OCL KAO PREDUVJET AUTOMATIZIRANOM GENERIRANJU PROGRAMSKOG KODA

OCL AS PREREQUISITE OF AUTOMATED SOURCE CODE GENERATION

Zlatko Stapić, Tihomir Orehovački, Mario Konecki

SAŽETAK

Sve zahtjevniji i složeniji korisnički zahtjevi predstavljaju nove izazove u razvoju aplikacija. Neprestane promjene korisničkih zahtjeva rezultiraju promjenama funkcionalnosti aplikacije što posljedično znači i potrebu za izmjenom ili nadogradnjom programskog koda. U svrhu pojednostavljenja procesa ažuriranja UML modela koji opisuju funkcionalnost aplikacije razvoj softvera je potrebno podići na višu razinu apstrakcije, to jest na višu razinu zrelosti koja znači automatsko generiranje programskog koda iz jasno i nedvosmisleno kreiranih modela. Na posljednjoj, petoj, razini zrelosti uporabe modela pri razvoju aplikacija, osim strukture programskog koda i poslovne logike, u modele je potrebno ugraditi i različita ograničenja. U svrhu rješavanja ovog problema, OMG je u UML specifikaciju ugradio i OCL, formalni jezik za definiranje ograničenja nad objektima. Prepoznavši važnost i potrebu korištenja OCL-a u ovom radu smo predstavili šest razina zrelosti korištenja modela (MMLs) pri razvoju softvera, metodički prikazali uporabu UML-a i OCL-a pri spomenutim razinama zrelosti te prikazali važnost uporabe OCL-a u MDA arhitekturi. Potom smo predstavili osnovne značajke, koncepte i sintaksu OCL-a te u posljednjem poglavlju prikazali različite načine primjene ovog formalnog jezika za definiranje ograničenja nad objektima.

ABSTRACT

In the software development domain, users' requirements are becoming more and more complex and demanding. Constant changes of these requirements result in the necessary functionality modifications and subsequently in the source code transformation and alteration. With the purpose of a source code and a UML models mapping process simplification, the whole software engineering methodology should be raised at next abstraction level and next modelling maturity level. This means automated source code generation out of the simple and unambiguous UML models. At the modelling maturity level five, a models that describe application functionality should include an architectural and structural definitions, a business logic and obligatory some other constraints. In order to solve this problem, the OMG included the OCL as formal object constraint language into the UML specification. After recognizing the need of the OCL usage, in this paper we described six modelling maturity levels and discussed the UML and OCL utilization at each maturity level. The importance of OCL in MDA is also presented before the OCL main characteristics, the concepts and the syntax are provided. Finally, in the last chapter we pointed out a different ways of the OCL application.

1. UVOD

Programiranje i pisanje programskog koda oduzima najviše vremena pri razvoju programskih proizvoda. Upravo zbog toga, sve više i više se priča o potrebi automatiziranja ovog procesa. Ideja automatskog generiranja programskog koda nije nova i proteže se još od sredine 50-tih godina prošlog stoljeća, kada su uvedene prve asemblerske naredbe kao zamjena za strojni jezik [1]. Od tada pa do danas, kada programiramo jezicima treće, četvrte (i pete) generacije, proces prevođenja u asemblerski, a potom i u strojni jezik u potpunosti je automatiziran i nitko ne sumnja u ispravnost spomenutih automatski generiranih prijevoda.

Danas smo suočeni s većim izazovom koji se ogleda u sve kompleksnijim i širim korisničkim zahtjevima koji programske proizvode čine iznimno složenim i kompliciranim. Osim što su spomenuti programski proizvodi postali težak zalogaj projektnim timovima, posljedično, ovi programski proizvodi su i izuzetno teški za nadogradnju i održavanje.

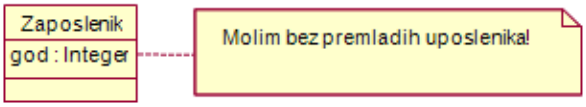
Rješenje koje se čini kao jedini izlaz iz „spomenute krize" nazire se u podizanju procesa modeliranja i specificiranja funkcionalnih i drugih zahtjeva na višu razinu, s preciznim i jasnim modelima koji su dopunjeni detaljnim i jasno definiranim ograničenjima i drugim zahtjevima, kako bi se većina programskog koda ili čak programski kod u cjelini mogli automatski kreirati. Iako je OMG (Object Management Group) organizacija za kreiranje industrijskih (a time i de facto svjetskih) standarda još prije nekoliko godina predstavila prvu verziju UML-a (od engleskog Unified Modelling Language), to jest jezika za modeliranje, sve više se osjetila potreba za specificiranjem dodatnog jezika kojim bi se na jedinstven način opisala ograničenja nad objektima koji se modeliraju.

Veliki korak u definiranju ovakvog jezika kojim bi se jasno i nedvosmisleno specificirala ograničenja je napravila tvrtka IBM (International Business Machines ltd.) koja je još 1995. godine unutar svog odjela za osiguranje predstavila OCL (od engleskih riječi Object Constraint Language) ili jezik za definiranje ograničenja nad objektima. OMG je prihvatio OCL, te je dorađenu specifikaciju istog pružio kao podskup UML 2.0 specifikacije. UML i OCL se i dalje nadograđuju, tako da su posljednje specifikacije ovih jezika za modeliranje objekata i definiranja ograničenja nad njima bile UML 2.1.1 specifikacija te OCL 2.0 specifikacija [2].

2. POTREBA ZA OCL-OM

U pojašnjavanju potrebe za OCL-om poći ćemo od konkretnog problema. Naime, pretpostavimo da UML dijagramom klasa želimo specificirati klasu objekta *zaposlenik*. Osim standardno u UML ugrađenih elemenata dijagrama klasa (koji nam na primjer omogućuju jasno definiranje svojstva i metoda spomenute klase ili povezivanja klasa objekata sa

drugima), potrebni su nam elementi kojima bi smo na nedvosmislen i formalno prihvatljiv način specificirati i poslovno pravilo koje kaže da *zaposlenik ne može biti maloljetna osoba.* Složit ćemo se da UML nema mogućnost definiranja ovog ograničenja osim na način prikazan na slici ispod, a to je pomoću obične bilješke.



