VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
INFORMATIKOS KATEDRA

Modeliais grįsto programų sistemų kūrimo tyrimas

Model Driven Software Development Research

Magistro baigiamasis darbas

Atliko:	Petras Petkus
Darbo vadovas:	lektorius Donatas Čiukšys
Recenzentas:	asistentas Viktor Kiško

Vilnius – 2009

Turinys

1 Santrauka

Modeliais grįsta sistemų architektūra (MDA) yra „Object Management Group“ (OMG) konsorciumo iniciatyva apibrėžti naują požiūrį į programų sistemų kūrimą remiantis modeliais ir automatizuota jų transformacija į programinį kodą. Siekdama standartizuoti šį požiūrį, OMG patvirtino visą eilę standartų, bet esminiai MDA principai ir praktikos glūdi modeliais grįstame sistemų kūrimo stiliuje, kuris yra fundamentalus programų sistemų inžinerijoje. MDA idėjos, pradžioje sukėlusios didelį entuziazmą IT bendruomenėje, ilgainiui peraugo į skepticizmą ir kai kurie autoriai atvirai pradėjo abejoti modeliais grįsto sistemų kūrimo perspektyva. Šiame darbe analizuojamos praktinio MDA taikymo programų sistemų kūrimo procese aspektai ir galimybės, analizuojami galimi sprendimai ir kliūtys, dėl kurių MDA požiūris gali būti sunkiai pritaikomas. Įvairių autorių įvardijami praktiniai MDA taikymo programų sistemų kūrimo procese sunkumai, didelės tam reikalingos investicijos, pastangos ir resursai, tinkamų instrumentų trūkumas, didelė standartų įvairovė ir sudėtingos technologijos iš dalies lėmė išaugusį skepticizmą MDA paradigmai. MDA apibrėžia naujus sistemų kūrimo principus ir standartais apibrėžia technologijas, kurios pagrindžia šiuos principus. Tai neišvengiamai įtakoja patį programų sistemų kūrimo procesą, kuris turi pasipildyti naujomis veiklomis, praktikomis ir technologijomis, kaip kurti sistemas taikant MDA požiūrį. Šiame kontekste būtinas tam tikras pragmatinis požiūris į MDA ir su juo susijusius OMG patvirtintus standartus. Perimdami geriausias jų idėjas MDA sistemų kūrimo karkasai siekia pritaikyti MDA požiūrį sistemų kūrimo procese integruodami geriausias klasikinių sistemų kūrimo metodų praktikas, modeliavimo praktikas, idėjas ir instrumentus.

Raktiniai žodžiai: modelis, metamodelis, modeliais grįsta sistemų architektūra, procesas, programų sistema, transformacija, platforma, standartas, karkasas, COMET, OMG, RUP, UML, MDA, DSL.

2 Summary

Model Driven Architecture (MDA) is an approach to using models in software development, which states that models and model-based transformations are a key part of effective automated software development. The Object Management Group (OMG) has defined standards for representing MDA models, but the principles and practice of MDA are rooted in model-based styles of development that have been fundamental to software engineering from its earliest days. Unfortunately, early enthusiasm for Model Driven Architecture (MDA) has dissipated to the point

that many people are openly skeptical of the value of any model-driven approach. This paper examines the practical realities of MDA, difficulties and challenges in adopting an MDA approach to software engineering process. While MDA requires additional efforts and high investment to be adopted in software engineering process, it doesn't provide any means or guidelines for this. This paper argues that to be successful, a pragmatic MDA approach must be executed in context of a sound Enterprise Architecture providing an integrated business architecture and governance structure that enables an organization to respond to business requirements quickly and appropriately.

Keywords: model, metamodel, transformation, software, model-driven development, model-driven architecture, process, standard, platform, framework, COMET, OMG, RUP, UML, MDA, DSL.

3 Įvadas

3.1 Tema ir tyrimo objektas

Modeliais grįsta sistemų architektūra (MDA) yra „Object Management Group" (OMG) konsorciumo iniciatyva apibrėžti naują požiūrį į programų sistemų kūrimą remiantis modeliais ir automatizuota jų transformacija į programinį kodą. MDA apibrėžtas naujas požiūris į sistemų kūrimą neišvengiamai įtakoja patį procesą, kuris turi pasipildyti veiklomis ir naujomis praktikomis, tačiau radome tik keletą šaltinių, kuriuose gana abstrakčiai pristatomas MDA taikymas programų sistemų procese. Šiame darbe analizuojamos praktinės MDA taikymo programų sistemų kūrimo procese galimybės ir realijos.

3.2 Darbo tikslas

Pirminis entuziazmas MDA idėjomis ilgainiui peraugo į skepticizmą ir kai kurie autoriai atvirai pradėjo abejoti modeliais grįsto sistemų kūrimo perspektyva. Nepaisant deklaruojamų pranašumų, kuriuos MDA požiūriu grįstas sistemų kūrimas turėtų prieš įprastinį, radome tik keletą šaltinių, kuriuose abstrakčiai pristatomi sėkmingi MDA taikymo programų sistemų (PS) kūrimo procese pavyzdžiai. Atsižvelgdami į tai, šiame darbe siekėme įvertinti modeliais grįsto požiūrio taikymo programų sistemų kūrimo procese aspektus, galimybes, tokio taikymo kliūtis ir jų priežastis, dėl kurių šis požiūris, nepaisant jo deklaruojamų pranašumų, kol kas nėra plačiai paplitęs.

Šio darbo tikslas yra:

> *Atskleisti modeliais grįsto požiūrio taikymo programų sistemų kūrimo procese galimybes.*

3.3 Uždaviniai

Siekdami aukščiau nurodyto tikslo, šiame darbe išskyrėme šiuos uždavinius:

- Pristatyti MDA požiūrio pagrindines idėjas ir sąvokas.
- Pristatyti MDA taikymo programų sistemų (PS) kūrimo procese pavyzdį.
- Remiantis pasirinktu klasikiniu PS kūrimo procesu:
  - įvertinti MDA požiūrio taikymo įtaką šiam procesui apibrėžiant, kaip keičiasi esamos proceso veiklos ir kokiomis naujomis veiklomis pasipildytų procesas;
  - identifikuoti ir įvertinti egzistuojančius MDA standartus, technologijas ir instrumentus, padedančius vykdyti šias veiklas;
  - identifikuoti MDA taikymo PS kūrimo procese kliūtis bei problemines vietas, ir įvertinti jų priežastis.
- Išskirti verslo informacinių sistemų klases, kurių kūrimui MDA grįstas procesas gali būti sėkmingai taikomas, arba toks taikymas šiuo metu būtų sudėtingas ir mažai efektyvus.

