5].

4.5 Obmedzenia spojov

Obmedzenia spojov (Link Constraints) sa používajú na kontrolu procesu vytvárania spojov. Štandardné "Live" obmedzenia kontrolujú obmedzujúce podmienky po sérií modifikačných príkazov, ktoré sa vykonajú ako celok. Ak štandardné obmedzenia nevyhovujú zadaným podmienkam, spôsobia tak obnovenie do pôvodného stavu pred transakciou. Voliteľne sa v užívateľskom rozhraní môže zobraziť hláška. Oproti tomu obmedzenia spojov sa nevyhodnocujú v celku, ale vyhodnotenie prebieha v niekoľkých fázach. Rozlišujú sa dva typy uzlov: zdrojový uzol a cieľový uzol. Na základe podmienok, ktoré boli definované pre zdrojový a cieľový uzol, umožňujú obmedzenia spojov povoliť/zakázať zahajovacú alebo koncovú fázu vytvárania spoja. Podmienky obmedzení spojov sú vyhodnoté nad doménovým prvkom ešte pred vytvorením spoja. Oproti tomu štandardné obmedzenia sa vyhodnocujú až v momente, keď bol spoj vytvorený. V prípade, že podmienky obmedzujúceho pravidla neboli pre cieľový uzol splnené, nie je možné spoj vytvoriť a užívateľské rozhranie to signalizuje užívateľovi zmenou kurzora (prešKrtnutá šípka) [2].

Obrázok 4.4 ilustruje štruktúru modelu mapovania, ktorá uľahčuje vytváranie obmedzení spojov. Každé mapovanie spoja (Link Mapping) môže mať obmedzenia spoja (Link Constraints) buď s cieľovým koncom (targetEnd), zdrojovým koncom (sourceEnd) alebo s obidvoma prvkami, ktoré obsahujú obmedzujúce pravidlá pre zúčastnené uzly. Obmedzenia zdrojového konca sú definované v kontexte zdrojového uzlu doménového prvku, ktorý je dostupný prostredníctvom premennej "self". Cieľový uzol doménového prvku je dostupný prostredníctvom premennej "oppositeEnd". Ak ešte nie je zvolený cieľový uzol, na začiatku



Obr. 4.4: Obmedzenia spojov [2].

vytvárania spoja, je premenná "oppositeEnd" nedefinovaná. Analogicky to platí aj pre obmedzenia cieľového konca, ktoré sú definované v kontexte cieľového prvku. V tomto prípade premenná "self" odkazuje na cieľový uzol doménového prvku a premenná "oppositeEnd" odkazuje na zdrojový uzol. [2].

Pre spracovanie obmedzení spojov sa používa nasledujúca logika:

- Spoj začína výberom zdrojového uzla. Vyhodnotia sa podmienky pre "sourceEnd", ktoré značia, že je uzol v stave akceptujúcom vytváraný spoj. V tomto momente je premenná "oppositeEnd" nedefinovaná [2].
- Nasleduje výber cieľového uzla. Opätovne sa vyhodnotí obmedzenie "sourceEnd", aby bolo zdrojovému uzlu umožnené prijať/odmietnuť svoj cieľ dostupný cez premennú "oppositeEnd", ktorá už v tomto prípade je definovaná. Ak je obmedzenie "sourceEnd" splnené, nasleduje vyhodnotenie obmedzenia "targetEnd". Spoj je realizovaný iba v prípade, ak uspejú obidva obmedzenia [2].

Ukážka použitia obmedzenia spojov je v kapitole 6.2.3.

Kapitola 5

Návrh riešenia

Pre implementáciu nástroja pre návrh čipu v UML bola zvolená platforma Eclipse, ktorá poskytuje široké možnosti modelovania. Na trhu existuje niekoľko výrobcov SW, ktorý sa zaoberajú problematikou návrhu vstavaných systémov v jazyku UML - napríklad produkt Magic Draw spoločnosti No Magic, Inc. Podobné produkty sú však nevyhovujúce, pretože nástroj, ktorý má byť implementovaný v rámci tohto projektu, musí umožňovať generovanie kódu z jazyka UML do jazkyka ISAC. ISAC je jazyk, ktorý sa používa v rámci projektu Lissom na popis architektúry vstavaného systému a následne vygenerovanie softvérových nástrojov. Framework EMF, ktorý je súčasťou platformy Eclipse, poskytuje modelovacie nástroje, ktoré je možné aplikovať na modelovanie vstavaných systémov. Z vytvoreného modelu je potom možné vygenerovať príslušný zdrojový kód. Nástroj pre návrh čipu bude implementovaný s využitím frameworku GMF platformy Eclipse, ktorý umožňuje vytvoriť grafický editor EMF modelov. `Uml2tools` je sada editorov založených nad GMF, ktoré sú určené na editovanie a prezeranie UML modelov. Tieto nástroje sú však pre účely tohto projektu nepoužiteľné, pretože návrhár by bol nútený vždy aplikovať na vytvorené modely sadu preddefinovaných UML profilov a takýto systém práce by bol nedostatočne komfortný a neprehľadný. Oproti tomu, ak bude vytvorený vlastný editor, umožní sa tak návrhárovi používať preddefinované šablóny HW prvkov, ktoré si vhodne parametrizuje. Tieto preddefinované šablóny budú založené na jazyku UMLISAC a umiestnené priamo v palete nástrojov. Posledným z dôvodov, prečo bola zvolená platforma Eclipse je ten, že sa už používa v projekte Lissom a užívateľ tak získa kompletný softvérový balík, s ktorým je zvyknutý pracovať.

Dôležitou vlastnosťou navrhnutého nástroja bude jeho schopnosť generovať iba syntakticky a sémanticky správny zdrojový kód. Musia byť teda navrhnuté mechanizmy, ktoré overia sémantickú správnosť modelu. Do úvahy prichádzajú dva základné prístupy.

Prvým je sémantická kontrola na najnižšej úrovni, ktorá využíva natívny syntaktický analyzátor jazyka ISAC. V tomto prípade je nutnosťou transformovať serializovaný model do XML medziformátu jazyka ISAC, ktorý sa následne skontroluje. Výhodou takéhoto riešenia je rezistentnosť voči zmenám v jazyku ISAC, pričom grafický editor bude produkovať stále validné modely. Nevýhodou je spôsob, akým sa budú reflektovať chyby v modeli návrhárovi. Pretože v tomto prípade prebieha sémantická kontrola na najnižšej vrstve a chyby, ktoré budú zistené už v tomto mieste, sa musia vynoriť až do najvyššej vrstvy. Týmto spôsobom by návrhár mohol vytvoriť nesprávny model, pričom by ho následné editor informoval o vzniknutých chybách formou správy, červeným vyznačením nesprávnych prvkov, prípadne vyskakovacím oknom. Návrhár by bol nútený opravovať chyby ručne, až do doby, kým by bol model validný.

Druhým prístupom je sémantická kontrola na najvyššej úrovni s využitím možnosti platformy Eclipse. Sémantickú kontrolu na tejto úrovni je možné do istej miery zabezpečit samotným doménovým metamodelom, predovšetkým však využitím jazyka OCL [1]. Podpora jazyka OCL je súčasťou frameworkov EMF a GMF. Nevýhodou tohto riešenia je slabá odolnosť voči zmenám v jazyku ISAC. Ak sa zmení špecifikácia jazyka, je potrebné vykonané zmeny premietnuť do doménového metamodelu a prispôsobiť pravidlá jazyka OCL. Toto riešenie však nedovolí návrhárovi vytvoriť nesprávny model, následkom čoho nemusí návrhar ručne opravovať vzniknuté chyby.

Zvolil som druhý spomínaný prístup, pretože predpokladám, že tento prístup je užívateľsky prívetivejší a z pohľadu návrhára viac očakávaný. Navyše zastávam názor, že postupom času už nebudú zmeny v špecifikácií jazyka ISAC natoľko invazívne, aby súčasné udržiavanie metamodelu a pravidiel jazyka OCL predstavovalo výrazné ekonomické nevýhody.

[1] OCL - Object Constraint Language je deklaratívny jazyk na popisovanie pravidiel v UML modeloch vyvíjaný spoločnosťou IBM.

Kapitola 6

Implementácia

V tejto kapitole je rozobratá implementácia dvoch základných diagramov, ktoré spoločne tvoria výsledný editor jazyka ISAC. Diagramy sú vytvorené pomocou Eclipse GMF frameworku. Štrukturálny diagram umožňuje návrhárovi vytvoriť model zdrojov. Napríklad umiestnením prvku registru, pamäti, funkčnej jednotky atp. na plátno diagramu. Prvky je možné medzi sebou navzájom prepájať s využitím mechanizmu portov a rozhraní. V návrhu je rozhranie, ktoré poskytuje dáta označené kolieskom a rozhranie, ktoré dáta vyžaduje oblúkom. Eclipse GMF nepodporuje navrhnutú notáciu, preto bola notácia upravená. Každé rozhranie je označené jednotne kolieskom. Medzi poskytovaným a požadovaným rozhraním sa rozlišuje pomocou typu spoja medzi portom a rozhraním. Pre rozhranie, ktoré dáta vyžaduje má tento spoj tvar prerušovanej čiary. Spoj rozhrania, ktoré dáta poskytuje, sa označuje plnou čiarou. Druhým zo spomínaných diagramov je diagram inštrukčnej sady. Tento diagram je grafickým ekvivalentom textovej podoby jazyka ISAC. V palete nástrojov sú pripravené šablóny prvkov, ktoré sa umiestňujú na plátno. Medzi základné stavebné bloky patrí operácia a skupina. Do operácie je možné vkladať inštancie iných operácií, atribúty a terminály, obdobne ako v jazyku ISAC. Každéj operácií je možné priradiť prvok Activation, Behaviour a Expression a definovať tak jej správanie. Namodelované operácie sa zaraďujú do skupín pomocou definovaných spojov z palety.

Každý z diagramov má vlastný metamodel Ecore [1], ktorý definuje vzťahy medzi prvkami v diagrame. Na vizualizáciu metamodelu Ecore sa používa diagram tried. Základným postupom práce je vytvoriť metamodel pre obidva diagramy. Každému doménovému prvku z Ecore vytvoriť grafický popis v definícií grafických prvkov [2], ktorý definuje tvar prvku na plátne. Dôležitým krokom je správne vytvoriť model mapovania [3], ktorý prepojí doménový metamodel s definíciou grafických prvkov a s paletou nástrojov. Ďalším krokom je vygenerovanie základných zásuvných modulov do platformy Eclipse, ktoré je nevyhnutné upraviť do takej podoby, aby výsledný editor spĺňal požadované správanie. Detaily tohto procesu rozoberajú následujúce podkapitoly.

6.1 Štrukturálny diagram

Metamodel Ecore štrukturálneho diagramu je zobrazený v diagrame tried na obrázku 6.1. Každý grafický editor vytvorený pomocou frameworku GMF obsahuje plátno, do ktorého je

[1] Modelu Ecore sa venuje kapitola 3.1.
[2] Definícia grafických prvkov je vysvetlená v kapitole 4.1.
[3] Model mapovania rozoberá kapitola 4.3.

možné vkladať prvky z palety. Plátno je namapované na doménový prvok, v tomto prípade na triedu *Canvas*. S využitím metamodelu Ecore je možné do istej miery kontrolovať sémantiku výsledných modelov. Pomocou agregácie a kardinality sa špecifikuje, ktoré prvky je dovolené umiestniť na plátno. Obrázok 6.1 znázorňuje, že na plátno (zastúpené triedou Canvas) je možné umiestňovať prvky *Register*, *FileRegister*, *Memory*, *FU*, *Decoder* a *Interface*. Kardinalita pri každom z nich udáva, že sa na plátne môžu vyskytovať v ľubovoľnom počte. Prvky obsahujú atribúty, ktoré návrhár pri vývoji systému vhodne parametrizuje. Na prepájanie stavebných blokov slúžia porty a rozhrania. Metamodel štrukturálneho diagramu obsahuje triedu *Port*. Užívateľovi je zabránené vytvoriť port v komponente, ak v metamodeli Ecore nie je medzi triedou komponenty a triedou portu vytvorená agregácia.