Slika 1. Ograničenja kao "bilješke"

Promatrajući gore navedenu specifikaciju jednostavnog ograničenja, doći ćemo do nekoliko zaključaka. Prvo možemo zaključiti da je korišten prirodni jezik za definiranje ograničenja, što znači da je ograničenje svima razumljivo. Ali druga strana medalje i nije tako svijetla. Naime, što u konkretnom primjeru znači riječ „premlad". Da li se to odnosi na zaposlenike s manje od 20 godina, s manje od 18 godina ili pak s manje od 16 godina? Odgovor će vjerojatno dati sam programer koji će funkcionalnost proširiti mogućnošću dojavljivanja pogreške ukoliko se zaposli osoba mlađa od *n* godina. Ova odluka nikako ne bi smjela biti na samom programeru.

Osim spomenutog, možemo zaključiti da prethodna UML specifikacija ne može biti uključena niti pri eventualnom automatskom generiranju programskog koda. Rezultat bi bio klasa koja ima definirana svojstva i metode, ali u metodama unosa ili ažuriranja podataka ne bi bilo provjere starosti zaposlenika, što bi značilo ne ispunjenje korisničkih zahtjeva, ali i mogućnost unosa netočnih i potpuno krivih podataka o zaposlenicima u bazu.

Ipak, prije nego objasnimo osnovne koncepte jezika za definiranje ograničenja, osvrnut ćemo se na potrebu korištenja OCL-a i modeliranja uopće kroz razine zrelosti korištenja modela u razvoju programskih proizvoda, te na arhitekturu vođenu modelima kao podlogu za korištenje UML modela i OCL specifikacija.

3. RAZINE ZRELOSTI U RAZVOJU SOFTVERA

Ovisno u stupnju i načinu korištenja modeliranja kao koncepta pri razvoju programskih proizvoda možemo identificirati šest razina zrelosti (MML - skraćenica od engleskog naziva „Modelling Maturity Levels"). Označavajući stupnjeve zrelosti od MML 0 do MML 5, označavamo stupanj korištenja i kvalitetu modela kojima se, kako na višim tako i na nižim razinama apstrakcije, pokušava predstaviti i opisati modelirani problem ili programski proizvod u cjelini. U spektru od šest spomenutih razina zrelosti polazimo od razine bez ikakvih specifikacija i modela, te podižući kvalitetu i korisnost modela krećemo se prema posljednjoj razini gdje su modeli dovoljno precizni i detaljni da omogućavaju automatsko generiranje programskog koda [1].

- MML 0 – Bez specifikacija
- MML 1 – Samo tekstualni opis
- MML 2 – Tekst s modelima
- MML 3 – Modeli s tekstom
- MML 4 – Precizni modeli
- MML 5 – Samo modeli

U nastavku su pobrojane razine zrelosti te je svaka od njih detaljnije opisana. Tekst koji slijedi je baziran na članku [2].

3.1 Šest razina zrelosti

Bez specifikacija (razina MML 0)

Najnižu razinu zrelosti (MML 0) karakterizira način razvijanja programskih proizvoda bez zapisivanja specifikacija i modela. Obično je na ovoj razini zrelosti riječ o neprofesionalnoj razvojnoj okolini u kojoj osobe koje razvijaju programski proizvod sve ideje drže u glavi. Ukoliko je riječ o profesionalnoj razvojnoj okolini, najčešće osoba razvija programski proizvod za sebe, pri čemu se povodi mišlju „što će mi specifikacije kad ja znam što zapravo želim", pri čemu svaku novu ideju odmah uključuju u programski proizvod koji razvijaju.

Karakteristike ovako nespecificiranih zahtjeva i ideja najjednostavnije se opisuju kroz negativnosti koje sa sobom nose. Tako je za razvoj softvera bez modela i specifikacija karakteristično da korisnici i razvojni tim (a osobito programeri) imaju različite poglede na funkcionalnost softvera. Drugi veliki problem je vezanje samo jedne osobe za softver jer ga samo ta osoba razumije, pri čemu poduzeće ne smije dopustiti da izgubi tog čovjeka. Čak i kad ista osoba pristupi programskom kodu kojeg je razvijala prije nekoliko mjeseci, neizbježan je (obično skupi i spori) proces reinženjeringa, jer je bilo nemoguće zapamtiti sve važne pojedinosti. Slijedeći problem je donošenje odluka od strane programera na „ad hoc" način, a neprestane promjene korisničkih zahtjeva još više otežavaju posao.

Samo tekstualni opis (razina MML 1)

Prvu razinu zrelosti korištenih modela pri razvoju softvera (MML 1) karakteriziraju šture specifikacije napisane prirodnim jezikom u obliku jednog ili više dokumenata. Specifikacije najčešće nisu dovoljno detaljne, nego se u obliku nekoliko rečenica opiše osnovna funkcionalnost proizvoda koji se razvija. Budući da prirodni jezik može biti dvosmislen, javlja se prvi problem. Naime, korisnici koji sada mogu čitati ono što je zapisano ponekad mogu pogrešno protumačiti značenje što dovodi do, u najmanju ruku, zbrke. Osim spomenutog, često se dogodi da programer donosi poslovne odluke, a uz to, nakon nekog vremena broj dokumenata postane tako veliki da ih je nemoguće neprestano držati ažuriranima.

Tekst s modelima (razina MML 2)

Na sljedećoj razini zrelosti (MML 2) uz specifikacije prirodnim jezikom, koristi se i nekoliko modela kojima se opisuje osnovna struktura budućeg sustava. Spomenuti modeli su najčešće dijagrami, a UML (od engleskog „Unified Modelling Language") se koristi kao de facto standardni jezik za modeliranje ovih dijagrama.

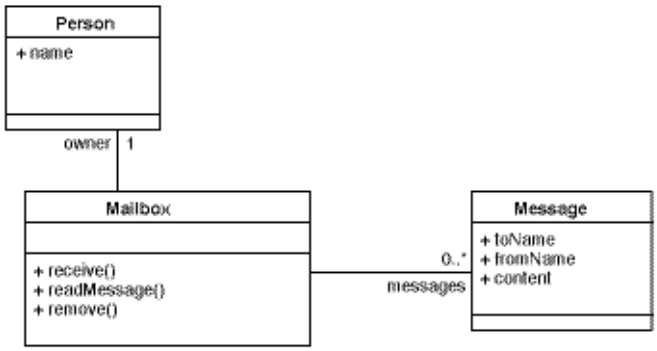
Modeli koji su generirani u razvojnom okruženju druge razine zrelosti su obično jako apstraktni. Tako model pruža određeni uvid u način funkcioniranja budućeg sustava bez obzira što nedostaju konkretni detalji kao što su polja i atributi ili pak operacije i metode koje će biti potrebne (korištene). Ipak, prednost i osnovna razlika u odnosu na prethodnu razinu zrelosti se ogleda u činjenici da je korisnicima uz pomoć priloženih modela tekstualne specifikacije lakše razumjeti. S druge strane, svi nedostaci iz razine zrelosti MML 1 su i dalje prisutni i predstavljaju značajan problem. Naime, specifikacije su još uvijek u osnovi tekstualne, a dijagrami nisu dovoljno precizni, te se ubrzo izgubi nit u ažuriranju istih.