3.4 Modeliais grįstas programų sistemų kūrimas

Programų sistemų kūrimo istorija - tai abstrakcijos lygio augimo istorija. Šio augimo pradžia siejama su David Wheeler, kuris kompiuterių eros pradžioje 1947 metais, dirbdamas su EDSAC kompiuteriu, pirmasis pasiūlė kodo fragmentus, vykdančius tas pačias operacijas, sudėti į atskiras bibliotekas ir procedūras, kurias programa gali iškviesti vykdymo metu [Bio07]. Tokiu būdu EDSAC tapo pirmuoju pasaulyje kompiuteriu, kurio vartotojai galėjo kompiliuoti programas naudojantis jau sukurtomis procedūromis nereikalaujant žinių, kaip tos procedūros yra realizuotos programiniame kode. Vėliau gimė tokios aukšto lygio kalbos, kaip „Fortran“, „Cobol“, „C“, kurios leido pernešti programas į kitas techninės įrangos platformas, o programuotojai pradėjo rašyti programas naudodami struktūrinį programavimo stilių, kuris palengvino programavimą, programinio kodo supratimą ir palaikymą [MSU+04]. Šiuo metu plačiai naudojamos objektiškai orientuotos programavimo kalbos „Java“, „C++“, C# ir kt., kurios leidžia struktūruoti duomenis ir sistemos elgesį į klases ir objektus. Su kiekviena nauja kalba abstrakcijos lygis dar labiau išauga ir su kiekvienu aukštesniu abstrakcijos lygiu mes turime instrumentus, kurie automatiškai transfor-

muoja kodą į vis žemesnį abstrakcijos lygį iki pat mašininio kodo. Šiame augime egzistuoja tam tikras dėsningumas: mes formalizuojame savo žinias apie sistemą aukščiausio kokia tik įmanoma lygio kalba. Vėliau mes išmokstame naudotis šia kalba ir nustatome jos taikymo taisykles ir principus. Šios taisyklės ir principai formalizuojami ir taip gimsta aukštesnio lygio programavimo kalba, kuri geba jos kalba parašytą programą transformuoti į žemesnio lygio kalbą, kuriai savo ruožtu jau egzistuoja transliatoriai, transformuojantys programą į dar žemesnį lygį ir taip toliau [MSU+04].

Šiuo metu tolesniu žingsniu abstrakcijos lygio augime laikomas *modeliais grįstas sistemų kūrimas* (*angl.* Model Driven Development), kuomet programų sistemos kuriamos naudojant nuo technologinės platformos (tokių, kaip Java EE[1] ar .NET[2]) nepriklausančius modelius. Šis nepriklausomumas analogiškas nuo aparatūrinės įrangos nepriklausomų sistemų kūrimui, kurį šiuo metu realizuoja tokios kalbos kaip „Java“ ar „C#“, o programos parašytos šiomis kalbomis be papildomų modifikacijų gali būti vykdomos įvairiose techninės įrangos platformose [MSU+04]. Kuriant modeliais grįstas sistemas, galutinis produktas gali būti realizuotas vienu metu daugelyje platformų nekeičiant pačių modelių.

Modeliais grįsta architektūra (*angl.* Model Driven Architecture – MDA) yra „Object Management Group“ (OMG) konsorciumo iniciatyva apibrėžti naują požiūrį į programų sistemų kūrimą remiantis modeliais ir automatizuota jų transformacija į programinį kodą. Šią iniciatyvą OMG pristatė 2001 ir formalizavo 2003 išleistame MDA gide [MDA03]. MDA paradigma – tai standartizuotas požiūris į programų sistemų kūrimą naudojant modelius, kuriuo siekiama iškelti sistemų projektavimą į aukštesnį abstrakcijos lygmenį, kur taptų įmanoma projektuoti sistemas neprisirišant prie konkrečių platformų ir tokiu būdu atskirti sistemos funkcionalumo specifikaciją nuo to, kaip ši sistema naudosis platformos galimybėmis. Toks sistemų projektavimas ir kūrimas nebepriklausytų nuo programinės aplinkos ir platformos, kurioje ši sistema veiks.

Sistemų kūrimas pagal MDA paradigmą apimtų tokius bendriausius etapus [MDA03]:

- sistema projektuojama nepriklausomai nuo platformos, kurioje sistema veiks;
- apibrėžiamos platformos-kandidatės;
- iš kandidačių parenkama konkreti platforma, kurioje numatoma paleisti sistemą;
- vykdoma sistemos projekto transformacija į funkcionuojančią pasirinktoje platformoje sistemą.

Toks MDA paradigmos keliamas tikslas leistų pilnai realizuoti sistemos [MDA03]:

[1] Java Enterprise Edition – kompanijos „Sun“ programinė platforma daugelio lygmenų verslo sistemų kūrimui.
[2] .NET – kompanijos „Microsoft“ programų sistemų kūrimo platforma.

- pernešamumą (*angl.* portability). Taptų įmanoma be didelių sąnaudų pernešti sistemą nuo vienos platformos prie kitos, nes sistemos specifikacija būtų kuriama nepriklausomai nuo platformos.
- įgalintų programų sistemų tarpusavio sąveiką (*angl.* interoperability). Jei atskiri sistemos komponentai transformuojami į skirtingas platformas ir šie komponentai turi tarpusavyje keistis duomenimis, MDA paradigma sprendžia šią problemą generuojant taip vadinamą tiltą tarp šių komponentų. Pavyzdžiui, PIM modelyje klasių diagrama apibrėžtas duomenų modelis gali būti transformuojamas į Java EE platformos EJB komponentą atitinkantį PSM modelį, o kita šio PIM modelio transformacija gali vykti į reliacinės duomenų bazės modelį. Jų tarpusavio ryšys jau yra apibrėžtas transformuojant PIM modelį į PSM. Transformacijų įrankis šį tarpusavio ryšį realizuoja generuodamas tiltą, kurio dėka informacija iš vieno komponento vienoje platformoje gali būti persiunčiama į kitą ir atgal [KWB03; MDA03].
- Pakartotinį modelių panaudojimą (*angl.* reusability). Kuo aukštesnis modelių abstrakcijos lygis, tuo lengviau yra modelius pakartotinai panaudoti, nes jie yra labai artimi dalykinei sričiai. Pavyzdžiui, tokie dalykinės srities objektai, kaip „Bankas“, „Klientas“, „Sąskaita“, „Duomenų bazė“, „Autorizacija“ labai retai keičiasi ir yra universaliai naudojami įvairiuose tai dalykinei sričiai skirtuose programiniuose produktuose. Kuriant modelius vadovaujantis MDA paradigma, tokie modeliai tampa universalūs ir nesunkiai be didesnių modifikacijų panaudojami kituose tos dalykinės srities produktuose [MSU+04].