Framework GMF definuje dva základné typy prvkov, uzly a spoje. Doteraz reprezentovala každá trieda z príkladu 6.1 jeden uzol. Triedy *ProvidedInterface* a *ReqiuredInterface* sa mapujú na spoje medzi rozhraním a portom. Trieda *ProvidedInterface* prepája port a všeobecné rozhranie (triedu *Interface*) a vytvára tak rozhranie, ktoré poskytuje dáta. Analogicky trieda *RequiredInterface* vytvára rozhranie, ktoré vyžaduje dáta. Framework GMF nedovoľuje vytvoriť spoj, ktorý nemá odpovedajúci doménový prvok. Každý spoj obsahuje zdroj a cieľ určený asociáciou *source* a *target* s kardinalitou 0..1, čo umožňuje kontrolovať vytváranie spojov iba medzi povolenými uzlami. Veľmi dôležitá je agregácia medzi triedou *Port* a triedou *RequiredInterface* resp. *ProvidedInterface*. Túto agregáciu využíva Model mapovania a definuje smer, v ktorom sa vytvára spoj, teda smerom od portu k rozhraniu.

Častokrát sémantická kontrola na úrovni doménoveho modelu nie je dostatočná, pretože nie je možné zachytiť všetky obmedzujúce pravidla jazyka ISAC. Napríklad nie je možné pomocou doménového modela ošetriť situáciu, aby návrhár parametrizoval potrebné atribúty správne. Validačný framework opísaný v kapitole 3.3 umožňuje definovať invarianty pre doménové prvky. Invariant je v kontexte EMF definovaný ako operácia v danej triede [4]. Jednotlivé operácie *hasName* a *hasSize* z príkladu 6.1 implementujú kontrolu atribútu name a size.

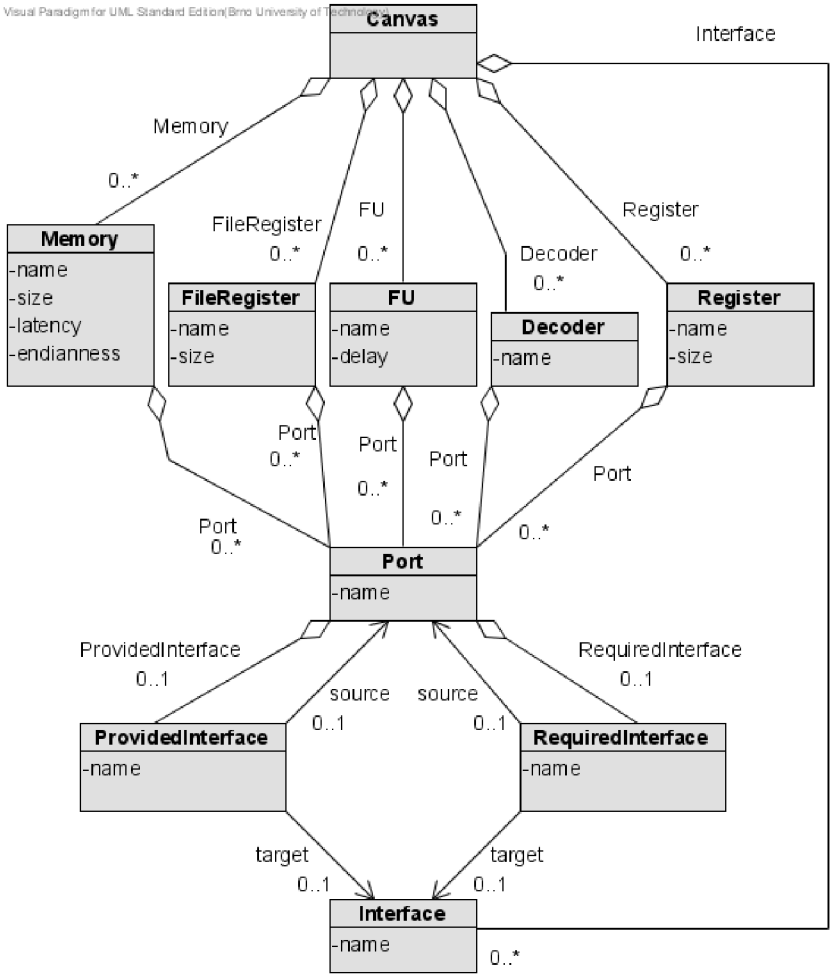
6.1.1 Vytváranie portov

Kapitola 2.2 ukazuje použitie portov, ktoré patria medzi štrukturálne vlastnosti jazyka UML. Následne v kapitole 2.4 je vysvetlené v akej súvislosti sa porty použijú v spojení s modelovacím jazykom UMLISAC. Táto časť práce sa zaoberá s implementáciou spomínaných portov v prostredí Graphical Modeling Framework.

V metamodeli EMF je port zastúpený vlastnou triedou. Každý register môže obsahovať niekoľko užívateľom definovaných portov. Preto je medzi triedou registra a triedou portu agregácia s kardinalitou 0 až *. Agregácia v metamodele EMF je pomenovaná “hasPorts”. Toto pomenovanie využijeme v modeli mapovania. Port v jazyku UMLISAC predstavuje malý štvorec, ktorý sa nachádza na okraji zvolenej komponenty. Definícia grafických prvkov 4.1 udáva ako presne bude takýto port vyzerať. Je potrebné definovať uzol portu (Node) a popisovač tvarov (Figure Descritor). Obrázok 6.3 zobrazuje úsek obsahu súboru s popisom definície grafických prvkov portu.

Popisovač tvaru portu má potomka “Rectangle”, ktorý zabezpečí, aby port vyzeral ako štvorec. “Rectangle” má potomka “Preferred Size”, ktorý nastavuje východziu veľkosť tvaru. Dôležitým prvkom je uzol nazvaný “PortNode”. Takémuto uzlu je nutné prideliť definovaný tvar “PortFigure” a nastaviť vlastnosť “Affixed Parent Side” na hodnotu “NSEW”,

[4] Podrobné informácie ako implementovať invariant v kontexte EMF sú rozobraté v kapitole 3.3.

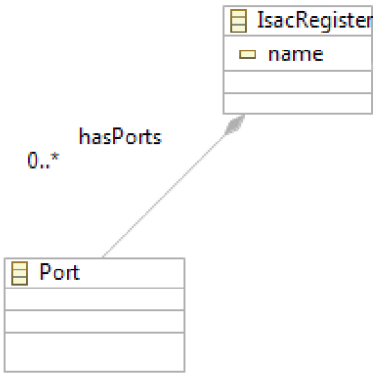


Obr. 6.1: Ecore metamodel štrukturálneho diagramu.

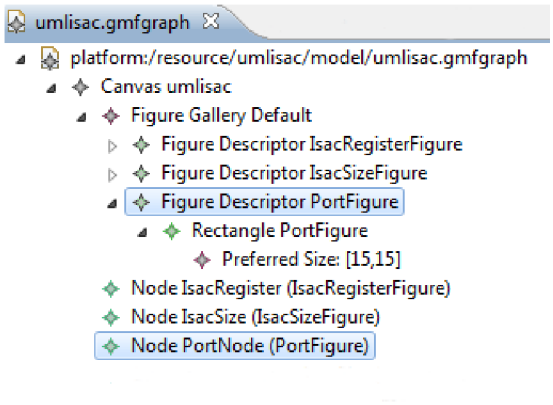
ktorá zabezpečí, že zobrazovaný tvar portu sa bude zobrazovať na okraji nadradenej komponenty.

Najzložitejšou úlohou je definovať model mapovania, ktorý prepája metamodel s definíciou tvaru a s nástrojom palety pre vytvorenie prvku. Obrázok 6.4 znázorňuje tento model mapovania.

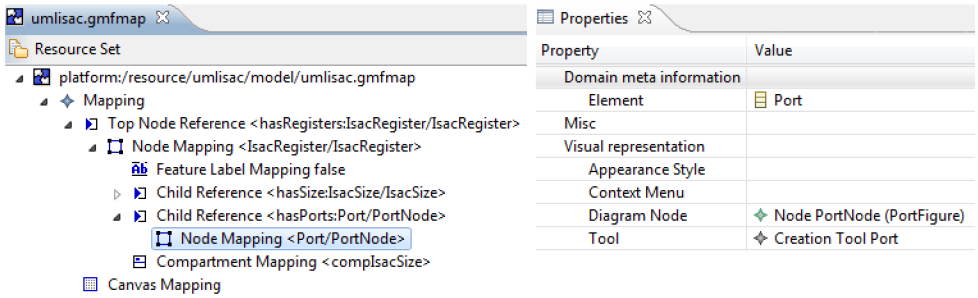
Aby mohol register obsahovať jednotlivé porty, je potrebné nadefinovať odkaz na potomka (Child Reference). Takýto odkaz musí obsahovať ku akému potomku sa viaže, v tomto prípade je to uzol portu a zároveň, ku ktorému vzťahu sa viaže - agregácia "hasPorts". Dôležitým prvkom je uzol mapovania (Node Mapping), ktorý určuje vzťah medzi metamodelom EMF, definíciou tvaru a nástrojom palety, s ktorým port vytvoríme. Pravá časť obrázku 6.4 zobrazuje vlastnosti uzlu mapovania. Mapuje sa trieda Port z metamodelu EMF na uzol "PortNode", ktorý je určený v súbore s definíciou grafických prvkov - *.gmfgraph.



Obr. 6.2: Diagram tried metamodelu EMF.



Obr. 6.3: Definícia grafických prvkov.



Obr. 6.4: Model mapovania.

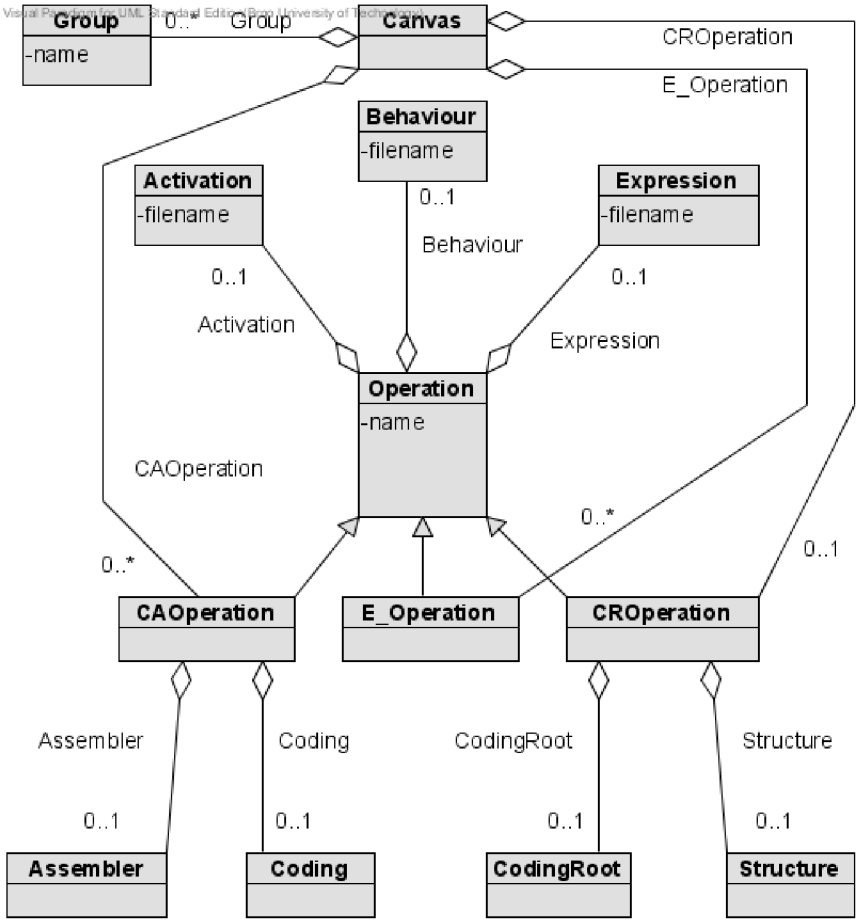
6.2 Diagram inštrukčnej sady

Metamodel Ecore diagramu inštrukčnej sady je oproti metamodelu štrukturálneho diagramu mnohonásobne zložitejší a komplexnejší. Z tohto dôvodu budú v práci uvedené len jednotlivé jeho fragmenty vysvetľujúce podstatné vzťahy. Kompletný diagram je umiestnený na doprovodnom digitálnom nosiči [5] a je možné ho zobraziť v rámci platformy eclipse.