Modeli s tekstom (razina MML 3)

Tek na trećoj razini zrelosti možemo reći da modeli u osnovi opisuju i predstavljaju razvijani programski proizvod. Naime, na ovoj MML 3 razini osnova specifikacije su modeli, a tek ponegdje se koristi prirodni

jezik kojim se opisuju detalji, pozadina i poslovni model ili pak poslovna logika.

Modeli nastali u okruženju treće razine predstavljaju strukturu i opis stvarnog koda kojeg treba izgraditi, te kao takvi ne predstavljaju samo površnu i apstraktnu sliku softvera. Može ih se smatrati dobrom specifikacijom izvornog koda. Primjer koji slijedi prikazuje dijagram klasa aplikacije za slanje internih poruka. Na dijagramu se može pročitati nazive klasa, nazive svojstava i metoda, te način povezivanja istih. Osim spomenutog, možemo vidjeti da su na dijagramu prikazani i drugi detalji, kao što su vidljivost podataka i metoda ili pak broj parametara određene metode. Primjer je preuzet iz članka [2].

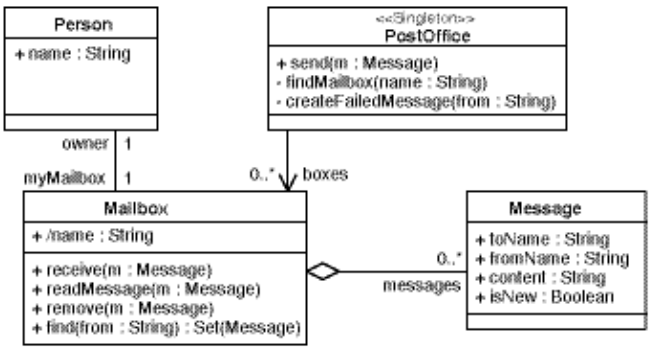


Slika 2. Primjer dijagrama klasa (MML 3)

Na trećoj razini zrelosti, pri razvoju programskog proizvoda, koriste se gotovo svi UML dijagrami i svi koncepti modeliranja. Iako je manje-više sva funkcionalnost softvera specificirana i prikazana dijagramima, programeri još uvijek moraju programski kod programirati „ručno“. Drugi problem je također teško ažuriranje specifikacija sukladno promjenama na programskom kodu koje nastupaju na zahtjev korisnika ili nakon pronalaska programskih logičkih ili semantičkih pogrešaka.

Precizni modeli (razina MML 4)

U ovom razvojnom okruženju, s pretposljednjom razinom zrelosti, za specificiranje softvera koji se razvija koriste se svi UML dijagrami. Iako se prirodni jezik još uvijek može koristiti, njegova upotreba je svedena na „pisanje komentara“ ili opis pozadine i motivacije za nekakvu odluku. Ukoliko se koristi nekoliko različitih dijagrama (a obično ih se mora koristiti), uspostavljaju se veze između njih, što omogućava programerima jednostavno razmatranje posljedica svake promjene koju žele napraviti na modelu. Tako na primjer, jednostavno se može vidjeti koji dijagrami će biti promijenjeni ako se obriše jedna klasa iz dijagrama klasa. Slijedi isti primjer s prethodne razine, ali sada s većom razinom detaljizacije i konkretnijim informacijama [2].



Slika 3. Primjer dijagrama klasa (MML 4)

Modeli na četvrtoj razini zrelosti dovoljno su detaljni i precizni da se može ostvariti direktna veza s kodom. To donosi nekoliko prednosti. Prvo, ažuriranje modela i koda je jednostavno i vrši se po principu preslikavanja. Zatim, programeri više ne donose poslovne odluke, jer se njihov posao napokon svodi samo na dosljednu i kvalitetnu implementaciju specificiranih korisničkih zahtjeva i osmišljene funkcionalnosti (ukoliko se koristi generator programskog koda, velika većina posla se može uraditi automatski). Na posljetku, ove prednosti se ogledaju u činjenici da je omogućen iterativni i inkrementalni pristup razvoju budući da se sve promjene vrše samo na modelima, nakon čega se programski kod jednostavno i brzo ažurira Na ovoj razini zrelosti je moguće primijeniti i bilo koju metodologiju pri razvoju programskih proizvoda.

Samo razvojna okruženja koja su na ovoj razini zrelosti mogu reći da pri razvoju programskih proizvoda koriste MDA (engl. Model Driven Architecture) koncepte.

Vraćajući se na Sliku 3, primijetit ćemo da je dijagram klasa dopunjen određenim ograničenjima (detaljnije modelirane brojnosti i vrste veze i slično), što omogućuje lakše razumijevanje ovisnosti i načina komuniciranja modeliranih klasa objekata. Kao dodatak može se koristiti i OCL naredbe da bi se spomenuti model učinio jasnijim. U nastavku slijedi nekoliko OCL izraza kojima je upotpunjen prikazani dijagram klasa.

```
context Mailbox::readMessage(m:Message)
post: m.isNew = false

context PostOffice::createFailedMessage
(from:String)
post: result.isNew=true and
      result.toName=from and
      result.fromName='PostOffice'and
      result.content='Cannot deliver
     message, addressee unknown'
```

Isječak 1. Ograničenja uz dijagram klasa

Iako ćemo sintaksu i semantiku pojedinih ključnih riječi detaljnije opisati tek u sljedećim poglavljima, iz prikazanog primjera se može vidjeti da je na nedvosmislen način definirano da poruka nakon čitanja mora poprimiti svojstvo „da više nije nova“ (prvi izraz), te da je potrebno korisnika obavijestiti o neuspjeloj isporuci poruke, ukoliko je primatelj nepoznat (drugi izraz).

Samo modeli (razina MML 5)

Peta razina zrelosti korištenja modela pri razvoju softvera je cilj koji još nije u potpunosti dostignut. Ideja je razvoj programskih proizvoda svesti na modeliranje i definiranje različitih ograničenja, a programski kod koji bi bio u potpunosti automatski generiran bi bio skriven i ne vidljiv, kao asembler danas. Programeri bi generiranom kodu u potpunosti vjerovali.