Modeliais grįstos architektūros paradigma ir ja apibrėžiamas požiūris į sistemų kūrimą sukėlė didelį susidomėjimą IT bendruomenėje. Buvo paskelbta daug publikacijų, straipsnių, pagrindžiančių šį požiūrį, OMG patvirtino visą eilę standartų, susijusių su modeliais ir modeliavimu. Pirmoje šio darbo dalyje pristatysime pagrindines modeliais grįsto programų sistemų kūrimo paradigmos idėjas ir sąvokas. Antroje dalyje analizuojamos šios paradigmos praktinio taikymo programų sistemų kūrimo procese galimybės.

3.4.1 Modelis

Pagal MDA paradigmą, modelis tikslingai aprašo tam tikrą konkrečią ar abstrakčią esybę ar interesų sritį su kuria yra susijęs per analogijas [MSU+04; MDA03]. Modelius daug paprasčiau kurti, nei realų dalyką, ir jais patogu naudotis planuojant problemos sprendimą. Modelių gama apima pradedant pačius paprasčiausius eskizinius modelius iki pilnai detalizuotų ir vykdomų modelių. Svarbi gero sistemos modeliavimo sąlyga yra abstrakcijos ir klasifikacijos naudojimas

[MSU+04]. Abstrakcijos dėka ignoruojama informacija, kuri neaktuali konkrečiame modeliavimo kontekste. Klasifikuojant grupuojama svarbi informacija, pasižyminti bendromis savybėmis [MSU+04]. MDA paradigmos pagrindas yra skirtingų abstrakcijos lygio modelių kūrimas ir jų jungimas kartu. Kai kurie modeliai būtų nepriklausomi nuo platformos, tuo tarpu kiti būtų specifiški konkrečiai platformai. Kiekvienas modelis užrašomas tekstu ir grafiškai naudojant diagramas. Sistemų kūrimas būtų nuo platformos nepriklausančių modelių (*angl.* Platform Independent Models – PIM) apibrėžimas ir jų automatizuotas transformavimas į vieną ar kelis nuo platformos priklausančius modelius (*angl.* Platform-Specific Models – PSM) [CH03], o šie savo ruožtu automatizuotai būtų transformuojami į programinį kodą ir taip būtų gaunama veikianti sistema.

3.4.2 MOF, metamodelis ir platforma

Metamodelis yra modeliavimo kalbos modelis [MSU+04]. Jis apibrėžia modelių šeimos struktūrą, semantiką ir ribojimus. Modelių šeima – tai grupė modelių, kurie aprašyti bendra sintakse ir semantika.

Kiekvienas modelis ar jų šeima turi savo metamodelį. Pavyzdžiui, UML diagramų modeliai yra apibrėžiami UML metamodeliu, kuris nurodo, kokia turi būti UML modelių struktūra, kokius elementus tie modeliai gali turėti ir kokiomis savybėmis tie elementai gali pasižymėti. Metamodelis gali aprašyti ir konkrečios platformos savybes. Savo ruožtu platformos savybės gali būti aprašytos ne vienu metamodeliu.

Platforma apibrėžiama kaip modelių rinkinio vykdymo aplinkos specifikacija [MSU+04]. Java EE platforma, CORBA, .NET, Linux, Solaris OS, Windows yra tokių platformų pavyzdžiai.

MDA paradigma vadovaujasi OMG konsorciumo patvirtintu MOF (*angl.* Meta Object Facility) standartu, kuris apibrėžia, kaip turi būti aprašomi metamodeliai [MOF02]. Taigi, MOF yra meta-metamodelis – modelis, apibrėžiantis metamodelių rašymo sintaksę ir semantiką. MOF apibrėžia daugelį struktūrinių ir elgsenos UML modelių aspektų, tačiau jis neapibrėžia, kaip UML modeliai yra saugomi, kaip jie grafiškai vaizduojami ar modifikuojami. MOF apibrėžia, kaip modeliai gali būti pasiekiami, kaip jais galima keistis. Tam naudojama OMG patvirtinta XML metaduomenų apsikeitimo sąsaja (*angl.* XML Metadata Exchange – XMI). XMI apibrėžia taisykles, kaip galima gauti XML schemą iš suderinamos su MOF modeliavimo kalbos, o taip pat taisykles, kaip modelį užrašyti XML dokumente. MOF ir XMI standartai yra bazinės MDA infrastruktūros kertiniai akmenys, kurie įgalina užtikrinti modelių nepriklausomumą nuo modeliavimo ir transformavimo įrankių [MSU+04].

3.4.3 Modeliavimas

Populiariausia šiuo metu modeliavimo kalba yra OMG standartizuota Universali modeliavimo kalba (*angl.* Universal Modelling Language – UML). Ši kalba naudoja grafines notacijas ir tekstines anotacijas (stereotipus) abstraktaus sistemos modelio kūrimui ir vaizdavimui, ir toks modelis vadinamas UML modeliu [UML08]. UML leidžia apibrėžti, vizualiai pavaizduoti ir dokumentuoti sistemos modelius, jų struktūrą ir ryšius su kitais modeliais [ATL07]. UML vizualinės notacijos naudojamos pavaizduoti tiek nepriklausomus nuo platformos modelius (PIM), tiek specifinius platformai modelius (PSM), o anotacijomis modeliams galima suteikti norimą specifiškumą bei ryšius su kitais modeliais. UML 2.0 standartas apibrėžia 13 diagramų tipų, kuriomis įvairiais aspektais galima apibrėžti sistemos statinę struktūrą, elgesį ir sąveiką. UML metamodelis atitinka MOF specifikaciją ir todėl OMG siūlo naudoti UML modelius kuriant sistemas, grindžiamas MDA paradigma [MDA03].