Fragment diagramu tried metamodelu inštrukčnej sady je znázorňený na obrázku 6.5. Analogicky ako v štrukturálnom diagrame predstavuje plátno v diagrame inštrukčnej sady trieda *Canvas*. S ohľadom na sémantiku jazyka ISAC bolo potrebné navrhnúť metamodel tak, aby bol prienik s gramatikou jazyka ISAC maximálny. Výsledkom sú tri typy operácií: *CAOperation*, *CROperation* a *E_Operation*, ktoré dedia od spoločnej nadtriedy *Operation*. Operation je agregačná trieda a triedy Activation, Behaviour a Expression sú konštitučné. Typ operácie *CAOperation* vždy obsahuje jednu sekciu *Assembler* a *Coding*. Operácia typu *E_Operation* je prázdnou operáciou, tj. neobsahuje žiadnu sekciu. Pretože nadtriedou operácie typu *E_Operation* je trieda Operation, môže tento typ operácie obsahovať prvky Activation, Behaviour a Expression. Špeciálnym typom je operácia *CROperation*, ktorá obsahuje sekcie *Structure* a *CodingRoot*. Kardinalita vzťahu medzi triedou Canvas a CROperation udáva, že diagram musí obsahovať maximálne jednu inštanciu operácie tohto typu.

Operácie obsahujú zložené oddiely, do ktorých sa namapujú sekcie Assembler, Coding,

[5] Štruktúra a obsah nosiča je uvedená v dodatkoch.



Obr. 6.5: Časť z Ecore metamodelu diagramu inštrukčnej sady.

Structure a CodingRoot. Zložené oddiely bližšie vysvetľuje kapitola 4.1. Je žiadúce, aby každá sekcia (Assembler, Coding atď.) bola vytvorená automaticky pri vytváraní inštancie operácie. Najjednoduchším spôsobom by bolo zmeniť typ kardinality z pôvodnej hodnoty *0..1* na **1** alebo zmeniť vzťah agregácie na kompozíciu. Framework GMF to ale neumožňuje. Preto sa používajú tzv. *"Edit Helpers"* [6], ktoré dokážu modifikovať príkazy na vytváranie prvkov. Nasledujúci príklad ilustruje automatické vytvorenie sekcie Assembler a Coding v operácii typu CAOperation:

```
/**
 * @generated NOT
 */
@Override
protected ICommand
        getConfigureCommand(ConfigureRequest req) {
```

[6]Modifikácia správania typu metamodelu je definovaná v jeho "edit helper". Edit helper je továreň na editačné príkazy. Tieto príkazy sú inštanciované v odpovedi na žiadosť modifikácie objektu modela, ktorého *EClass* odpovedá typu metamodelu.

```
        CAOperation op =
                (CAOperation) req.getElementToConfigure();
        Assembler asm = InstructionSetFactory.
                        eINSTANCE.createAssembler();
        op.setAssember(asm);
        Coding cod = InstructionSetFactory.
                        eINSTANCE.createCoding();
        op.setCoding(cod);
        return super.getConfigureCommand(req);
}
```

Z parametru sa získa objekt operácie. Pomocou továrenskej metódy sa vytvoria objekty sekcie Assembler a Coding, priradia sa operácii a zavolá sa pôvodný konfiguračný príkaz operácie. Pokiaľ sa zmení doménový model, framework GMF automatický obnoví odpovedajúce pohľady v prípade, že je nad daným prvkom nainštalovaná politika *"CanonicalEditPolicy"*. Terminológia *"Canonical"* slúži na popísanie kontajnera, ktorý udržuje pohľad na sémantické data synchronizovaný s potomkami sémantických dát. Pohľad sa po opätovnom otvorení diagramu neobnoví správne, ak nie je opravená oficiálna chyba GMF: 281014.

Automatickým vytvorením sekcií Assembler, Coding, Structure a CodingRoot je udržovaná sémantická správnosť modelu. Aby nebola integrita modelu porušená je potrebné zabrániť užívateľovi zmazať jednotlivé sekcie z operácie. Stačí v metóde *getDestroyElementCommand* vrátiť inštanciu príkazu, ktorý nie je možné vykonať a užívateľ nebude mať príležitosť vymazať sekciu pomocou kontextového menu. Na nasledujúcom príklade je ukážka úpravy metódy getDestroyElementCommand:

```
/**
 * @generated NOT
 */
protected ICommand
        getDestroyElementCommand(DestroyElementRequest req) {
        // Bráni vymazaniu tvaru
        return UnexecutableCommand.INSTANCE;
}
```

Prvky umiestnené v diagrame reagujú na niektoré preddefinované klávesové skratky. Napríklad po označení prvku v diagrame je možné stlačením klávesy *del* vybraný prvok odstrániť. Nainštalovaním politiky "ComponentEditPolicy" a modifikovaním metódy *getCommand* sa dá odchytiť udalosť stlačenia klávesy del a vrátiť príkaz, ktorý nie je možné vykonať. Detaily implementácie sú v nasledujúcom príklade:

```
/**
 * @generated NOT
 */
protected void createDefaultEditPolicies(){
    installEditPolicy(EditPolicyRoles.CREATION_ROLE,
            new CreationEditPolicy());
    super.createDefaultEditPolicies();
    installEditPolicy(EditPolicyRoles.SEMANTIC_ROLE,
            new SchoolItemSemanticEditPolicy());
```

```
    installEditPolicy(EditPolicy.LAYOUT_ROLE,
            createLayoutEditPolicy());
    installEditPolicy(EditPolicy.COMPONENT_ROLE,
            new ComponentEditPolicy() {

        public Command getCommand(Request request){
            //Ak užívateľ stlačí tlačítko delete,
            //tvar sa nevymaže
            if (request instanceof GroupRequestViaKeyboard
                    && RequestConstants.
                            REQ_DELETE.equals(request.getType())){
                return UnexecutableCommand.INSTANCE;
            }
            return super.getCommand(request);
        }
    });
}
```

6.2.1 Activation, Behaviour, Expression

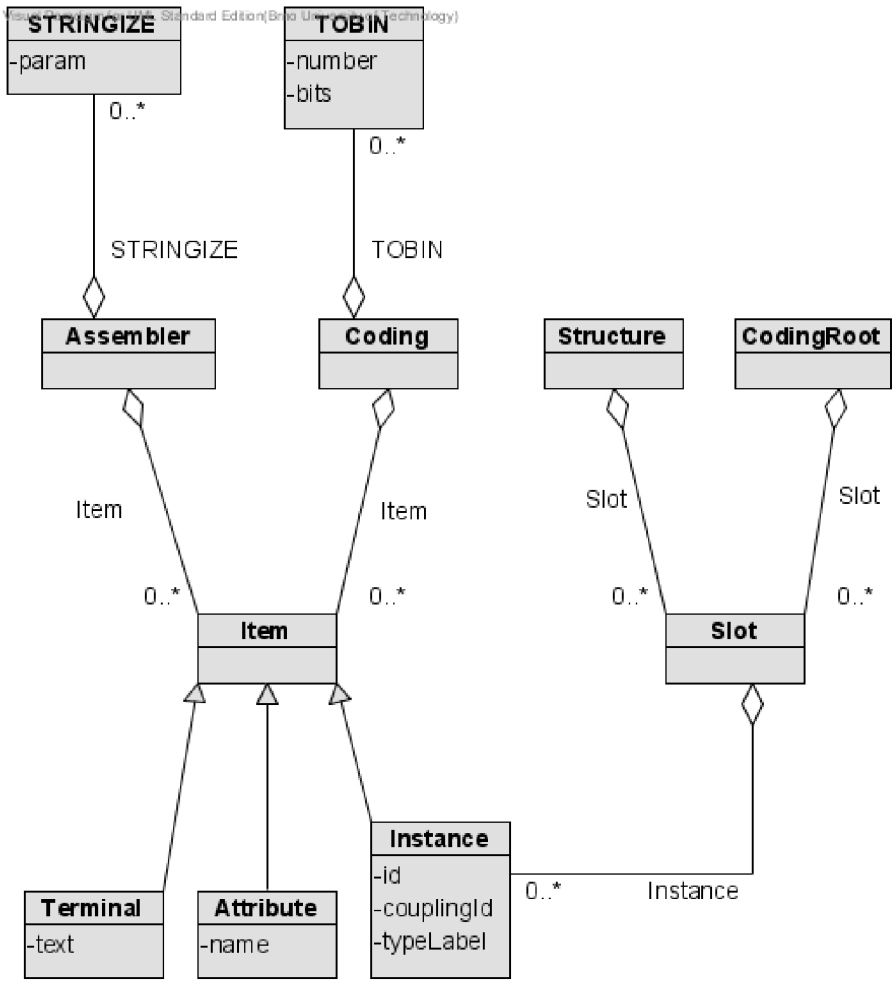
Funkciou sekcie activation je plánovanie rôznych operácií v dôsledku operácie, ktorá obsahuje sekciu activation. Sekcia behaviour obsahuje postupnosť akcií opísaných v jazyku ANSI C. Postupnosť akcií vyjadruje správanie operácie. Sekcia expression má podobný význam ako sekcia behaviour, ale obsahuje len návratovú hodnotu. Bližšie informácie o tejto problematike sa nachádzajú v kompletnej dokumentácií jazyka ISAC v [9]. Diagramy pre vizualizáciu týchto sekcií zatiaľ neboli navrhnuté, ale pretože sekcie activation, behaviour a expression predstavujú dôležitú súčasť jazyka ISAC, bolo nevyhnutné umožniť ich editáciu. Zvolené riešenie dovoľuje užívateľovi v diagrame inštrukčnej sady do prvku operácie vložiť z palety nástrojov prvky activation, behaviour a expression. Vložený prvok má tvar červeného štvorca, v ktorom je čierne písmeno označujúce sekciu, o ktorú sa jedná. Dvojklikom nad týmto prvkom sa otvorí súbor s obsahom zvolenej sekcie. Dočasné riešenie využíva vstavaný textový editor v eclipse, ktorý nezvýrazňuje syntax. V dohľadnej dobe budú navrhnuté nové diagramy pre spomínané sekcie, preto nebola pozornosť sústredená práve na editor týchto sekcií.

Súbor s obsahom sekcie je s doménovým modelom asociovaný pomocou atribútu *filename*, ktorý obsahuje cestu k súboru v projekte. Názov súboru závisí od názvu operácie, do ktorej sekcia patrí. V prípade, že operácia nemá pridelený názov, vygeneruje sa náhodny názov súboru automaticky pomocou *EcoreUtils* generátoru.

6.2.2 Prvky Instance, Terminal, Attribute

Operácie v závislosti od typu môžu obsahovať sekcie assembler, coding, alebo structure a codingroot. Operácia typu CAOperation obsahuje sekcie assembler a coding, do ktorých je možné umiestňovať prvky *Instance*, *Attribute* a *Terminal*. Prvým spôsobom ako vyriešiť vzťah sekcie assembler a coding k jej prvkom je vytvoriť pre každú triedu (Instance, Attribute a Terminal) v doménovom modele vzťah agregácie medzi triedou sekcie a triedou prvku. Uvedné riešenie má jeden základný problém. Pre jazyk ISAC je podstatné, v akom poradí sa prvky nachádzajú. Framework GMF ale pri serializácií do medziformátu XMI

nezoradí prvky v takom poradí, v akom boli postupne pridávané, ale zoradí ich podľa typu. Na obrázku 6.6 je zobrazená časť metamodelu Ecore diagramu inštrukčnej sady opisujúca správne riešenie uvedeného problému.



Obr. 6.6: Časť metamodelu Ecore inštrukčnej sady.

Trieda *Item* je spoločným nadtypom pre prvky Instance, Attribute a Terminal. Spoločný nadtyp už framework GMF interpretuje správne a vytvorený model serializuje do potrebného tvaru. Operácia typu CROperation obsahuje sekcie structure a codingroot, do ktorých je možné umiestňovať sloty. Slot je jednotka, ktorá združuje skupinu prvkov Instance. Odpovedajúca časť metamodelu Ecore je na obrázku 6.6.