Dva su razloga zašto možemo sa sigurnošću reći da će cilj „samo modeli“ biti postignut. Prvo, danas postoje generatori aplikacija koji su specijalizirani za određenu domenu, što nam ukazuje na činjenicu da je cilj moguće dostići. Drugo, zahtjevi za složenošću programskih proizvoda postaju sve veći te u budućnosti oni neće moći biti ispunjeni bez obzira na ljudske i vremenske resurse koji mogu biti čak i neograničeni. Jedini način da se pobijedi u borbi s „složenošću“ je da se programiranje i razvoj programskih proizvoda podigne na višu razinu apstrakcije. Stoga je potrebno pisanje programskog koda u potpunosti automatizirati, a razvojni tim svoju energiju će moći usmjeriti na modeliranje i specificiranje funkcionalnosti softvera.

3.2 Upotreba UML-a na različitim razinama zrelosti

U prethodnom smo poglavlju opisali šest razina zrelosti korištenja modeliranja pri razvoju programskih proizvoda. Iz opisa svake pojedine razine se može zaključiti o prisutnosti ili neprisutnosti modeliranja kao koncepta, te se tako spektar korištenje modeliranja proteže od potpunog izostajanja do potpunog korištenja istog. Stoga možemo reći da modeli zrelosti označavaju koliko su potpuni, precizni i konzistentni modeli koji se koriste, ali i koliko je potrebno koristiti UML-a i OCL-a u svrhu jasnog i detaljnog specificiranja svih komponenti i zahtjeva.

Slika ispod (Slika 6) prikazuje presjek svih šest razina zrelosti, te odnose modeliranja i programiranja pri razvoju programskih proizvoda.

Budući da je UML de facto standard, koji je prihvaćen za modeliranje programskih proizvoda, zaključit ćemo da je upotreba UML-a na različitim razinama zrelosti jednaka prisutnosti modeliranja. Stoga, promatrajući sliku ispod (Slika 4) možemo zaključiti da je trud koji je potrebno uložiti u programiranje same aplikacije obrnuto proporcionalan trudu koji se uloži u modeliranje i specificiranje zahtijeva. S druge strane, kompleksnost aplikacije koju je moguće pokriti i udovoljiti zahtjevima korisnika direktno su proporcionalne potrebi za korištenjem modela pri razvoju.

Slika 4: Udjeli programiranja i modeliranja

Kao zaključak ovom poglavlju možemo reći da korištenje UML-a kao jedinog jezika za modeliranje nije dovoljno ukoliko želimo kvalitetno specificirati funkcionalnost, te se stoga već na četvrtoj, a osobito na petoj razini zrelosti moramo koristi i druge jezike za specifikaciju dodatnih zahtjeva ili ograničenja. OCL kao jezik za definiranje ograničenja je najčešće korišten.

4. ARHITEKTURA VOĐENA MODELIMA (MDA)

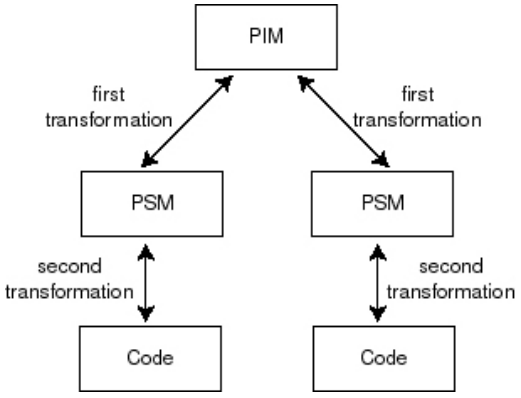
Govoreći o važnosti modela i njihovog korištenja u razvoju softvera, nezaobilazna priča je MDA okvir (engl. MDA framework) koji zaokružuje proces razvoja softvera u cjelinu čije je težište na korištenju modela. Naziv MDA dolazi od engleskog Model Driven Architecture, što možemo prevesti kao *arhitektura pogonjena (ili vođena) modelima.* Sam proces razvoja softvera se unutar MDA svodi na aktivnosti modeliranja.

4.1 Osnovni koncepti MDA

Prema [5], MDA proces je definiran s tri najvažnija koraka. Prvi korak je izgradnja modela s visokim stupnjem apstrakcije neovisnog o budućoj platformi. Drugi korak je transformacija spomenutog modela u jedan ili više modela specifičnih konkretno odabranim platformama, te konačno slijedi treći korak u kojem se svaki model specifičan platformi transformira u kod. Spomenuta tri koraka su prikazana na slici ispod (Slika 5).

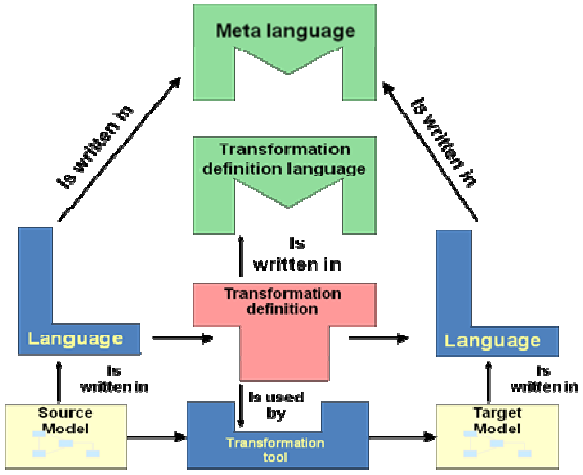
Aktivnosti izgradnje PIM (od engleskog Platform Independent Model) modela su prvi korak koji razvojni tim mora napraviti pri razvoju programskog proizvoda. Prilikom izgradnje PIM modela, uzima se u obzir samo konkretna funkcionalnost budućeg programskog proizvoda. Za opis funkcionalnosti najčešće se koristi standardni jezik za modeliranje (UML), a dodatne specifikacije i ograničenja se definiraju OCL-om. Kreirani model mora biti dovoljno detaljan i konkretan da bi bilo omogućeno nedvosmisleno transformiranje PIM modela u PSM (od engleskog Platform Specific Model) model.

Kako im samo ime govori, PSM modeli moraju biti izgrađeni na temelju specifične platforme (na primjer JAVA, J2EE, .NET, i tako dalje).