3.4.4 Modelių tipai

MDA apibrėžia 4 tipų modelius [MDA03]:

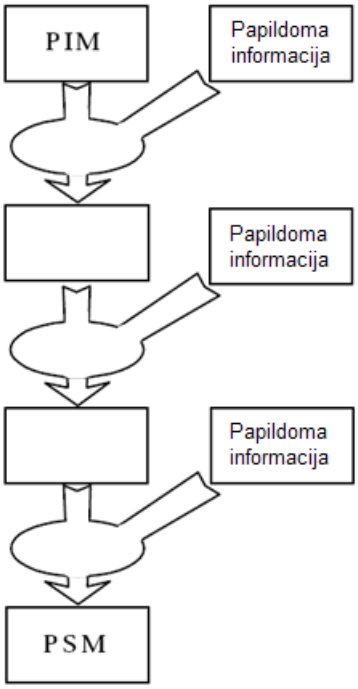
- Nuo algoritmizacijos nepriklausomi modeliai (*angl.* Computation Independent Model – CIM). Tai aukščiausio abstrakcijos lygio modeliai, dar vadinami dalykinės srities modeliais (*angl.* Domain-Specific Models). Tai žodynas, kuriuo naudojasi dalykinės srities praktikas tai sričiai apibūdinti. Laikoma, kad dalykinės srities praktikas, naudodamas CIM modelius, neturi žinių apie modelius, kuriais bus realizuotas sistemos funkcionalumas. CIM modeliai vaidina svarbų vaidmenį apjungiant dalykinės srities ekspertus ir tos srities reikalavimus iš vienos pusės, ir sistemos projektavimo ir konstravimo ekspertus iš kitos. CIM sistemos modelis demonstruoja sistemą toje aplinkoje, kurioje ji veiks ir taip leidžia suprasti, ko tikimasi iš sistemos.
- Nuo platformos nepriklausomi modeliai – PIM (*angl.* Platform-Independent Models). Tai modeliai, su kuriais yra modeliuojamos verslo užduotys, kurias spręs sistema, tačiau pačių sprendimų realizacija nepriklauso nuo techninės platformos. Transformuojant PIM modelius, gaunami vienas ar keli PIM tipo arba PSM tipo modeliai.
- Specifiški konkrečiai platformai modeliai – PSM (*angl.* Platform-Specific Models*)*. Tai tolesnis etapas modelių transformacijų eigoje. Šie modeliai, apimdami PIM modeliuose apibrėžtą sistemos specifikaciją, atsižvelgia į planuojamą naudoti platformą ir pilnai apibrėžia, kaip sistema joje veiks [KWB03; MDA03].

- Platformos modelis. Šis modelis yra skirtingų platformos struktūrinių dalių ir jų teikiamo funkcionalumo specifikacija. Jis apibrėžia platformos elementus ir koncepcijas, naudojamus PSM modelyje apibrėžti sistemos veikimą toje platformoje ir platformos teikiamo funkcionalumo naudojimą.

Programinio kodo MDA nelaiko modeliu, o veikianti sistema generuojama iš PSM modelių vykdant transformaciją iš PSM į platformos vykdomąjį kodą. MDA teigia, kad PSM modeliai, iš kurio generuojamas vykdomas kodas, turi būti pakankamai pilni ir apimti visą informaciją, reikalingą generuoti sistemos realizaciją pasirinktoje platformoje [MDA06].

3.4.5 Transformacija

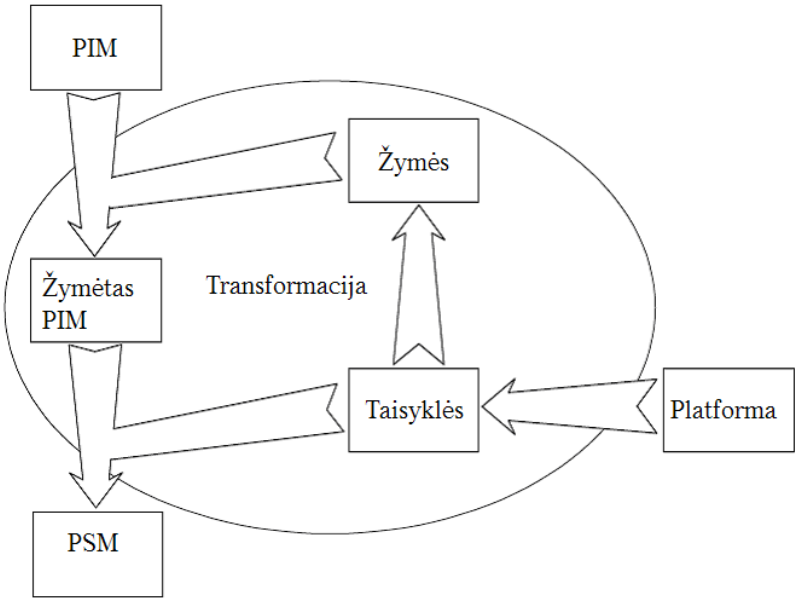
MDA apibrėžia PIM, PSM modelius, ir kaip šie modeliai tarpusavyje siejami. Šis susiejimas atliekamas modelių transformacijų ir modelių transformacijų į programinį kodą dėka. Modelių transformacija – tai procesas, kai vienas ar keli sistemos modeliai paverčiam kitais modeliais toje pačioje sistemoje [KWB03]. Kiekvienos transformacijos dėka gaunami vis detalesni modeliai. MDA siekia apibrėžti modelių transformacijas pradedant nuo PIM modelių iki pat kodo. CIM modeliai yra pernelyg abstraktūs, jie būna sąvokų lygyje ir todėl bet kokios CIM modelių transformacijos į PIM modelius turi būti atliekamos rankomis [KWB03]. Tokia rankinė transformacija nedaug skiriasi nuo tradicinio architektūros projektavimo, atliekamo kuriant sistemą klasikiniu būdu. Tačiau toks projektavimas vadovaujantis MDA paradigma atribojamas nuo platformos [MDA03]. Vėlesnės transformacijos iš PIM į PSM jau turėtų būti atliekamos automatizuotai naudojant specializuotus įrankius [KWB03]. 1 paveiksle pavaizduotas bazinis šablonas, kaip remiantis MDA paradigma turi būti transformuojami modeliai į detalesnius modelius [MDA03].