Prvok **Instance** predstavuje inštanciu operácie. Inštancia musí mať určený identifikátor a typ. Typovanie inštancií bude vysvetlené neskôr. Gramatika jazyka ISAC vyžaduje, aby sa do operácie pridávali inštancie vždy párovo, tzn. že sa prvok inštancie bude vyskytovať v obidvoch sekciách - assembler aj coding. Editor musí zaručiť konzistenciu vytváraného modelu, preto ak návrhár pridá prvok inštancie do jednej zo sekcií, musí sa v druhej automaticky vytvoriť ekvivalentná inštancia. Atribút *couplingId* v triede Instance doménového modelu sa používa na spárovanie inštancií. Tento atribút sa používa interne a pre koncového užívateľa nie je prístupný. Vytvorenie párovej inštancie je realizované pomocou príkazu

InstanceCreateCommand, ktorý je zobrazený na nasledujúcom príklade.

```
/**
 * @generated NOT
 */
protected CommandResult doExecuteWithResult(...)
    throws ExecutionException {
    ...
    Instance newElement = InstructionSetFactory.eINSTANCE
            .createInstance();
    Instance newElement2 = InstructionSetFactory.eINSTANCE
            .createInstance();
    newElement.setCouplingId(uuid);
    newElement2.setCouplingId(uuid);

    Assembler asm_owner = (Assembler) getElementToEdit();
    asm_owner.getItem().add(newElement);

    Coding cod_owner =
            ((CAOperation) asm_owner.eContainer()).getCoding();
    cod_owner.getItem().add(newElement2);
    ...
    return CommandResult.newOKCommandResult(newElement);
}
```

Továrenskou metódou *createInstance* sa vytvoria dva prvky, jeden pre sekciu assembler a druhý pre sekciu coding. Vytvorené prvky sa previažu spoločným *couplingId*, ktoré je vygenerované prostredníctvom EcoreUtil.generateUUID() generátoru. Napokon sú novovytvorené prvky vložené do jednotlivých sekcií. Na záver sa vykoná samotný príkaz. Odstránenie inštancie z modelu je realizované prostredníctvom príkazu *DestroyInstanceElementCommand*, ktorý vyhľadá obidva združené prvky podľa couplingId a odstráni ich.

Terminálny prvok (**Terminal**), ktorý sa nachádza v sekcii assembler je reťazcom. Ak je terminál umiestnený do sekcie coding, musí byť zadaný v binárnej podobe. Prvok **TOBIN** je v jazyku ISAC makro s dvoma parametrami, ktoré kóduje čísla do binárnej podoby. Prvým parametrom je číslo, ktoré sa kóduje. Druhý parameter udáva počet bitov, na ktorých bude číslo zakódované. Odpovedajúca Trieda v metamodeli Ecore je na obrázku 6.6 a obsahuje dva atribúty: number a bits. Obidva atribúty je možné parametrizovať po označení prvku TOBIN v diagrame na záložke properties. Druhým spôsobom parametrizácie je priama editácia štítka (label) v uzle TOBIN. Najčastejšie sa v GMF mapuje jeden atribút na jeden štítok. Obrázok 6.7 zobrazuje nastavenie "Feature Label Mapping" v modeli mapovania. V tomto prípade je nutné namapovať obidva atribúty na jeden štítok. V mapovaní "Feature Label Mapping" sa prostredníctvom vlastností "Features to display" a "Features to edit" zvolia atribúty z doménového modela, ktoré bude možné zobraziť a editovať v rámci zvoleného štítka. Vlastnosť "Diagram Label" vyberá štítok z modelu gmfgraph (grafická definícia prvkov). Jedná sa o štítok, na ktorý sa namapujú vybrané atribúty. Najdôležitejšie sú vlastnosti, ktoré patria do kategórie "Visual representation". Určujú formát zobrazenia. *MessageFormat* poskytuje prostriedky na výrobu zreťazených správ nezávisle od jazyka. Výsledný štítok prvku TOBIN má tvar: **TOBIN(x , y)**, pričom argumenty x a y môže

užívateľ parametrizovať. Realizácia využíva práve MessageFormat na sformátovanie štítka. Predpis vzoru MessageFormat je nasledujúci:

```
MessageFormatPattern:
        String
        MessageFormatPattern FormatElement String

FormatElement:
    { ArgumentIndex }
```

Pri aplikovaní tohto predpisu na príklad z obrázka 6.7 (vlastnosť *Edit Pattern* a *View Pattern*) sa miesto argumentu {0} dosadí atribút *number* a miesto argumentu {1} atribút *bits*. Na začiatok reťazca sa doplní "TOBIN(" a na koniec znak ")". Analogickým postupom sa zostrojí prvok *STRINGIZE*. STRINGIZE je v jazyku ISAC makro s jedným parametrom, ktoré prevedie zadaný parameter na reťazec.

Property	Value
Domain meta information	
Features to display	TOBIN.number:EInt, TOBIN.bits:EInt
Features to edit	TOBIN.number:EInt, TOBIN.bits:EInt
Misc	
Diagram Label	Diagram Label TobinLabel
Read Only	false
Visual representation	
Edit Method	MESSAGE_FORMAT
Editor Pattern	
Edit Pattern	TOBIN({0}, {1})
View Method	MESSAGE_FORMAT
View Pattern	TOBIN({0}, {1})

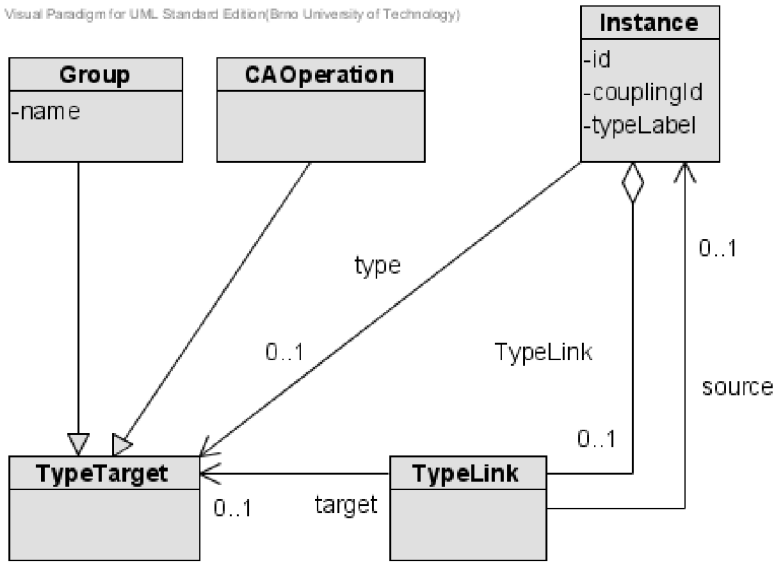
Obr. 6.7: Mapovanie atribútov prvku TOBIN.

6.2.3 Typovanie inštancií

V nasledujúcom texte bude opísaný spôsob typovania inštancií. Typ inštancie môže byť skupina alebo operácia typu CAOperation. Z užívateľského pohľadu existujú dva spôsoby ako definovať typ inštancie. Prostredníctvom nástroja "Specify Type" z palety nástrojov je možné prepojiť inštanciu so skupinou alebo operáciou. Medzi uzlom inštancie a uzlom skupiny, prípadne operácie vznikne spoj, ktorý definuje typ inštancie. Tento prístup je pre návrhára názorný, ale pre rozsiahle modely nie je dostatočne vhodný, pretože by v diagrame vzniklo príliš veľa spojov. Z tohto dôvodu existuje druhý spôsob určenia typu inštancie - pomocou dvojbodky. Zápis vyzerá následovne: **ID ":" ID**, kde prvé ID je identifikátor inštancie a druhé ID je typ.

Na obrázku 6.8 je časť metamodelu Ecore diagramu inštrukčnej sady opisujúca spracovanie typu inštancie. *TypeTarget* je nadtriedou pre triedy Group a CAOperation. Znamená to, že iba prvky, ktoré majú nadtriedu TypeTarget môžu určovať typ inštancie. Trieda *TypeLink* reprezentuje spoj určujúci typ (podobne ako pri štrukturálnom diagrame predstavuje spoj trieda *ProvidedInterface* a *RequiredInterface*). Asociácia (*type*) medzi triedou

Instance a TypeTarget sa používa na definovanie typu prostredníctvom notácie s dvojbodkou. Užívateľovi nie je umožnené definovať typ priamym zápisom s dvojbodkou, pretože by musel byť každý vstup zbytočne kontrolovaný. Aby bol udržaný vytváraný model v konzistencii, tak návrhár vyberie v pohľade "Properties" typ zo zoznamu dostupných skupín a operácií. Framework GMF tento zoznam vytvorí automaticky na základe asociácie type medzi triedou Instance a TypeTarget.



Obr. 6.8: Časť metamodelu Ecore inštrukčnej sady.

Konzistencia modelu musí byť zachovaná aj pri zmene názvu operácie, tzn. že pri zmene názvu operácie sa musí upraviť atribút *typeLabel* a referencia na typ. Najvhodnejšou realizáciou je využitie návrhového vzoru pozorovateľ. EMF adaptér plní úlohu pozorovateľa, ale okrem pozorovania umožňuje rozšíriť správanie objektov bez použitia dedičnosti. Inicializácia adaptéra je zobrazená na nasledujúcom príklade:

```
/**
 * @generated NOT
 */
protected InstanceImpl() {
    super();
    adapter = new AdapterImpl(){
          public void notifyChanged(Notification notification)
          {
            if(type instanceof Operation)
               setTypeLabel(((Operation)type).getName());
            if(type instanceof Group)
               setTypeLabel(((Group)type).getName());
          }
    };
}
```

Uvedený adaptér pozoruje zmeny atribútov skupin a operácií a odpovedajúco upravuje štítky. Každý EMF objekt obsahuje zoznam adaptérov, ktoré ho pozorujú. Volanie metódy *eAdapters()* vráti zoznam nainštalovaných adaptérov nad daným objektom. Adaptér vytvorený v konštruktore inštancie je potrebné nainštalovať na objekt, ktorý zastupuje typ inštancie. Metóda *setType*, zobrazená na nasledujúcom príklade, je volaná vždy pri zmene typu inštancie.

```
/**
 * @generated NOT
 */
public void setType(TypeTarget newType) {
    ...
    if(oldType != null)
        oldType.eAdapters().remove(adapter);

    newType.eAdapters().add(adapter);
    ...
}
```

Pri nastavovaní typu sa vždy volá metóda setType. Ak sa zmení typ je nutné odstrániť adaptér z objektu, ktorý určoval typ inštancie a zároveň je potrebné nainštalovať adaptér na nový objekt.

Typom inštancie nesmie byť operácia, v ktorej sa daná inštancia nachádza. Editor ošetruje túto situáciu pomocou obmedzení spojov. Obmedzenia spojov sú vysvetlené v kapitole 4.5. Primárne sa na zápis podmienok obmedzení používa Object Constrained Language. Základy jazyka OCL rozoberá kapitola 2.6. Model mapovania (gmfmap) obsahuje mapovanie spoja TypeLink. Vytvorením obmedzenia spoja (Link Constraint) pre TypeLink je možné obmedziť vytváranie spojov len medzi povolenými prvkami. Podmienka zapísaná v jazyku ocl, ktorá obmedzuje vytváranie TypeLink spojov je na nasledujúcom príklade:

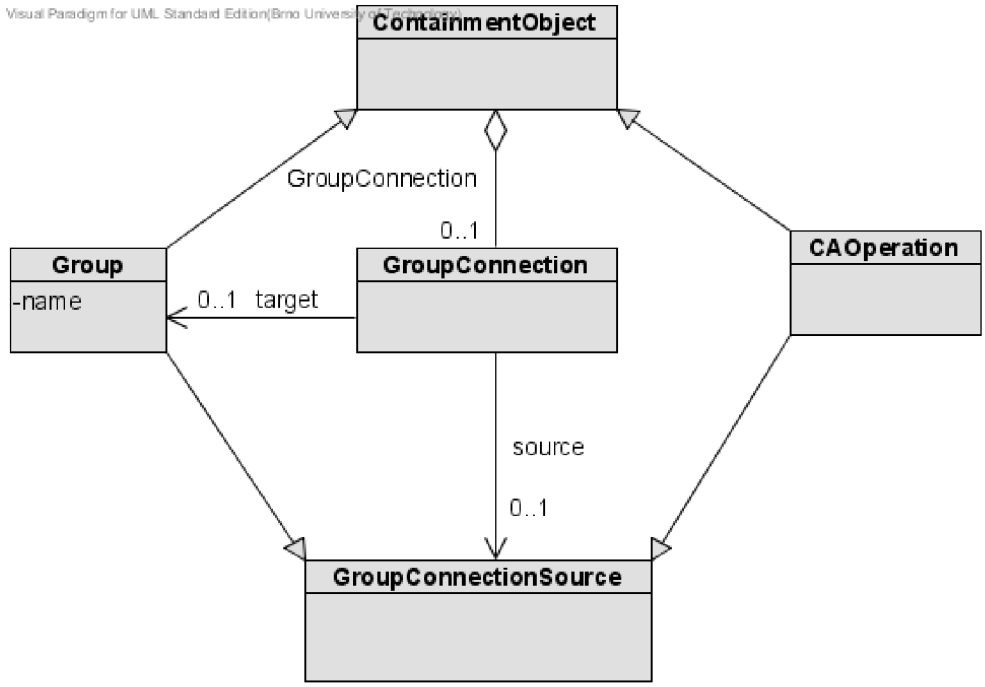
```
(self.oclIsTypeOf(Group)) or
(self.oclAsType(CAOperation)->Assember.Item->
                        forAll(item | item <> oppositeEnd))
```

Jedná sa o obmedzenie pre "targetEnd", preto je kontext nastavený na cieľový prvok, na ktorý sa odkazuje premenná self. Premenná oppositeEnd ukazuje na zdrojový prvok - inštanciu, z ktorej je vedený spoj. Hore uvedená podmienka OCL musí zaručiť, aby cieľovým prvkom bola skupina alebo operácia typu CAOperation a zároveň musí garantovať, aby cieľovým prvkom nebola operácia, v ktorej sa nachádza inštancia. Operácia *oclIsTypeOf* overí typ cieľového prvka. Ak je cieľový prvok typu skupina, výraz vráti "true" a spoj je realizovaný. V prípade, že cieľový prvok nie je skupina, musí platiť, že je typu CAOperation. Pomocou operácie "forAll" sa prejde kolekcia všetkých položiek v sekcii assembler, pričom musí byť zaručené, že žiadna položka nie je inštancia, z ktorej je spoj vedený.