Slika 5. Tri MDA koraka [5]

Aktivnost transformacije PIM modela u PSM model je najzahtjevnija aktivnost i ne može biti obavljena bez dodatnih definicija transformacije (engl. transformation definitions). Ova aktivnost se naziva mapiranje (engl. mapping) i najčešće se velikim dijelom obavlja pomoću alata za transformaciju (engl. transformation tools).



Slika 6. Transformacija modela [6]

Warmer (Slika 6) je naglasio potrebu za standardnim jezikom za definiranje transformacija (engl. Transformation Definition Language) [6]. I ovaj jezik mora biti formalan kako bi se mogao koristiti za automatsku transformaciju modela. Najvažniji doprinos MDA okvira je što pruža mogućnost i osnovu za izradu alata (engl. MDA tools) koji bi izvršavali spomenute transformacije iz PIM modela u ciljani PSM.

Posljednji korak je transformacija PSM modela u kod. Ovaj korak ne predstavlja izazov niti problem, budući da se PSM oslanja na točno određenu platformu i budući da postoje jasno definirane definicije transformacije u programski kod.

5. OBJECT CONSTRAINT LANGUAGE

Nakon detaljnog uvoda i obrađenih tema o potrebi jezika za definiranje ograničenja nad objektima, dolazimo do poglavlja u kojem ćemo se upoznati s svim važnim konceptima OCL-a, s njegovom sintaksom te semantikom.

5.1 Osnovne značajke jezika

U pokušajima da specificira jezik kojim bi popunio nedostatke UML-a, IBM odjel za osiguranje je 1995. godine kreirao jezik za definiranje ograničenja, skraćeno OCL. Budući da je UML grafički specifikacijski jezik, pomoću njega možemo opisati samo određene aspekte sustava. Bilješke po marginama nisu prihvatljivo rješenje za specificiranje drugih važnih informacija o funkcionalnosti programskog ili poslovnog sustava.

Formalni jezici su jedino rješenje zbog njihove preciznosti i nedvosmislenosti. Ipak, tradicionalno korišteni formalni jezici za definiranje određenih ograničenja su pokazali nekoliko važnih nedostataka i nije ih se moglo iskoristiti u ovu svrhu. Na primjer, jezik Z. zahtijeva dobro matematičko razumijevanje korisnika, te je njegova primjena svedene na razinu „akademske zajednice“. Osim spomenutog, teško ga je učiti i primijeniti, osobito kad je riječ o velikim sustavima s kompleksnim i brojnim ograničenjima.

Stoga je razvijen novi jezik koji ima nekoliko važnih ciljeva. Prvi uvjeti koji je OCL morao ispunjavati su bili potreba za mogućnošću specificiranja formalnih, preciznih i nedvosmislenih ograničenja. Drugi cilj kojim su se vodili kreatori OCL-a je njegova jednostavnost i mogućnost korištenja od strane velikog broja korisnika. I na posljetku, ništa manje značajan cilj je bila primjenjivost OCL-a u alatima za modeliranje ali i transformiranje modela i generiranje izvršnog koda.

Iako je OCL razvijen kao ekstenzija postojećem jeziku za modeliranje (UML-u), danas je OCL sastavni dio UML specifikacije. Trenutna verzija OCL-a je verzija 2.0 (Final Adoped Specification – formal/06-05-01), a od UML verzije 1.1 (1997) nalazi se unutar UML-a.

Važno je napomenuti da OCL nije programski jezik, nego specifikacijski jezik, što znači da se OCL-om ne može opisati kako nešto napraviti, nego što mora biti zadovoljeno. Za OCL se često kaže da je on čisti jezik izraza (engl. expression language) i tipa (engl. type language), a to znači da su spomenuti OCL izrazi deklarativni te slijedi da njihovo izvršenje nema popratnih efekata. Izvršenje OCL izraza znači evaluaciju izraza pri čemu je rezultat određena vrijednost koja ne mijenja model nad kojim je izraz definiran. Ipak, OCL izrazi mogu biti pisani na način da njihovo izvršenje promijeni stanje sustava, ali ova primjena nije uobičajena. Na kraju, izrazi mogu biti pisani i kao upiti na UML model. Ukoliko se izraz specificira kao upit na UML model, onda je on u potpunosti neovisan o programskom jeziku [7].

Budući da OCL nije programski jezik, njime nije moguće pisati programsku logiku ili kontrolu tijeka podataka unutar aplikacije, ali je ipak moguće logički kontrolirati evaluaciju samog OCL izraza, jer OCL na primjer ima mogućnost selekcije. S OCL-om se naravno ne može pokrenuti proces niti izvršiti bilo kakve izraze osim samih OCL izraza.

Kad kažemo da je OCL jezik tipa, to znači da svaki OCL izraz ima tip, a svaki klasifikator definiran modelom reprezentiran je različitim OCL tipom. Tipovi unutar OCL izraza moraju biti korišteni prema strogo definiranim pravilima. Drugim riječima, unutar OCL izraza se ne može uspoređivati dva različita tipa podataka (na primjer *Integer* i *String)*.

5.2 Sintaksa osnovnih koncepata OCL jezika

Pri razmatranju sintakse jezika, mora se početi od nekoliko ključnih riječi bez kojih ne postoji skoro niti jedna naredba unutar OCL-a. Pogledajmo primjer ispod.

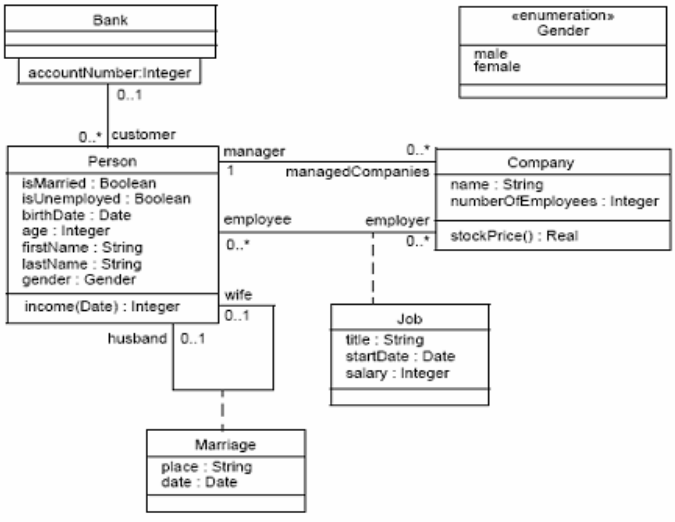
```
context TypeName
inv: 'this is an OCL expression with
     stereotype <<invariant>> in the
     context of TypeName
```

Isječak 2. Opis OCL izraza

Prva ključna riječ je **context** koja nas u svakom izrazu uvodi u kontekst elementa modela na koji se izraz odnosi*.