1 pav. Modelių transformacija [MDA03]

Nuo platformos nepriklausomi modeliai vykdant transformacijas yra jungiami kartu su papildoma informacija ir taip gaunami specifiniai platformos modeliai. Ši papildoma informacija turi suteikti PIM modeliams tą informaciją, kuri padėtų modelį detalizuoti, o vėliau ir specializuoti PSM modelius konkrečiai platformai. Būtina pažymėti, kad 1 pav. tėra iliustracija, demonstruojanti MDA transformacijos principą – kaip turint „mažiau“ (PIM modeliai) gauti „daugiau (detalesni PIM ar PSM modeliai). Kol pasiekiamas transformacijos į PSM etapas, gali tekti atlikti daug PIM transformacijų, tačiau jos irgi bus atliekamos pagal analogišką principą – transformacijos metu modeliai jungiami su papildančia informacija ir taip gaunami vis detalesni PIM modeliai.

Kad transformacija būtų kryptinga, naudojamos transformacijos taisyklės (*angl.* mapping rules) ir žymės (*angl.* marks) [MDA03]. Transformacijos taisyklės nurodo, kaip atliekama modelio transformacija, o žymės yra papildoma informacija, kuri reikalinga modelio transformacijai. Nebūdamos modelio dalimi žymės įtakoja transformaciją, bet ne patį modelį, kurio transformacijai jos naudojamos. Jei žymės priklausytų modeliui, šis taptų specifiškas tam tikrai transformacijai, o tai nėra pageidautina [MSU+04; MDA03]. Būtent modelio transformacijos taisyklės apibrėžia žymių rinkinį tam modeliui. Žymė atitinka tam tikrą PSM modelio sąvoką, kuri yra pritaikoma PIM modelio elementui ir taip nurodant, kaip modelio elementas turi būti transformuojamas.



2 pav. Modelio žymėjimas [MDA03]

2 pav. demonstruoja schemą, kaip susijusios transformavimo taisyklės ir žymės. Modelių transformacijų taisyklės apibrėžia žymių rinkinį, kuriais žymimi modeliai, ir nurodoma, kuria kryptimi jie turi būti transformuojami, kokie reikalavimai keliami įvykdžius transformaciją. Kaip vykdyti transformaciją aprašyta būtent transformacijų taisyklėse. Turėdamas tam tikra transformacijų kalba užrašytas taisykles ir gavęs modelį, kuriame elementai sužymėti žymėmis, atitinkančiomis tą taisyklę, transformacijų įrankis žinos, kaip tą modelį transformuoti.

Žymės taip pat gali nurodyti kokybinius reikalavimus, t.y. ne tik nurodyti transformacijos rezultatą, bet ir nurodyti tam tikrus reikalavimus tam rezultatui. Transformacija tuomet turi būti vykdoma tokiu būdu, kad jos rezultatas atitiktų žyme nurodytus reikalavimus [MDA03].

PIM elementas gali būti žymimas kelis kartus su skirtingų taisyklių žymėmis. Tuomet tai liudija, kad šis elementas dalyvauja keliose transformacijose ir kas kartą bus transformuojamas pagal atitinkamos žymės taisyklę. Gautame modelyje gali atsirasti papildomų elementų arba tie elementai pasižymėti papildomomis savybėmis [MDA03]. Tokiu būdu, nekeičiant pačių modelių, o tik papildant žymes ir transformacijų taisykles, galima gauti papildomą sistemos funkcionalumą.

3.4.6 Šablonai

Transformacijų taisyklės gali apibrėžti transformavimo šablonus (*angl.* transformation templates), kurie apibrėžia tam tikros rūšies transformaciją. Šie šablonai yra tarsi tipiniai projektavimo sprendimai (*angl.* design patterns), tačiau apima žymiai daugiau specializuotos

informacijos, kuri nukreipia transformacijas tam tikra linkme. Šablonai gali apibrėžti transformacijas visam modelio elementų rinkiniui, bet turėti savo žymių rinkinius [MOF02]. Pavyzdžiui, PIM modelio atskiri elementai (klasės ar objektai) gali būti žymimi kaip esybės (*angl.* Entity) taip nurodant, kad būtina naudoti esybių šabloną transformuojant tuos elementus.

3.4.7 Transformacijų kalbos

Transformacijų taisyklės, pagal kurias vienas modelis transformuojamas į kitą, yra apibrėžiamos tam tikra transformacijų kalba. MDA neapibrėžia ir neišskiria kurios nors vienos transformacijų kalbos, tačiau akcentuoja, kad tokia kalba turi užtikrinti pernešamumą (*angl.* portability), t.y. transformacijos galėtų būti vykdomos naudojant skirtingus įrankius [MDA03]. Tam tikslui OMG konsorciumas 2007 metais patvirtino modelių transformacijų QVT[3] kalbos specifikaciją. Šis standartas apibrėžia rinkinį tarpusavy susijusių formalizmų, kuriais išreiškiama modelis į modelį transformacija [QVT07; QVT08].

Apibrėžti ir vykdyti modelių transformacijas gali būti naudojamos įvairios programavimo paradigmos. Transformacijos gali būti aprašytos ir vykdomos taikant procedūrinį, objektiškai orientuotą, funkcinį ar loginį požiūrį ar įvairių požiūrių hibridą [MCG05]. Nepaisant to, egzistuoja esminiai skirtumai tarp transformacijos mechanizmų priklausomai nuo to, ar jie remiasi deklaratyviu ar operaciniu (imperatyviu) požiūriu [MCG05].

- *Deklaratyvus požiūris*. Šis požiūris akcentuoja, *kas* turi būti transformuojama ir *kas* turi būti gauta transformacijos dėka. Akcentuojamas rezultatas, o ne algoritmas, kaip tas rezultatas turi būti gautas.
- *Operacinis (imperatyvus) požiūris*. Šis požiūris akcentuoja, *kaip* turi būti vykdoma transformacija. Operacinė transformacijos specifikacija, kaip ir programavimo kalbose, yra instrukcijų seka.