6.2.4 Hierarchia skupín a operácií

Diagram inštrukčnej sady umožňuje prostredníctvom nástroja "Group Connection" z palety editora prepájať skupiny a operácie. Vzniká hierarchia skupín a operácií. Operácie sa zaraďujú do skupín, pričom skupiny môžu obsahovať iba operácie typu CAOperation alebo iné

skupiny. Oproti klasickému prípadu vytvárania spojov, pri ktorom sa spoj vytvára medzi dvoma rôznymi prvkami, je v tomto prípade potrebné upraviť metamodel, pretože editor musí umožniť viesť spoj od operácie ku skupine a taktiež od skupiny ku skupine. Časť metamodelu Ecore opisujúca spomínanú problematiku je na obrázku 6.9.



Obr. 6.9: Časť metamodelu Ecore inštrukčnej sady.

Spoj je namapovaný na triedu *GroupConnection*. Každá trieda, ktorá je namapovaná na spoj musí vo frameworku GMF odpovedať určitým pravidlám. Musí obsahovať dve asociácie a jednu agregáciu. Asociácia target vytvára vzťah s triedou cieľového prvku a asociácia source vytvára vzťah s triedou zdrojového prvku. Agregácia *GroupConnection* určuje prvok, z ktorého je spoj vedený. Aby bolo možné spoj viesť z prvku skupiny aj z prvku operácie, bola vytvorená zastrešujúca trieda *ContainmentObject*, pretože GMF neumožňuje vytvoriť dve agregácie "GroupConnection" medzi triedou GroupConnection a Group a medzi triedou GroupConnection a CAOperation. Trieda ContainmentObject je nadtriedou pre triedy Group a CAOperation. Analogicky bola vytvorená trieda *GroupConnectionSource*, pretože obdobne nie je v GMF možné vytvoriť dve asociácie source. Realizované riešenie umožňuje kontrolovať vytváranie spojov na úrovni metamodelu.

6.2.5 Drag & Drop prvkov

Základná dátova štruktúra, do ktorej sa ukladajú prvky zložených oddielov GMF, je zoznam. Prvky vkladané do zložených oddielov sú štandardne pridávané na koniec zoznamu a užívateľ nemôže meniť ich pozíciu. To predstavuje dva základné problémy. Užívateľské rozhranie editora neumožňuje určiť poradie jednotlivých prvkov inak ako vymazaním a následným vložením prvkov v správnom poradí. Takéto riešenie nie je dostatočne prívetivé a intuitívne a práca s editorom nie je pre užívateľa efektívna a pohodlná. Druhým podstatným problémom je, že z pohľadu jazyka ISAC je pozícia jednotlivých prvkov dôležitá

a editor teda musí umožniť meniť pozíciu jednotlivých prvkov. V nasledujúcom texte bude ukázaný spôsob riešenia. GMF zobrazuje zoznam prvkov v zložených oddieloch štandardne vo vertikálnom smere. S ohľadom na navrhnutý diahram inštrukčnej sady sa vyžaduje, aby boli prvky umiestnené v horizontálnom smere. Metóda *createFigure* zloženého oddielu zabezpečuje jeho "layout". Najprv sa vytvorí základný layout, ktorý sa následne parametrizuje a priradí tvaru. Metóda *setHorizontal(true)* zariadi horizontálne umiestňovanie prvkov. Telo metódy *createFigure* je ilustrované na nasledujúcom príklade:

```
/**
 * @generated NOT
 */
public IFigure createFigure() {
    ResizableCompartmentFigure rcf =
            (ResizableCompartmentFigure) super.createFigure();
    FlowLayout layout = new FlowLayout();
    layout.setMajorSpacing(getMapMode().DPtoLP(5));
    layout.setMinorSpacing(getMapMode().DPtoLP(5));
    layout.setHorizontal(true);

    rcf.getContentPane().setLayoutManager(layout);
    rcf.setTitleVisibility(false);
    return rcf;
}
```

Na úpravu funkčnosti základného editora slúžia politiky. Každý prvok má kontrolér (EditPart), do ktorého sa inštalujú editačné politiky (EditPolicies). Je to mechanizmus, ktorý umožňuje spravovať požiadavky a vytvárať z nich odpovedajúce príkazy. Editačné politiky sa kategorizujú do rozličných rolí. Rola je zodpovedná za uspokojenie požiadavkov vrátením odpovedajúceho príkazu. Napríklad rola *CREATION_ROLE* sa používa v politike, ktorá rozumie požiadavkom na vytváranie prvkov. Politika *CompartmentChildEditPolicy* plní rolu CREATION_ROLE a umožňuje vkladať prvky na presnú pozíciu v zloženom oddiele. Funkcionalitu Drag & Drop implementuje *CompartmentEditPolicy*, ktorá plní rolu DRAG_DROP_ROLE. Ukážka inštalácie politík pre zložený oddiel prvku assembler je na nasledujúcom príklade:

```
/**
 * @generated NOT
 */
protected void createDefaultEditPolicies() {
    ...
    installEditPolicy(EditPolicyRoles.CREATION_ROLE,
            new CompartmentChildCreationEditPolicy());
    installEditPolicy(EditPolicyRoles.DRAG_DROP_ROLE,
            new CompartmentEditPolicy(
                InstructionSetPackage.Literals.ASSEMBLER_ITEM));
}
```

6.2.6 Zásuvný modul kontextového menu

Zásuvný modul je najmenšia jednotka poskytujúca určitú funkcionalitu, ktorá môže byť vyvíjaná a dodávaná oddelene. Platforma Eclipse obsahuje stovky zásuvných modulov, ktoré spolu vytvárajú komplexný nástroj Eclipse. Zásuvné moduly medzi sebou komunikujú prostredníctvom rozšírení a bodov rozšírení. Každý zásuvný modul má vlastný manifest súbor (plug-in.xml), ktorý definuje prepojenie medzi jednotlivými zásuvnými modulmi. Model prepojenia je jednoduchý: zásuvný modul deklaruje určitý počet bodov rozšírení (**extension points**) a nejaký počet rozšírení (**extensions**) k jednému alebo viacerým bodom rozšírení v iných zásuvných moduloch. Bod rozšírenia jednoducho hovorí: "Mám tu pre Teba slot, aby si mi poskytol nové správanie." a rozšírenie hovorí: "Tu je nové správanie, ktoré si žiadal.". Zásuvné moduly môžu zastupovať obidve role - deklarovanie bodu rozšírenia a poskytovanie k tomuto bodu rozšírenie. Definovanie bodu rozšírenia je podobné definovaniu nejakého API [7]. Jediný rozdiel spočíva v tom, že bod rozšírenia je deklarovaný za použitia XML, namiesto signatúry kódu. Hlavným cieľom tohto spôsobu je, že užívateľ neplatí pamäťové a výkonnostné penalty za zásuvné moduly, ktoré sú nainštalované, ale sa nepoužívajú. Vysvetľujúca podstata platformového modelu rozšírení dovoľuje jadru rozhodnúť, ktoré rozšírenia a body rozšírenia sú poskytované zásuvným modulom bez toho, aby daný zásuvný modul bolo potrebné spustiť. Preto môžu byť nainštalované mnohé zásuvné moduly, ale žiaden z nich nebude spustený dovtedy, kým funkcia poskytovaná zásuvným modulom nebude požadovaná podľa aktivity užívateľa. Toto je dôležitá vlastnosť v poskytovaní robustnej platformy [14].

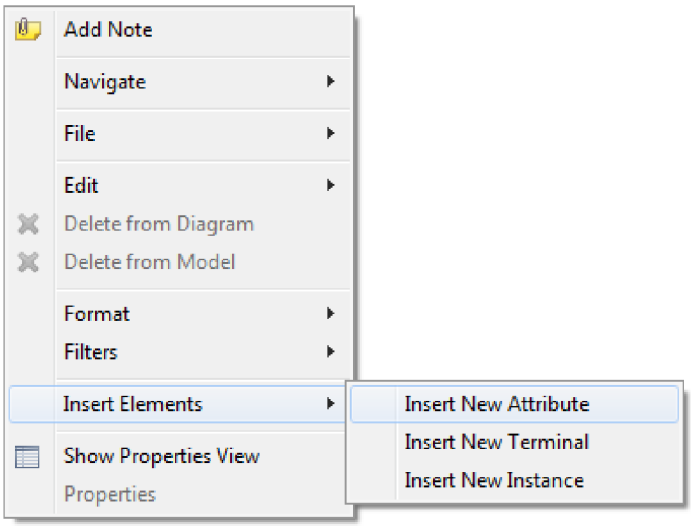
Momentálne je pridávanie prvkov do sekcií assembler a coding nepraktické a zdĺhavé, pretože užívateľ musí najprv vybrať prvok z palety nástrojov a potom kliknúť na uzol, kam bude prvok následne vložený. Ak je potrebné vložiť niekoľko prvkov za sebou, stáva sa toto riešenie pre užívateľa neefektívnym. Najlepším riešením je vytvorenie kontextového menu, ktoré umožní užívateľovi pridávať prvky do vybraných sekcií priamo z vybraného uzlu. Kontextové menu v platforme Eclipse je realizované pomocou mechanizmu bodov rozšírení. V nasledujúcom texte budú vysvetlené body rozšírenia, ktoré sa používajú na implementáciu kontextového menu. Podrobné informácie o všeobecnej problematike implementácie zásuvných modulov sú vysvetlené v [7]. Bod rozšírenia *<extension point = "org.eclipse.ui.popupMenus">* umožňuje upravovať kontextové menu pohľadov a editorov. Rozšíriť kontextové menu je možné dvoma spôsobmi. S využitím tzv. "objectContribution" alebo prostredníctvom "viewerContribution". ViewerContribution rozširuje kontextové menu pohľadu alebo editora prostredníctvom jeho ID. Napríklad by bolo týmto spôsobom možné rozšíriť kontextové menu pohľadu "Package Explorer". ObjectContribution asociuje kontextové menu s objektom zvolenej triedy. Tento prístup umožňuje zobrazovať kontextové menu iba nad takými prvkami v diagrame, ktoré sú inštanciou triedy definovanej v bode rozšírenia pre kontextové menu. Diagram inštrukčnej sady používa na rozšírenie kontextového menu objectContribution, pretože je požadované, aby bolo kontextové menu asociované s objektom sekcie assembler a coding v uzle operácie. Zásuvný modul *InstructionSet.diagram.custom* implementuje rozšírenie kontextového menu o akcie pridávania prvkov do sekcie assembler a coding. Na nasledujúcom príklade je zobrazený úsek manifest súbora, ktorý implementuje rozšírenie základného kontextového menu pre sekciu assembler:

```
<extension point="org.eclipse.ui.popupMenus">
   <objectContribution
```

[7] API - Application programming interface.

```
        objectClass=
            "InstructionSet.diagram.edit.parts.AssemblerEditPart"
        id="InstructionSet.diagram.custom.contribution1">
      <menu
          label="Insert Elements"
          path="additions"
          id="InstructionSet.diagram.custom.menu1">
          <separator
              name="group1">
          </separator>
      </menu>
      <action
          label="Insert New Instance"
          class="instructionset.diagram.custom.popup.
                                        actions.InsertInstanceAction"
          menubarPath="InstructionSet.diagram.custom.menu1/group1"
          enablesFor="1"
          id="InstructionSet.diagram.custom.InsertInstanceAction">
      </action>
       ...
    </objectContribution>
     ...
</extension>
```

Atribút *objectClass* určuje triedu objektu, pre ktorý sa zobrazí rozšírené kontextové menu. Pre sekciu assembler je to trieda *InstructionSet.diagram.edit.parts.AssemblerEditPart*. Prostredíctvom prvku *menu* sa definuje nová položka pre kontextové menu. Názov položky určuje prvok *label*. Element *separator* vytvorí nové submenu pre pridanú položku menu, do ktorého sa umiestnia položky na pridávanie prvkov Instance, Attribute a Terminal. Ukážka kontextového menu je na obrázku 6.10.