Sljedeća ključna riječ koju imamo je **inv**, kojom označavamo da će stereotip „nepromjenjivo“ biti korišten (engl. invariant). To znači da izrazom definirani uvjet mora biti ispunjen za sve instance navedenog tipa (na primjer svaku instancu klase *Student),* bez obzira na način, vrijeme i mjesto instanciranja. Taj ćemo stereotip u daljnjem tekstu zvati „nepromjenjivi uvjet“ ili samo „uvjet“. Osim ovog stereotipa, unutar OCL izraza se mogu naći i stereotipovi **pre** i **post** koji označavaju da uvjet iz klauzule ovog tipa mora biti izvršen kao preduvjet, odnosno postuvjet respektivno pri izvršavanju događaja u definiranom kontekstu.

Pretpostavljamo da je sintaksu i semantiku jezika najlakše objasniti kroz primjere. Stoga ćemo sve primjere OCL izraza navesti u kontekstu primjera UML modela klasa prikazanog na Slici 7.



Slika 7. Primjer UML modela [7]

Ako pogledamo sliku iznad, primijetit ćemo da klasa „Poduzeće“ ima svojstvo „broj zaposlenika“. Kako bi smo pokazali primjer nepromjenjivog ograničenja, spomenutom svojstvu definirat ćemo zahtjev da poduzeće ne smije imati manje od 50 zaposlenih.

* *Ključne riječi ćemo označavati podebljano u svrhu lakšeg čitanja OCL izraza.*

```
context Company
inv: self.numberOfEmployees > 50
```

Isječak 3. Primjer „invariant“ izraza

Iz prethodnog primjera možemo primijetiti još i rezerviranu riječ *self*, koja označava da se izraz odnosi na definirani kontekst. Zbog jednostavnosti, ova ključna riječ može biti ispuštena ili zamijenjena drugom vlastito definiranom riječi, kao u primjeru ispod.

```
context c: Company
inv: c.numberOfEmployees > 50
```

Isječak 4. Preimenovani kontekst

Izrazi ne moraju biti definirani u kontekstu cijele klase, nego mogu biti definirani u kontekstu bilo kojeg drugog elementa dijagrama klasa, ali i drugih dijagrama. Na primjeru ispod pokazat ćemo da izraz može biti definiran i u kontekstu metode.

```
context Person::getCurrentSpouse():
   Person
pre:   isMarried = true
body:  mariages->select
       ( m | m.ended = false ).spouse
```

Isječak 5. Ključna riječ "body"

U prethodnom primjeru smo uveli i nekoliko novih koncepata ali i nekoliko novih ključnih riječi. Prvo treba primijetiti da izraz napisan OCL-om može vraćati vrijednost, pri čemu opcionalno možemo definirati i povratni tip. U našem primjeru, povratni tip je *Person* (definiran na kraju klauzule za definiranje konteksta). Vraćanje vrijednosti OCL izraza možemo učiniti na dva načina: pomoću ključnih riječi **result i body**. Osnovna razlika dvaju navedenih ključnih riječi je u tome što se unutar „body“ nalazi cijeli izraz za evaluaciju, a „result“ je s druge strane definiran unutar druge klauzule, te se vrijednost vraća samo ako je uvjet iz klauzule ispunjen.

Potrebno je napomenuti da se „body“ klauzula, ali i uvjeti, preduvjeti i postuvjeti mogu kombinirati i koristiti unutar istog izraza. U primjeru iznad zajedno su korišteni izrazi koji su preduvjet, ali i izraz koji vraća vrijednost.

```
context Person::income(d:Date):Integer
pre parameterOk:  Date > ...
post resultOk:  result = 5000
```

Isječak 6. Parametri u OCL-u

Razmatrajući prethodni isječak koda možemo vidjeti da se rezultat vraća kroz ključnu riječ *result.* Isto tako, u prethodnom primjeru možemo vidjeti da smo preduvjet i postuvjet imenovali. Spomenuta imena u UML modelu mogu biti vrijednosti atributa unutar meta-klase *Ograničenja.*

5.3 Drugi koncepti OCL jezika

Budući da se OCL razvija već nekoliko godina, u ovom radu detaljno objasniti sve koncepte jezika bilo bi nerazumno. Stoga, u ovom poglavlju ćemo se osvrnuti na još nekoliko OCL koncepata, a na druge ćemo samo uputiti čitatelja da postoje.

Ključnom riječi **init** specificiramo inicijalnu vrijednost atributu ili jednog kraja veze.

```
context Job::salary: Integer
init: salary = 900
```

Isječak 7. Inicijaliziranje vrijednosti

Važne ključne riječi su **let** i **def**. Ključnom riječi *let* definiramo varijablu koja se može koristiti unutar više klauzula istog OCL izraza, dok ključnom riječi *def* definiramo varijablu stereotipom <<define>>, što znači da se ista varijabla može koristiti unutar više različitih OCL izraza.

```
context Person inv:
let income: Integer =
    self.job.salary->sum() in
    if isUnemployed then
        income < 100
    else
        income >= 100
    endif


context Person
def: income: Integer =
    self.job.salary->sum()
def: nickname: String = 'Nickname'
def: hasTitle(t: String): Boolean =
    self.job->exists(title = t)
```

Isječak 8. Primjer let i def klauzula

U prethodnom primjeru smo pokazali način uporabe i sintaksu pri korištenju *let* i *def* ključnih riječi. Osim ovoga, prethodni primjer nas uvodi i na OCL mogućnosti kontrole tijeka logike zaključivanja pomoću jednostavnog uvjeta.

Složeni tipovi podataka

Kroz objašnjenja sintakse osnovnih koncepata, indirektno smo spomenuli da su osnovni tipovi podataka podržani OCL-om. Neki od osnovnih tipova podataka su: *String, Integer, Real, Boolean…* S druge strane, OCL podržava i složene tipove podataka, u koje možemo ubrojiti slijedeće:

- Collection
- Set
- Sequence
- Bag

Budući da je OCL jezik tipova, moramo reći da su tipovi unutar ovog jezika složeni po hijerarhiji. Drugim riječima, valjanost OCL izraza, između ostalog ovisi o međusobnom „slaganju“ (engl. conformance) tipova unutar njega. Međusobno slaganje tipova možemo opisati kao mogućnost zamjene jednog tipa drugim, na svako mjestu gdje se dotični tip pojavi. Pravila za slaganje su jednostavna. Prvo, svaki tip se slaže sa svakim svojim pod tipom i drugo, u međusobnom slaganju tipova vrijedi zakon tranzitivnosti.