QVT yra deklaratyvaus požiūrio ir operacinio (imperatyvaus) požiūrio hibridas. *Deklaratyvioji* QVT dalis sudaryta dviejų lygmenų architektūros principu (3 pav.). Šie lygmenys – ryšiai (*angl.* Relations) ir branduolys (*angl.* Core), yra *imperatyvios* dalies – Operacinių transformacijos taisyklių (*angl.* Operational Mappings), – vykdymo semantikos (*angl.* execution semantics) pagrindas [QVT07; QVT08]. Tokiu būdu QVT apibrėžia 3 transformacijų kalbas, kuriomis gali būti transformuojami modeliai:

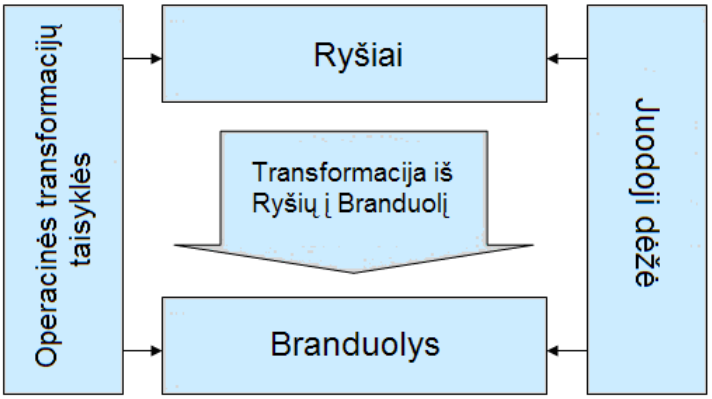
- Ryšiai (*angl.* Relations);
- Branduolys (*angl.* Core);

[3] *angl.* Queries/Views/Transformations

- Operacinės transformacijų taisyklės (*angl.* Operational Mappings).

Ryšiai ir branduolys skirtingais abstrakcijos laipsniais apibrėžia ryšius tarp metamodelių, o transformacija iš Ryšių kalbos į Branduolio kalbą vykdoma naudojant normatyvines taisykles (analogiškai, kaip Java kalba užrašyta programa transliuojama į Java baito kodą). Operacinė transformacijų taisyklių kalba yra imperatyvi kalba, išplečianti tiek Ryšių, tiek Branduolio kalbas ir leidžianti užrašyti transformacijas naudojant imperatyvią sintaksę.

QVT taip pat apibrėžia juodosios dėžės (*angl. Black Box implementation*) mechanizmą, kurio dėka transformacijos gali būti užrašytos kitomis kalbomis. Tai leidžia integruoti transformacijas vykdančias bibliotekas, užrašytas kitomis kalbomis, o taip pat vykdyti sudėtingas modelių transformacijas, kurias yra neįmanoma užrašyti naudojant QVT sintaksę [QVT08].



3 pav. QVT architektūra [QVT08]

QVT transformacijų kalbos pradiniams ir galutiniams modelio transformacijų elementams apibrėžti dažniausiai naudoja OCL išraiškų kalbą. OCL kalba – tai OMG standartizuota modelių išraiškų kalba, kuria galima rašyti užklausas modelių elementams, atitikimo ir validavimo sąlygas ir pan. Transformacijų apibrėžime OCL išraiškomis galima efektyviai apibrėžti pradinius modelio elementus, kurie bus transformuojami, bei galutinius modelio elementus, gaunamus šios transformacijos dėka [KWB03].

Šiuo metu plačiausiai žinoma yra ATL transformacijų kalba. Ši kalba – tai Prancūzijos nacionalinio kompiuterijos mokslo instituto (*pranc.* Institut National de Recherche en Informatique et en Automatique – INRIA) sukurta modelių transformavimo kalba, skirta apibrėžti transformacijas remiantis QVT specifikacija. Šiuo metu ATL yra viena iš svarbiausių Eclipse EMF projekto komponenčių [AMM08]. Programa, parašyta ATL kalba, yra sudaryta iš taisyklių, kurios apibrėžia,

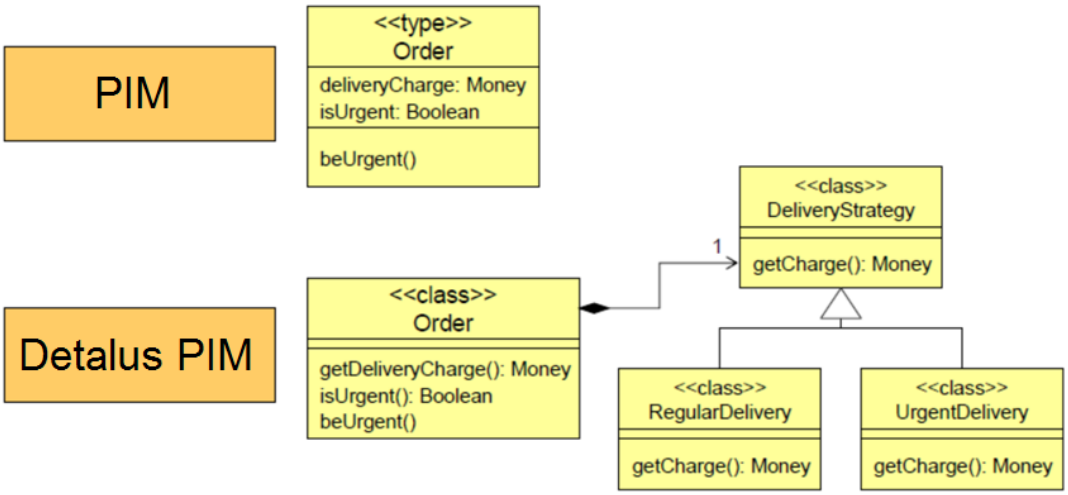
kaip pirminio modelio elementai atitinka ir veda prie reikalingų transformacijų, kurių dėka gaunamas kitas modelis ir jo elementai. Pagrindinis ATL įrankis šiuo metu yra Eclipse pagrindu sukurta integruota kūrimo aplinka (IDE) „ATL Development Tools“. Jame integruotas tekstinis ATL kalbos redaktorius, kuriuo naudojantis ATL kalba galima aprašyti modelių transformacijas [ATL08].