Obr. 6.10: Ukážka kontextového menu.

Vytvorené menu sa síce zobrazí, ale zatiaľ nemajú jednotlivé položky pridelené žiadne akcie. Prvok *action* definuje akciu, ktorá sa po výbere vykoná. Atribút *class* odkazuje na triedu, ktorá implementuje rozhranie *IObjectActionDelegate* a definuje akciu, ktorá sa má vykonať. Metóda *void selectionChanged(IAction action, ISelection selection)* zobrazuje menu v závislosti od vybraného typu objektu. Zavolaním metódy *void run(IAction action)* sa spustí akcia. Na vytváranie prvkov sa používa architektúra Eclipse Command Framework. Základným stavebným kameňom tejto architektúry je príkaz, ktorý po spustení vykoná činnosť definovanú požiadavkou. V prvom kroku akcie sa vytvorí požiadavka typu "CreateViewRequest", ktorá definuje prvok ktorý budeme vytvárať, napríklad inštanciu. Z vytvorenej požiadavky sa zostaví príkaz, ktorý po spustení vykoná operáciu opísanu prostredníctvom požiadavky a v diagrame vznikne nový prvok.

6.3 Kontrola platnosti modelu

V predchádzajúcich kapitolách bol predstavený metamodel Ecore štrukturálneho diagramu a diagramu inštrukčnej sady, v ktorom sú zachytené hlavné vzťahy medzi doménovými prvkami jednotlivých diagramov. Pre návrhára, ktorý používa grafický editor je podstatné, aby mu bolo zabránené vytvoriť model, ktorý nie je v konzistentnom stave. V niektorých prípadoch však môže nastať situácia, že sa model dostane do nekonzistentného stavu. Napríklad užívateľ vytvorí v diagrame inštrukčnej sady operáciu a nepriradí jej názov. Tento nekonzistentný stav nie je možné kontrolovať pomocou metamodelu Ecore. Otázkou teraz zostáva ako riešiť momentálne porušenie konzistencie. Zabrániť užívateľovi v práci, kým nevyplní názov operácie, by v tomto prípade nebolo vhodným riešením, pretože by celková práca s editorom stratila komfort. Vhodnejším riešením by bola kontrola platnosti modelu až v dobe, keď to je nevyhnutné. Napríklad pred transformáciou diagramu na zdrojové súbory jazyka ISAC. Ak by sa v tomto bode zistila nekonzistencia modelu, transformácia by sa prerušila a chyby by sa prezentovali vhodným spôsobom užívateľovi.

Dômyselnú kontrolu platnosti modelu zabezpečuje validačný framework, ktorý je vysvetlený v kapitole 3.3. V nasledujúcom texte bude pozornosť sústredená na implementáciu invariantov v jednotlivých diagramoch, ktoré kontrolujú platnosť modela. V texte budú popísané predovšetkým podmienky invariantov. Celková štruktúra metódy invariantu je opísaná v kapitole 3.3. Každá operácia umiestnená na plátno diagramu musí mať pridelený **unikátny** názov. V tomto prípade nie je potrebné vytvárať pre kontrolu unikátnosti názvu invariant, pretože je možné využiť vlastnosť frameworku GMF, ktorá dovoľuje označiť atribút triedy v modeli Ecore ako identifikátor. Nastavenie vlastnosti ID atribútu *name* v triede *operation* na hodnotu "true" zaručí, že validačný framework automaticky pri spustení kontroly overí unikátnosť identifikátoru v celom diagrame. Kontrola unikátnosti názvu operácie neodhalí nekonzistentný stav, pri ktorom ostane atribút *name* nezadaný. V tomto prípade je nutné definovať invariant **hasName**, ktorý kontroluje hodnotu atribútu *name*. Podmienka invariantu *hasName* je nasledujúca:

```
if (this.name == null || this.name.isEmpty())
```

Pre inštancie vložené do operácie musí platiť, aby kontrola konzistencie zaručila, že inštancia má pridelený typ a identifikátor inštancie je unikátny v rámci operácie, v ktorej sa inštancia nachádza. Pre kontrolu unikátnosti identifikátoru inštancie nie je možné použiť mechanizmus, opísaný vyššie, ako v prípade kontroly unikátnosti názvu operácie, pretože neumožňuje zadať kontext, v ktorom sa má unikátnosť kontrolovať. Pretože ak by nastala

situácia, v ktorej existujú dve inštancie s rovnakým identifikátorom v dvoch rôznych operáciach, validačný framework by rozpoznal konflikt identifikátorov. Trieda inštancie má definované dva invarianty: **hasUniqueId** a **hasType**. Podmienka invariantu hasType je podobná ako pri invariante hasName. Atribút type sa testuje na hodnotu "null". Podmienka invariantu hasUniqueId je nasledujúca:

```
if (!hasUniqueId())
```

Podmienka využíva privátnu metódu, ktorá prostredníctvom iterátora prejde všetky inštancie v kontajneri a overí, že testovaná inštancia je v danom kontajneri jedinečná. Algoritmus metódy je zobrazený na nasledujúcom príklade.

```
private boolean hasUniqueId()
{
    EObject parent = this.eContainer();
    Iterator it = parent.eContents().iterator();
    while(it.hasNext())
    {
        Item item = (Item) it.next();
        if((item instanceof Instance)
                && !(item.equals(this))
                && !(parent instanceof Coding))
        {
            if(((Instance) item).getId().equals(this.getId()))
                return false;
        }
    }
    return true;
}
```

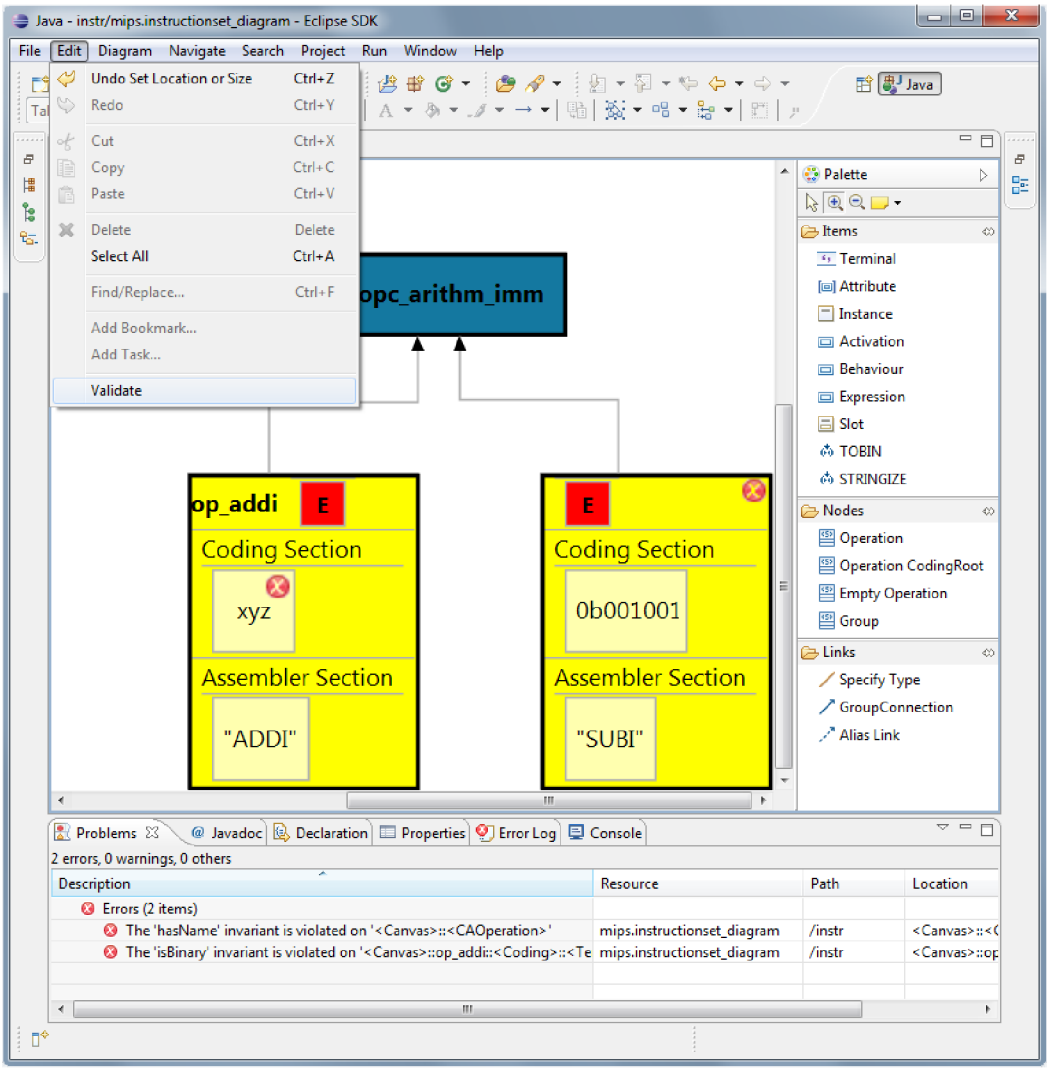
Prvok *Terminal* je možné umiestňovať do sekcie assembler alebo coding v operácii typu CAOperation. Tento prvok má jeden atribút s názvom *text*, ktorý predstavuje hodnotu terminálu. Ak je terminál umiestnený do sekcie assembler nadobúda jeho atribút hodnotu ľubovoľného reťazca. Pre terminál umiestnený do sekcie coding platí pravidlo, ktoré udáva, že hodnota atribútu musí byť zadaná v presne definovanom binárnom tvare. Model Ecore neumožňuje lexikálnu kontrolu atribútov, preto má trieda *Terminal* definovaný invariant **isBinary**, ktorý kontroluje lexikálnu správnosť atribútu *text*. Podmienka invariantu sa člení na niekoľko častí. V prvom kroku sa kontroluje, či sa terminál nachádza v sekcii coding. V nasledujúcom kroku sa pomocou regulárnych výrazov kontroluje hodnota atribútu *text*. Invariant je splnený v prípade, ak hodnota odpovedá požadovanému binárnemu tvaru odpovedajúcemu regulárnemu výrazu. Podmienka invariantu **isBinary** je nasledujúca:

```
if (this.eContainer() instanceof Coding) {

    Pattern p = Pattern.compile("0b[0-1]+");
    Matcher m = p.matcher(this.getText());
    if(m.matches())
        return true;
```

```
        //spracuj diagnostiku ...
    }
```

V predchádzajúcom texte boli predstavené hlavné invarianty zabezpečujúce konzistentnosť modela. Obrázok 6.11 ilustruje prezentáciu výsledkov neúspešnej validácie.



Obr. 6.11: Ukážka interpretácie výsledkov validácie.

Chyby sa zobrazujú v pohľade "Problems". Tento pohľad poskytuje informácie o invariante, ktorý bol porušený, o prvku, v ktorom nastala chyba a niekoľko dodatočných informácií popisujúcich chybu. Veľmi užitočnou vlastnosťou je spôsob dekorácie prvkov pomocou červeného kruhu. Tento kruh označuje prvok porušujúci konzistenciu. Užívateľ tým získava rýchly prehľad o výskyte chyby. Obidve metódy reprezentácie chýb musia byť povolené v modeli generátora [8]. Tabuľka 6.1 zobrazuje vlastnosti, ktoré je potrebné nastaviť

[8]Model generátora (***.gmfgen**) popisuje spôsob akým sa vygeneruje z vytvorených gmf modelov (model mapovania, definícia grafických prvkov atď.) výsledný editor a aké budú jeho vlastnosti. Bližšie informácie je možné nájsť v [3].

v modeli generátora, aby framework GMF správne interpretoval výsledky validácie. Užívateľ má možnosť spustiť kontrolu platnosti modelu prostredníctvom menu, tak ako to ilustruje obrázok 6.11.