Hijerarhija gore navedenih složenih tipova podataka također postoji. Tako možemo reći da su *set*, *sequence* i *bag* podtipovi tipa *collection*. Osnovna razlika između navedenih tipova ogleda se u slijedećim činjenicama:

- *set* – skup elemenata kod kojih se svaki element može pojaviti samo jedanput.
- *bag* – skup elemenata kod kojih se svaki element može pojaviti više puta
- sequence – skup elemenata kod kojih se svaki element može pojaviti više puta (kao kod *bag*-a), ali pri čemu su element složeni određenim redoslijedom (na primjer, sortirani).

Na kraju treba spomenuti da unutar OCL-a, ali i unutar svakog tipa podataka postoji još nekoliko različitih operacija i operatora (matematički, logički, binarni, …) koji se mogu primijeniti. Karakteristične operacije ovih složenih tipova podataka su na primjer *ForAll*, *Iterate* i *Exists*.

Ostali tipovi unutar OCL-a

Svi dosada navedeni tipovi podataka (osnovni i složeni) spadaju u skupinu predefiniranih tipova podataka. Osim predefiniranih tipova, unutar OCL-a postoje tipovi koji opisuju različite elemente modela, te na kraju i korisnički definirani tipovi podataka. Slijedi prikaz tipova i podtipova podataka.

1 Predefinirani
   a) *Osnovni tipovi*
      (String, Integer, Real, Boolean i dr.)
   b) *Složeni tipovi*
      (Collections, Sets, Bags i Sequences)
2 Tipovi elemenata modela
   (Classes, Subclasses, Association Classes, Interfaces i dr.)
3 Korisnički definirani tipovi podataka
   (Korisnik, Datum i dr.)

Hijerarhijski promatrano, tipovi podataka najviše razine apstrakcije su *korisnički definirani tipovi podataka*, zatim *tipovi elemenata modela*, a tek onda *predefinirani tipovi* i to po hijerarhiji *složeni* pa *osnovni*. Važno je napomenuti da se unutar određenih skupine također mogu uspostaviti hijerarhijski odnosi (kao na primjer unutar *složenih predefiniranih tipova* podataka, ili unutar *tipova elemenata modela*. Hijerarhijski odnos je moguće uspostaviti i unutar *korisnički definiranih tipova* u koliko to korisnik tako definira.

Osim navedenih tipova, unutar OCL-a se mogu definirati i strukture podataka (n-torke ili engl. tuples). Struktura može imati više imenovanih dijelova, svaki od kojih može biti različitog tipa podataka. Primjer strukture i različitih mogućnosti definiranja iste slijedi ispod.

```
Tuple {name: String = 'John,' age:
Integer = 10} is equivalent to
Tuple {name = 'John,' age = 10} and to
Tuple {age = 10, name = 'John'}
```

Isječak 9. Primjer strukture unutar OCL-a

5.4 Predefinirane operacije nad svim objekata

Nekoliko predefiniranih svojstava i operacija primjenjivo je na sve objekte definirane OCL-om. Ove operacije su pobrojane u nastavku:

- oclIsTypeOf (t : OclType) : Boolean
- oclIsKindOf (t : OclType) : Boolean
- oclInState (s : OclState) : Boolean
- oclIsNew () : Boolean
- oclAsType (t : OclType) : instance of OclType

Tako na primjer, operacija *oclIsTypeOf(t: OclType)* se koristi pri provjeravanju tipa definiranog konteksta te vraća *true* ako su je *self* proslijeđenog tipa *t*.

```
context Person
inv: self.oclIsTypeOf( Person )        -
- is true
inv: self.oclIsTypeOf( Company)       --
is false
```

Isječak 10. Primjer predefinirane operacije

Sve druge operacije imaju sličnu primjenu, pa tako za razliku od prethodne operacije koja provjerava samo direktne tipove objekta operacija *oclIsKindOf(t: OclType)* određuje da li je objekt direktnog tipa (engl. Direct Type) ili jedan od nadređenih tipova objekata (engl. Supertype of an Object).

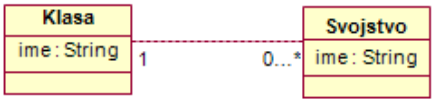
Na kraju ovog poglavlja treba naglasiti da unutar OCL-a postoje različite operacije i svojstva za svaki specifični koncept (objekt, tip objekta i slično), ali zbog složenosti istih i ograničenosti ovog rada njih nećemo detaljno obrađivati. Tako na primjer specifične operacijama nad kolekcijama (engl. Collection Operations) su *Select Operation, Reject Operation*, *Collect Operation, ForAll Operation, Exists Operation* i *Iterate Operation*.

6. PRIMJENA OCL-A

Jezik za definiranje ograničenja (OCL) ima trostruku ulogu u MDA. Tako možemo govoriti o upotrebi OCL u svrhu kreiranja preciznijih modela, u svrhu definiranja jezika za modeliranje i objašnjenja ograničenja za pojedine definicije, te o uporabi OCL pri definiranju transformacijskih definicija pri prevođenju jednog modela u drugi.

Upotreba OCL za definiranje preciznijih modela je već objašnjena u prvom poglavlju rada. Umjesto korištenja tekstualnih bilješki, OCL specifikacijom možemo na nedvosmislen način u UML model ugraditi i određena poslovna ili druga pravila i ograničenja. Osim spomenutog, OCL-om se mogu definirati nepromjenjive (engl. Invariants), te preduvjeti i postuvjeti koje atributi (svojstva) ili metode (funkcije) moraju ispunjavati. Zatim, OCL se koristi za specifikaciju ciljeva na koje se određene poruke i akcije odnose, te za definiranje inicijalnih ili izvedenih vrijednosti samih atributa.

Na sličan način možemo definirati ograničenje prilikom kreiranja metamodela – to jest specifikacija samih modela. Jednostavan primjer može biti potreba da se na formalan način opiše pravilo koje kaže da dva svojstva unutar iste klase ne mogu imati isto ime, te pravilo da dvije klase ne smiju imati isto ime. Na slici ispod je prikazan dio metamodela klasa kojim su jednostavno opisani koncepti klase i svojstva te njihova međusobna veza.