3.4.8 Modelių transformacijų metodai

Transformacijos gali būti vykdomos tiek rankomis, tiek automatiškai, tiek derinant abu šiuos metodus. OMG išskyrė 4 principus (*angl.* approaches), kuriais gali būti vykdomos transformacijos [MDA03]:

1. Rankiniu būdu vykdomos transformacijos. Ankstyvajame projektavimo etape kuriant projektą jau galima naudotis modeliais. Jų transformacija nedaug skiriasi nuo egzistuojančios programų sistemų projektavimo praktikos, tačiau projektuojant pagal MDA išreikštinai atskiriami nuo platformos nepriklausomi modeliai nuo specifinių platformai modelių.
2. Transformacijos naudojant UML profilius. PIM gali būti paruošiami naudojant nuo platformos nepriklausomą UML profilį. Toks modelis gali būti transformuojamas į PSM naudojant jau kitą – specifinį platformai UML profilį. Tokiu atveju PIM modelis būtų sužymėtas žymėmis, kurias suteikia specifinės platformos UML profilis. UML 2 išplečiantys profiliai (*angl.* Profile Extension Mechanism) gali turėti operacijų specifikaciją. Ši specifikacija apibrėžtų transformacijų taisykles ir tokiu būdu jau UML profiliu galima būtų apibrėžti transformacijas [MDA03].
3. Transformacijos naudojant tipinius projektavimo sprendimus (*angl.* design patterns) ir žymes (*angl.* marks). Apibrėžiant transformacijos taisykles gali būti naudojami tipiniai projektavimo sprendimai ir žymės, atitinkantys tam tikrus to sprendimo taikymo aspektus. Tipinio sprendimo žymėmis pažymėti PIM modeliai tuomet transformuojami į modelius, realizuojančius šį tipinį sprendimą. 4 pav. pavaizduota, kaip PIM modelis gali būti detalizuojamas į PIM modelį, realizuojantį strategijos tipinį sprendimą (*angl.* Strategy Design Pattern).
4. Automatinė transformacija. Jau egzistuoja tokios sritys, kur PIM modeliai gali suteikti visą informaciją, reikalingą užbaigtos sistemos realizacijai, ir nebereikia papildomų žymių ar profilių, kad generuoti kodą. Viena iš tokių sričių yra brandus komponentais grįstas sistemų kūrimas, kur tarpinė (*angl.* middleware) programinė įranga gali

pasiūlyti visą rinkinį tam reikalingų įrankių ir paslaugų. Visi reikalingi architektūriniai sprendimai būna padaromi vieną kartą visiems tos srities projektams, skirtiems kurti analogiškas sistemas. Šie sprendimai jau būna realizuoti įrankiuose, sistemos kūrimo procese, šablonuose, programų bibliotekose ir kodo generatoriuose. Tokiame kontekste sistemos projektuotojui turėtų būti įmanoma sukurti išbaigtus PIM modelius, t.y. jie turėtų savyje visą reikalingą informaciją generuoti programinį kodą. Įrankis tiesiogiai interpretuoja modelius ir transformuoja juos į programinį kodą [MDA03].



4 pav. PIM modelio transformacija į tipinį sprendimą realizuojantį detalesnį PIM

4 MDA grįstas sistemų kūrimo procesas

Kiekvienas programų sistemų kūrimo metodas apibrėžia visumą koordinuotų veiklų, kurias atlieka proceso veikėjai (aktoriai) siekdami sukurti apibrėžtą produktą [LE04]. Kuriant metodą siekiama nustatyti gaires, kuriomis būtų galima vadovautis sistemų kūrimo projektuose. Šios gairės apibrėžia proceso veiklas, pagrindinius veikėjus (vaidmenis), darbo produktus ir pan. Sistemų kūrimo metodai yra labai vertinami kuriant stambias verslo sistemas. Tokios sistemos retai kuriamos nuo nulio integruotoje aplinkoje rašant vieną programinio kodo eilutę po kitos, bet kuriamos išplečiant esamus sprendimus juos papildant specifine dalykinės srities logika, jungiant informaciją iš skirtingų šaltinių ir projektuojant komponentus, teikiančius turtingesnes vartotojo sąveikos ir vaizdavimo paslaugas [BC05; NN08]. Tokie metodai taikomi programų sistemų inžinerijos procese, kurio tikslas – pagal nustatytą biudžetą ir terminą sukurti programų sistemą,

tenkinančią užsakovo reikalavimus [Kru03]. Esami programų sistemų kūrimo procesai (pvz., RUP[4], MSF[5], Agile) yra bendro pobūdžio ir gali būti taikomi įvairiems sistemų kūrimo projektams. Šie procesai integruoja geriausias praktikas ir siūlo gaires, kaip efektyviai organizuoti sistemų kūrimo procesą. Organizacijos, priklausomai nuo savo dydžio ir turimų resursų, savo versle paprastai adaptuoja vieną iš procesų.

MDA apibrėžia naujus sistemų kūrimo principus ir standartais apibrėžia technologijas, kurios pagrindžia šiuos principus (sistemų projektavimas ir apibrėžimas nepriklausomais nuo platformos modeliais, modeliavimo standartai, transformacijos ir t.t.). Klasikiniai procesai neakcentuoja sistemų kūrimo modeliais, transformacijomis ir pan., tačiau nustato bei apibrėžia veiklas, kaip kokybiškai ir efektyviai kurti sistemas. MDA apibrėžtas naujas požiūris į sistemų kūrimą neišvengiamai įtakoja patį procesą, kuris turi pasipildyti veiklomis ir naujomis praktikomis, kaip kurti sistemas taikant MDA požiūrį [Bro08]. Sistemų kūrimas modeliais yra tik nedidelė dalis to, kaip modeliai gali padėti kuriant sistemas. MDA grįstas procesas turėtų pasipildyti tokiomis veiklomis, kaip modeliavimo kalbų kūrimas ar esamų kalbų išplėtimas, modelių kūrimas, jų analizė, validavimas, transformacijų, galinčių vykti keliais etapas iki sistemos galutinio realizavimo, apibrėžimas ir pan. [Coo04]. Tačiau MDA neapibrėžia jokių specifinių sistemų kūrimo metodikų, nesuteikia jokių proceso gairių ir nenurodo proceso veiklų, etapų (fazių), vaidmenų ir jų atsakomybių. Be to MDA technologijos nėra aiškiai susietos su proceso veiklomis, kadangi šios technologijos yra kuriamos tokiu būdu, kad būtų galima jas taikyti nepriklausomai nuo organizacijos viduje adaptuoto sistemų kūrimo proceso [GBP+04]. Todėl galima padaryti išvadą, kad MDA nėra PS kūrimo procesas, o yra PS kūrimo proceso *stilius*, kurį galima (bent jau teoriškai) pritaikyti PS kūrimo procesui. Mokslinėje literatūroje yra pristatyta keletą bandymų adaptuoti esamą sistemų kūrimo procesą arba kurti atskirą [GBP+04; NN08; MAS04; COM07]. Vienų jų, COMET, trumpai pristatome tolimesniame skyriuje.