Vlastnosť	Hodnota
Live Validation UI Feedback	true
Validation Decorators	true
Validation Enabled	true
Validation Decorator Provider Priority	Medium

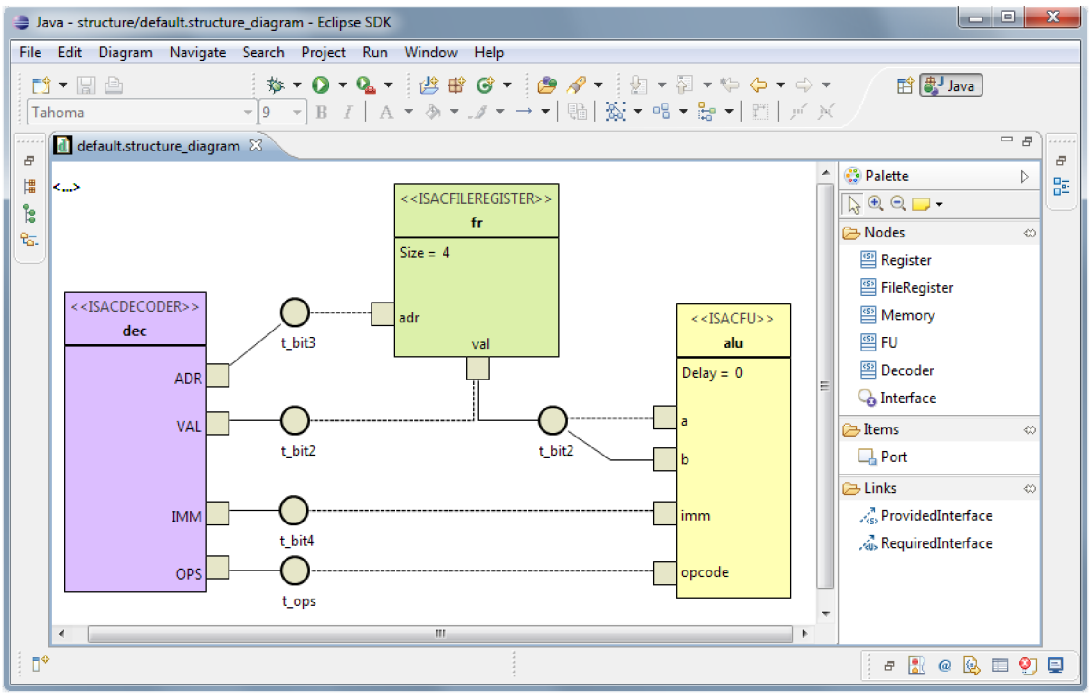
Tabuľka 6.1: Nastavenie vlastností v modeli generátora pre potreby validácie.

Kapitola 7

Výsledný editor

Kapitola 6 rozoberá implementáciu štrukturálneho diagramu a diagramu inštrukčnej sady. Výstupom implementácie je sada jedenástich zásuvných modulov do platformy Eclipse, ktoré tvoria výsledný editor. Vytvorený editor je plánované nasadiť ako grafický nástroj pre návrh čipu v projekte Lissom. Nesporná výhoda implementovaného riešenia spočíva v tom, že návrhár získa kompletný produkt postavený na platforme Eclipse, ktorý bude používať spoločne a na rovnakej úrovni so súčasnými nástrojmi projektu Lissom. Pre užívateľa teda nevznikne potreba používať nástroje tretích strán a jeho práca získa na komforte.

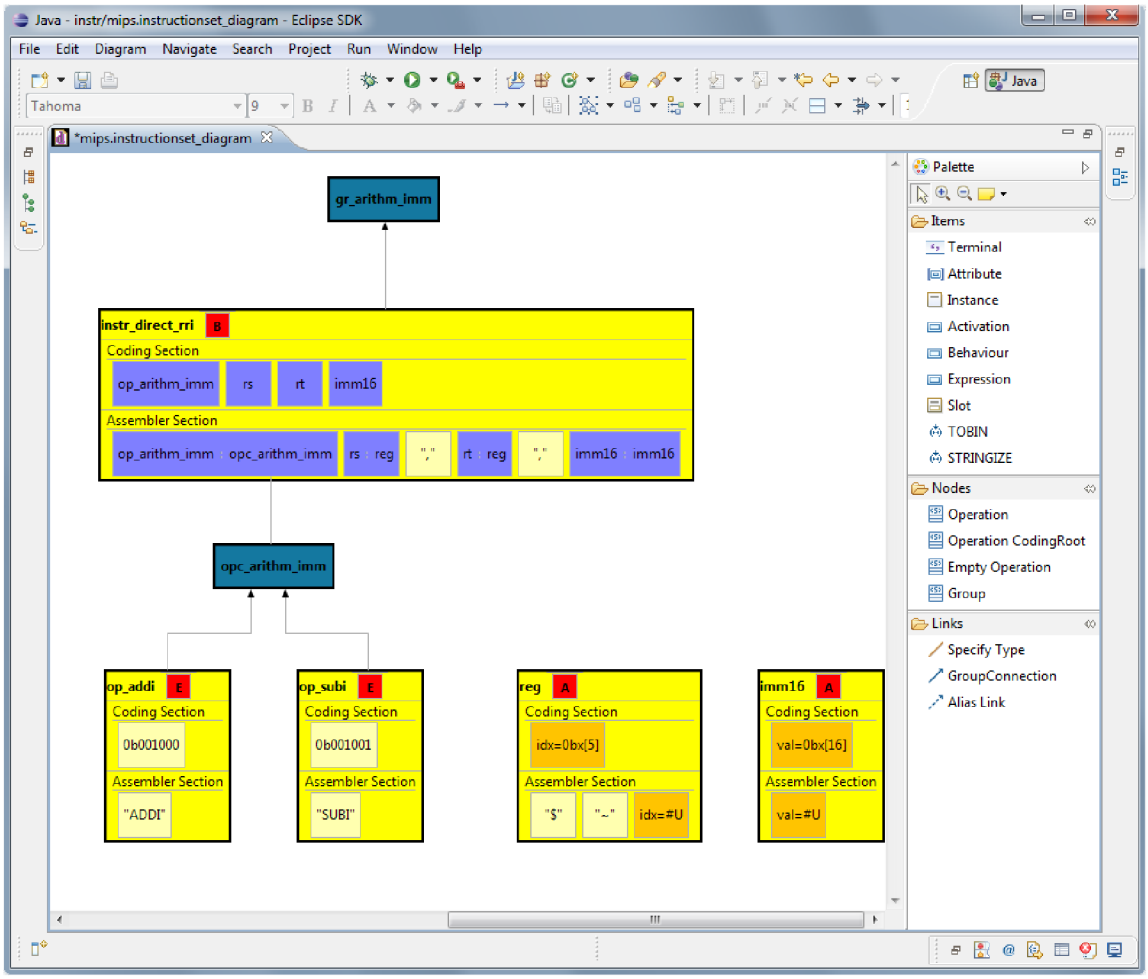
Obrázok 7.1 ilustruje jednoduchý príklad, ktorý demonštruje implementovaný editor štrukturálneho diagramu. Jedná sa o jednoduché jadro mikroprocesoru podobné príkladu 2.3 z kapitoly 2.4. Paleta je kategorizovaná na uzly ako napríklad pamäť, register atd., ktoré návrhár umiestňuje na plátno. Následne pridá porty, vytvorí rozhrania a prepojí jednotlivé komponenty. Tento proces je jednoduchý a názorný.



Obr. 7.1: Ukážka editora štrukturálneho diagramu.

Textová reprezentácia štruktuálneho diagramu z príkladu 7.1 je uložená vo formáte XMI. Príloha C ilustruje jej presnú podobu. Transformácia textovej reprezentácie na zdrojové súbory jazyka ISAC nebola predmetom tejto práce, ale je plánovaná na najbližšie obdobie. Proces transformácie na zdrojové prvky jazyka ISAC vysvetľuje kapitola 2.4.

Druhým implementovaným diagramom je diagram inštrukčnej sady. Ukážka editora diagramu inštrukčnej sady je na obrázku 7.2. Obrázok ilustruje príklad vytvorenia niekoľkých jednoduchých inštrukcií. Názorne je ukázané akým spôsob sú vizualizované prvky *Instance*, *Terminal*, *Attribute* (farebne odlíšené v jednotlivých častiach operácie) a väzby medzi jednotlivými operáciami a skupinami. Podobne ako pri štrukturálnom diagrame je textová reprezentácia uložená vo formáte XMI. Jej obsah, ktorý odpovedá príkladu 7.2, je v prílohe D. Odpovedajúcu transformáciu na zdrojové súbory jazyka ISAC vysvetľuje kapitola 2.4.



Obr. 7.2: Ukážka editora diagramu inštrukčnej sady.

Kapitola 8

Zhodnotenie platformy a vývoja

Nasledujúci text hodnotí zvolenú platformu, pomocou ktorej bol implementovaný výsledný nástroj pre návrh čipu. Hlavnými stavebnými prvkami sú Eclipse frameworky: Eclipse Modeling Framework (EMF), Graphical Editing Framework (GEF) a Graphical Modeling Framework (GMF). GMF umožňuje relatívne v krátkom čase vytvoriť grafický editor postavený nad frameworkom EMF a GEF, ktorý je zasadený do kontextu vývojovej platformy Eclipse. Rozsiahlosť jednotlivých frameworkov na jednej strane umožňuje programátorovi využívať dômyselne koncipované aplikačné rozhrania, ktoré mu poskytujú takmer neobmedzené implementačné možnosti navrhnutých systémov. Na druhej strane však stojí veľmi strmá učiaca krivka, ktorá je navyše umocnená malým množstvom odbornej literatúry k frameworku GMF. GMF je pomerne novým frameworkom a do teraz nebola vydaná žiadna kniha, ktorá by popisovala jeho možnosti. Programátor je tak odkázaný iba na sériu doprovodných návodov, diskusné fóra a javadoc dokumentáciu. Opakom je framework EMF, ku ktorému existuje oficiálna literatúra vysvetľujúca celú jeho architektúru. Na základe tejto skutočnosti sa stáva framework EMF ďaleko viac použiteľný oproti frameworku GMF.

Uvedené skutočnosti maly podstatný vplyv na rýchlosť vývoja editora pre štrukturálny diagram a diagram inštrukčnej sady. Implementácia prebiahala v niekoľkých etapách:

- vytvorenie metamodelu Ecore,
- definovanie tvarov jednotlivých prvkov v modeli **.gmfgraph*,
- určenie mapovania medzi doménovými prvkami metamodelu, paletou nástrojov a ich grafickým popisom,
- úprava modelu generátora,
- vygenerovanie základnej sady zásuvných modulov,
- modifikácia a rozšírenie funkčnosti vygenerovaných zásuvných modulov,
- implementácia invariantov a obmedzujúcich podmienok na zaručenie konzistentnosti vytvoreného modelu s využitím validačného frameworku.

Rozsiahlosť a komplexnosť výsledného editora odzrkadľuje aj počet a rozsah zdrojových súborov: vyše 579 súborov s viac ako 121 000 riadkami kódu.

Kapitola 9

Záver

Cieľom tejto práce bolo vytvoriť nástroj pre návrh čipu v UML, ktorý by umožnil zvýšiť produktivitu návrhu počítačovej architektúry. Práca využíva jazyk UMLISAC, ktorý navrhol Ing. Karel Masařík, Ph.D. Jazyk UMLISAC je grafický popisný jazyk založený na jazyku UML vytvorený s ohľadom na popisný jazyk ISAC. Úvodné kapitoly vysvetľujú štrukturálne vlastnosti a vlastnosti správania modelovacieho jazyka UML použiteľné pre návrh počítačovej architektúry. Ukázalo sa, že modelovanie správania jednotlivých komponent pomocou diagramov aktivít a stavových diagramov, tak ako to popisuje jazyk UMLISAC, sú vhodné iba pre jednoduchšie modely CPU. Pri zložitejších modeloch sa stávajú výsledné diagramy neprehľadné, pretože obsahujú veľký počet stavov. Z tohto dôvodu nie je vhodné takéto diagramy používať, pretože pre návrhára neposkytujú dostatočnú mieru abstrakcie a neprinášajú dostatočné zefektívnenie návrhu vstavaného systému.

Na základe týchto skutočností bol navrhnutý nový diagram pre grafický popis inštrukčnej sady. Tento diagram už je možné používať aj pre zložitejšie modely procesorov, pretože poskytuje vyššiu mieru abstrakcie. Vznikli tak dva základné diagramy: štrukturálny diagram a diagram inštrukčnej sady. Štrukturálny diagram využíva koncept diagramu tried z jazyka UML a umožňuje navrhovať zdrojové prvky jazyka ISAC. Diagram inštrukčnej sady je grafickým ekvivalentom jazyka ISAC popisujúci inštrukčnú sadu a správanie.