Slika 8. Dio metamodela klasa

Iz priloženog metamodela (Slika 8) možemo vidjeti da je jasno definirano da svaka klasa mora imati ime (koje je znakovni niz) te da svaka klasa može ali i ne mora imati specificirana svojstva. S druge strane, svako svojstvo također mora imati ime (koje je također znakovni niz) i odnosi se na jednu i samo jednu klasu. Ipak, ovaj model mora biti dopunjen OCL specifikacijom kojom će biti opisano pravilo da dva svojstva unutar iste klase ne smiju imati isto ime i pravilo da dvije klase unutar istog modela ne smiju imati isto ime.

```
context Klasa inv:
attributes->isUnique(ime)

context Svojstvo inv:
attributes->isUnique(ime)
```

Isječak 11. OCL kao potpora metamodeliranju

Na kraju treba spomenuti da se uz više detalja OCL može koristiti i pri specificiranju definicija transformacije. OCL-om se u ovom slučaju mogu definirati preduvjeti i/ili

uvjeti koji moraju biti ispunjeni nakon izvršenja transformacije, ali se OCL-om mogu definirati i sama pravila transformacije.

7. ZAKLJUČAK

Jezik za specificiranje ograničenja nad objektima (OCL od engleskog Object Constraint Language) postaje nezaobilazni dio svih specifikacija i modela izrađenih u okruženju s višom razinom zrelosti u uporabi modeliranja pri razvoju programskih proizvoda. Osim ovog, uloga OCL-a je značajna i pri realizaciji težnje podizanja cijelog procesa razvoja programskih proizvoda na višu razinu uz pomoć automatske transformacije modela neovisnih o platformi u modele ovisne o platformi i u konačnici u sam programski kod, što je osnovni koncept arhitekture vođene modeliranjem (MDA – od engleskog Model Driven Architecture). Spoj formalnosti i jednostavnosti omogućuje uporabu OCL-a od strane svih korisnika, ali i u akademske svrhe, čime je prvi puta uklonjena i ova granica između spomenutih dviju domena.

Na kraju možemo konstatirati da će se OCL nastaviti razvijati, budući da MDA vrijeme tek dolazi, a OCL je dio već sada de facto standarda – UML standarda.

Reference:

1 Mohamad, J. A Journey Throught Programming Language Generations. http://www-dse.doc.ic.ac.uk/~nd/surprise_96/journal/vol2/mjbn/article2.html. [učitano 11. 2007.]

2 Object Management Group. Catalog of OMG Modeling and Metadata Specifications. Object Management Group Web site. [učitano 11. 2007.]

3 Kleppe, A., Warmer, J., i Bast, W. MDA Explained - The Model Driven Architecture: Practice and Promise. Addison-Wesley Professional, 2003.

4 Kleppe, A. i Warmer, J. Getting Started with Modeling Maturity Levels. DevX.com. [učitano 12. 2007.] http://www.devx.com/enterprise/Article/26664.

5 Klasse Objecten. OCL Center [učitano 12. 2007.] http://www.klasse.nl/ocl/index.html.

6 Warmer, Jos. The Role of OCL in Model Driven Architecture. 2003.

7 Object Management Group. Object Constraint Language - Final Specification. OMG, 2006.

8 Guttman, M. i Parodi, J. Real-Life MDA - Solving Business Problems with MDA. Morgan Kaufmann Publishers, 2007.

Podaci o autorima:

Zlatko Stapić, dipl. inf.
e-mail: zlatko.stapic@foi.hr

Tihomir Orehovački, dipl. inf.
e-mail: tihomir.orehovacki@foi.hr

Mario Konecki, dipl. inf.
e-mail: mkonecki@foi.hr

Fakultet organizacije i informatike
Pavlinska 2
42000 Varaždin
tel: +385 42 390 800
fax: +385 42 213 413

Zlatko Stapić je od 2006. godine asistent na Katedri za razvoj informacijskih sustava na Fakultetu organizacije i informatike u Varaždinu, te polaznik poslijediplomskog doktorskog studija Informacijske znanosti na istom fakultetu. Kroz studij, te tijekom dosadašnjeg kratkog radnog iskustva dobio je više od deset različitih nagrada i priznanja, uključujući nagradu za najboljeg studenta, Rektorovu nagradu, nagradu „Zlatna Arca“ za inovativnost te nekoliko Dekanovih nagrada. Zlatko je do sada sudjelovao u nekoliko znanstvenih i stručnih projekata, pohađao je seminare i radionice u svrhu proširenja praktičnih znanja, te je objavljivao znanstvene i stručne radove u području razvoja programskih proizvoda, rudarenja podataka i sigurnosti, što su mu ujedno i područja od primarnog interesa. Zlatkov detaljnjiji životopis, s drugim važnim podacima može se pronaći na njegovoj osobnoj web stranici, na http://www.foi.hr/nastavnici/stapic.zlatko/index.html.

Tihomir Orehovački diplomirao je 2005. godine na Fakultetu organizacije i informatike u Varaždinu, smjer Informacijski sustavi. Nakon diplomiranja, kraće je vrijeme radio kao nastavnik u osnovnoj i srednjoj školi. Od 2006. godine radi na Fakultetu organizacije i informatike u zvanju asistenta te sudjeluje u izvođenju nastave na predmetima Programiranje I i Strukture podataka. Praktično informatičko znanje nadopunjavao je nizom seminara i radionica. Područja od posebnog interesa su mu web tehnologije, e-obrazovanje, umjetna inteligencija, upravljanje znanjem, generativno programiranje i simulacijsko modeliranje te je iz tih područja objavio desetak znanstvenih i stručnih radova. Polaznik je poslijediplomskog sveučilišnog doktorskog studija na matičnom fakultetu.

Mario Konecki je diplomirao na Fakultetu organizacije i informatike u Varaždinu 2005 godine. Tijekom studija dva puta je nagrađen kao najbolji student na godini. Za vrijeme studija bio je demonstrator na kolegijima "Programiranje I", 'Operacijski sustavi" i "Sustavi temeljeni na znanju". Tijekom dvije posljednje godine studija počeo je raditi na većim projektima kako za Fakultet organizacije i informatike, tako i za razna poduzeća. Po završetku fakulteta radio je jedan semestar kao asistent na Tekstilno-tehnološkom fakultetu u Varaždinu na kolegiju "Računalstvo". Od 3. mjeseca 2006. godine radi na Fakultetu organizacije i informatike u Varaždinu u zvanju asistenta na kolegiju "Programiranje I".