Výsledkom práce je sada jedenástich zásuvných modulov do platformy Eclipse, ktoré tvoria výsledný editor. Vytvorený editor je plánované nasadiť ako grafický nástroj pre návrh čipu v projekte Lissom. Nesporná výhoda implementovaného riešenia spočíva v tom, že návrhár získa kompletný produkt postavený na platforme Eclipse, ktorý bude používať spoločne a na rovnakej úrovni so súčasnými nástrojmi projektu Lissom. Pre užívateľa teda nevznikne potreba používať nástroje tretích strán a jeho práca získa na komforte.

Veľký dôraz bol kladený na textovú reprezentáciu grafických prvkov diagramu, ktorá tvorí základ pre budúce pokračovanie projektu. Konečné rozhodnutie padlo na formát XMI. Formát XMI prináša niekoľko výhod. Je štandardizovaný asociáciou OMG, poskytuje pomerne jednoduchú väzbu medzi grafickými prvkami a doménovým modelom a hlavne je podporovaný frameworkami, na ktorých stojí celá implementácia.

Na nasledujúce obdobie je plánované vytvoriť generátor zdrojových súborov jazyka ISAC, ktorý z textovej reprezentácie grafického popisu vytvorí odpovedajúci zdrojový súbor v jazyku ISAC. Diagramy sú navrhnuté tak, aby bolo možné medziformát v XMI prostredníctvom priamej transformácie previesť do jazyka ISAC. Ďalším rozšírením do budúcna je prepojenie vytvorených diagramov tak, aby bolo možné umiestňovať prvky definované v štrukturálnom diagrame priamo do diagramu inštrukčnej sady. V neposlednom rade je plánované vytvoriť nové diagramy, ktoré doplnia funkcionalitu vytvoreného nástroja. Jedná

sa o diagram rozloženia a nový diagram popisujúci prvok activation. Vznikne tak komplexný nástroj, ktorý bude poskytovať rovnaké možnosti návrhu architektúry vstavaného systému ako jazyk ISAC, ale na grafickej úrovni.

Literatúra

[1] Object Constraint Language, OMG Available Specification. http://www.omg.org/spec/OCL/2.0/PDF, 2006 [cit. 8.5.2010].

[2] GMF Constraints. http://wiki.eclipse.org/GMF_Constraints, 2010 [cit. 10.2.2010].

[3] GMF GenModel. http://wiki.eclipse.org/GMF_GenModel, 2010 [cit. 20.2.2010].

[4] OCL: Basic Features. http://swt.informatik.uni-mannheim.de/studies/ocl_basics/l7_ocl_basics.pdf, [cit. 14.2.2010].

[5] Developer Guide to Diagram Runtime Framework. http://help.eclipse.org/help33/index.jsp?topic=/org.eclipse.gmf.doc/prog-guide/runtime/Developer%20Guide%20to%20Command%20Framework.html, [cit. 9.5.2010].

[6] Beckert, B.: Introduction to OCL. http://www.uni-koblenz.de/ beckert/Lehre/Verification/10OCL.pdf, [cit. 8.5.2010].

[7] D'Anjou, J.; Fairbrother, S.; Kehn, D.; aj.: *The Java Developer's Guide to Eclipse*. Addison-Wesley, druhé vydání, 2005, ISBN 0-321-30502-7.

[8] Gronback, R.; Tikhomirov, A.: Developing a Domain-Specific Modeler with the Eclipse Graphical Modeling Framework (GMF). http://wiki.eclipse.org/images/c/c6/GMF_ECOOP2006.ppt.zip, 2006 [cit. 4.12.2009].

[9] Hruška, T.; Masařík, K.; Zámečníková, E.: *ISAC Manual*. Brno, druhé vydání, 2009.

[10] Martin, G.; Müller, W.: *UML for SOC Design*. Springer, 2005, ISBN 978-0-387-25744-6.

[11] Masařík, K.: *Systém pro souběžný návrh technického a programového vybavení počítačů*. FIT VUT v Brně, 2008, ISBN 978-80-214-3863-7.

[12] Moore, B.; Dean, D.; Gerber, A.; aj.: *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM: International Technical Support Organization, 2004, ISBN 0-738-45316-1.

[13] Selic, B.: What's New in UML™ 2.0? ftp://ftp.software.ibm.com/software/rational/web/whitepapers/intro_uml2.pdf, 2005 [cit. 20.11.2009].

[14] Srna, P.: Moderní editor zdrojového kódu pro zadaný jazyk, Bakalářská práce, Vysoké učení technickém v Brně. 2008.

[15] Steinberg, D.; Budinsky, F.; Paternostro, M.; aj.: *EMF Eclipse Modeling Framework*. Pearson Education, Inc., druhé vydání, 2009, ISBN 978-0-321-33188-5.

[16] Venkatesan, M. D.: *Generation of Diagram editors, taking the Enterprise Application Integration Patterns as Case study*. Diplomová práce, Technische Universität Hamburg-Harburg, 2006.

Dodatok A

Obsah CD

Doprovodný dátový nosič má nasledujúcu štruktúru a obsah:

/doc - technická správa
/src - zdrojové súbory
/examples - príklady diagramov
/deploy - zostavené zásuvné moduly

Dodatok B

Návod na inštaláciu

Inštalácia vyžaduje Eclipse vo verzii 3.5 a vyššej. Vývojový nástroj Eclipse je možné stiahnuť z oficiálnych stránok: http://www.eclipse.org/downloads/. Pri inštalácií zásuvných modulov postupujte podľa nasledujúcich krokov:

1. V Eclipse otvoriť: Help ->Install New Software
2. Vytvoriť novú "Site". Otvorí sa okno, kde je potrebné vyplniť Názov umiestnenia (zvolte ľubovoľný, napr. "ISACDiagrams") a cestu k zostaveným modulom (zvoľte súbor z doprovodného dátového nosiša, ktorý sa nachádza v "/deploy/ISACDiagramsUpdateSite.zip").
3. Operáciu potvrďte tlačidlom OK.
4. Zo zoznamu vyberte "Features", ktoré sa budú inštalovať:
   - InstructionSet_Diagram_Feature
   - Structure_Diagram_Feature
5. Nasledujúce kroky sú už samovysvetľujúce. Postupujte podľa pokynov Eclipse.

Preddefinovaných sprievodcov na vytvorenie jednotlivých diagramov je možné nájsť v menu "File ->New ->Example". Doprovodný dátový nosič obsahuje ilustračné príklady v priečinku "/examples". Príklady naimportujte ako projekty do Eclipse a otvorte súbor s diagramom. Pre štrukturálny diagram má tento súbor príponu **.structure_diagram* a pre diagram inštrukčnej sady zvolte súbor s príponou **.instructionset_diagram*.

Dodatok C

Textová reprezentácia štrukturálneho diagramu

```
<?xml version="1.0" encoding="UTF-8"?>
<Canvas
      xmi:version="2.0"
      xmlns:xmi="http://www.omg.org/XMI"
      xmlns="http://Structure">
  <Interface name="t_bit3"/>
  <Interface name="t_bit2"/>
  <Interface name="t_bit4"/>
  <Interface name="t_ops"/>
  <Interface name="t_bit2"/>
  <FileRegister name="fr" size="4">
    <Port name="adr">
      <RequiredInterface
            source="//@FileRegister.0/@Port.0"
            target="//@Interface.0"/>
    </Port>
    <Port name="val">
      <ProvidedInterface
            target="//@Interface.4"
            source="//@FileRegister.0/@Port.1"/>
      <RequiredInterface
            source="//@FileRegister.0/@Port.1"
            target="//@Interface.1"/>
    </Port>
  </FileRegister>
  <FU name="alu">
    <Port name="a">
      <RequiredInterface
            source="//@FU.0/@Port.0"
            target="//@Interface.4"/>
    </Port>
    <Port name="b">
```

```
      <ProvidedInterface
            target="//@Interface.4"
            source="//@FU.0/@Port.1" />
    </Port>
    <Port name="imm">
      <RequiredInterface
            source="//@FU.0/@Port.2"
            target="//@Interface.2" />
    </Port>
    <Port name="opcode">
      <RequiredInterface
            source="//@FU.0/@Port.3"
            target="//@Interface.3" />
    </Port>
  </FU>
  <Decoder name="dec">
    <Port name="ADR">
      <ProvidedInterface
            target="//@Interface.0"
            source="//@Decoder.0/@Port.0" />
    </Port>
    <Port name="VAL">
      <ProvidedInterface
            target="//@Interface.1"
            source="//@Decoder.0/@Port.1" />
    </Port>
    <Port name="IMM">
      <ProvidedInterface
            target="//@Interface.2"
            source="//@Decoder.0/@Port.2" />
    </Port>
    <Port name="OPS">
      <ProvidedInterface
            target="//@Interface.3"
            source="//@Decoder.0/@Port.3" />
    </Port>
  </Decoder>
</Canvas>
```

Dodatok D

Textová reprezentácia diagramu inštrukčnej sady

```
<?xml version="1.0" encoding="UTF-8"?>
<Canvas
      xmi:version="2.0"
      xmlns:xmi="http://www.omg.org/XMI"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns="http://InstructionSet">
  <Group name="opc_arithm_imm"/>
  <Group name="gr_arithm_imm"/>
  <CAOperation name="instr_direct_rri">
    <GroupConnection
          target="gr_arithm_imm"
          source="instr_direct_rri"/>
    <Behaviour filename="instr_direct_rri.behaviour"/>
    <Assember>
      <Item
          xsi:type="Instance"
          id="imm16"
          couplingId="_VmXAsF9HEd-X26PncaqFjg"
          type="imm16"
          typeLabel="imm16"/>
      <Item
          xsi:type="Instance"
          id="rt"
          couplingId="_9MujYF9GEd-X26PncaqFjg"
          type="reg"
          typeLabel="reg"/>
      <Item
          xsi:type="Terminal"
          text=","/>
      <Item
          xsi:type="Instance"
          id="rs"
```

```
            couplingId="_8uH70F9GEd-X26PncaqFjg"
            type="reg"
            typeLabel="reg"/>
        <Item
            xsi:type="Terminal"
            text=","/>
        <Item
            xsi:type="Instance"
            id="op_arithm_imm"
            couplingId="_4zWk0F9FEd-X7pVGIZG3tg"
            type="opc_arithm_imm"
            typeLabel="opc_arithm_imm">
            <TypeLink
                target="opc_arithm_imm"
                source="//@CAOperation.0/@Assember/@Item.5"/>
        </Item>
    </Assember>
    <Coding>
        <Item
            xsi:type="Instance"
            id="op_arithm_imm"
            couplingId="_4zWk0F9FEd-X7pVGIZG3tg"/>
        <Item
            xsi:type="Instance"
            id="rs"
            couplingId="_8uH70F9GEd-X26PncaqFjg"/>
        <Item
            xsi:type="Instance"
            id="rt"
            couplingId="_9MujYF9GEd-X26PncaqFjg"/>
        <Item
            xsi:type="Instance"
            id="imm16"
            couplingId="_VmXAsF9HEd-X26PncaqFjg"/>
    </Coding>
</CAOperation>
<CAOperation name="op_addi">
    <GroupConnection
            target="opc_arithm_imm"
            source="op_addi"/>
    <Expression filename="op_addi.expression"/>
    <Assember>
        <Item xsi:type="Terminal" text="ADDI"/>
    </Assember>
    <Coding>
        <Item xsi:type="Terminal" text="0b001000"/>
    </Coding>
</CAOperation>
```

```
  <CAOperation name="op_subi">
    <GroupConnection
          target="opc_arithm_imm"
          source="op_subi"/>
    <Expression filename="op_addiu.expression"/>
    <Assember>
      <Item xsi:type="Terminal" text="SUBI"/>
    </Assember>
    <Coding>
      <Item xsi:type="Terminal" text="0b001001"/>
    </Coding>
  </CAOperation>
  <CAOperation name="reg">
    <Activation filename="reg.activation"/>
    <Assember>
      <Item xsi:type="Terminal" text="$"/>
      <Item xsi:type="Terminal" text="~"/>
      <Item xsi:type="Attribute" name="idx=#U"/>
    </Assember>
    <Coding>
      <Item xsi:type="Attribute" name="idx=0bx[5]"/>
    </Coding>
  </CAOperation>
  <CAOperation name="imm16">
    <Activation filename="imm16.activation"/>
    <Assember>
      <Item xsi:type="Attribute" name="val=#U"/>
    </Assember>
    <Coding>
      <Item xsi:type="Attribute" name="val=0bx[16]"/>
    </Coding>
  </CAOperation>
</Canvas>
```