4.1 Programų sistemų kūrimo metodas COMET

MDA neapibrėžia programų sistemų kūrimo proceso, tai tik idėja, kuo grįstas turi būti programų sistemų kūrimo procesas, tarsi rėmai, į kuriuos jis turi būti įspraustas. Rinkoje laisvai prieinamas MDA taikymo metodas COMET (*angl.* Component and Model-based Methodology), – tai į modelius orientuotas ir užduotimis (*angl*. Use Case) grindžiamas programų sistemų kūrimo metodas. Šis metodas suteikia gaires, kaip organizuoti programų sistemų kūrimo procesą ir šį

[4] Rational Unified Process
[5] Microsoft Solutions Framework

procesą grindžiant modeliais kurti sistemas ir jų komponentus [COM07]. COMET procesą apibrėžia kaip struktūruotą veiklų, procedūrų ir modeliavimo praktikų rinkinį, kuris yra naudojamas apibrėžiant, kuriant, testuojant ir diegiant produktus, sistemas ar jų komponentus. COMET siekia detalizuoti MDA ir apibrėžti visą programų sistemų kūrimo procesą pradedant reikalavimais projektuojamai sistemai ir baigiant kodo generavimu. COMET išskiria atskirus programų sistemų kūrimo etapus, kurie modeliuojami atitinkamais to etapo modeliais, ir naudojant to etapo metamodelius šiems modeliams nustato reikalavimus ir apribojimus [COM07].

COMET apibrėžia 4 pagrindinius programų sistemų kūrimo proceso etapų modelius [COM07]:

- *Verslo modelis*. Šis modelis apima tikslus, verslo procesus, žingsnius verslo procese, vaidmenis ir resursus. Modelio apimtis (*angl.* scope) arba dėmesio sritis yra bet kuri dalykinės srities dalis, kuri yra svarbi organizacijai ar kitiems suinteresuotiesiems, ir kuri įtakoja galutinio produkto elgesį ir charakteristikas.
- *Reikalavimų modelis*. Reikalavimų modelis nustato reikalavimus sistemai. Jie apima funkcinius, nefunkcinius reikalavimus ir apribojimus (*angl.* constraints). Nefunkciniai reikalavimai yra teiginiai, susiję su sistemos našumu, prieinamumu (*angl.* availability), patikimumu ir pan. Reikalavimai, apibrėžiantys apribojimus, susiję su prieinamais resursais, specifiniais vartotojo poreikiais, kompanijos strategija ir pan. Užduotys (*angl.* use cases) ir scenarijų aprašai yra naudojami aprašyti funkcinius reikalavimus. Jie taip pat naudojami ir struktūruojant kai kuriuos nefunkcinius reikalavimus ir apribojimus.
- *Architektūros modelis*. Šis modelis aprašo bendrą sistemos architektūrą ir jos skaidymą į komponentus, apibrėždamas komponentų ir posistemių tarpusavio sąveiką ir elgesį, komponentų struktūrą, sąsajas (interfeisus) ir duomenų apsikeitimo protokolus. Šis modelis aprašo komponentų elgesį dviem aspektais: statiniu (struktūros) ir dinaminiu (elgesio). Struktūros modelis aprašo komponentus, jų priklausomybes ir sąsajas. Dinaminis modelis aprašo komponentų sąveikas ir protokolus. Šiais dviem modeliais siekiama apibrėžti komponentus nurodant, kokias sąsajas jie turi, kokias sąsajas naudoja ir kaip turi būti naudojamos (t.y., koks duomenų apsikeitimo protokolas).
- *Specifinis platformai modelis*. Šis modelis apibrėžia komponentų modelio transformaciją į pasirinktos specifinės platformos programinę realizaciją. COMET metodas remiasi MDA paradigmoje apibrėžtais PIM modelių transformacijų į PSM ir kodą būdais [MDA03], kai PIM modeliai gali būti transformuojami naudojant UML profilius arba žymes.

COMET metodas akcentuoja iteratyvų sistemos, sudarytos iš komponentų, kūrimą, o modelių darbo produktai apibrėžia vieną ar kitą architektūrinį požiūrį į komponentinę sistemą ar jos atskirą komponentą. Patys modeliai ir jų darbo produktai yra sistemos kūrimo proceso rezultatas.

Sistemos komponentų kūrimo procesas taikant COMET metodą apima modelių ir su jais susijusių darbo produktų kūrimą. Išskyrus kai kuriuos verslo ir reikalavimų modelio darbo produktus, visi kiti darbo produktai yra UML diagramų modeliai ir kiekvienas toks darbo produktas aprašomas viena ar keliomis UML diagramomis, kurios atitinka to etapo UML profilį ir metamodelį [COM07].

4.2 COMET metodo analizė

COMET metodas yra vienas iš nedaugelio MDA taikymo praktikoje iliustracijų, kuriuos pavyko rasti. Jis apima procesą nuo pat verslo dalykinės srities modeliavimo iki pat produkto diegimo etapo. COMET metodas apibrėžia veiklas, vaidmenis ir veikėjus, atsakingus už tas veiklas, ir tų veiklų darbo produktus. Visgi šis metodas yra daugiau iliustracija, nei praktinis vadovas, kuriuo galima vadovautis vykdant projektą ar pritaikant MDA požiūrį organizacijoje adaptuotam procesui. Tokią išvadą darome remdamiesi šiais argumentais:

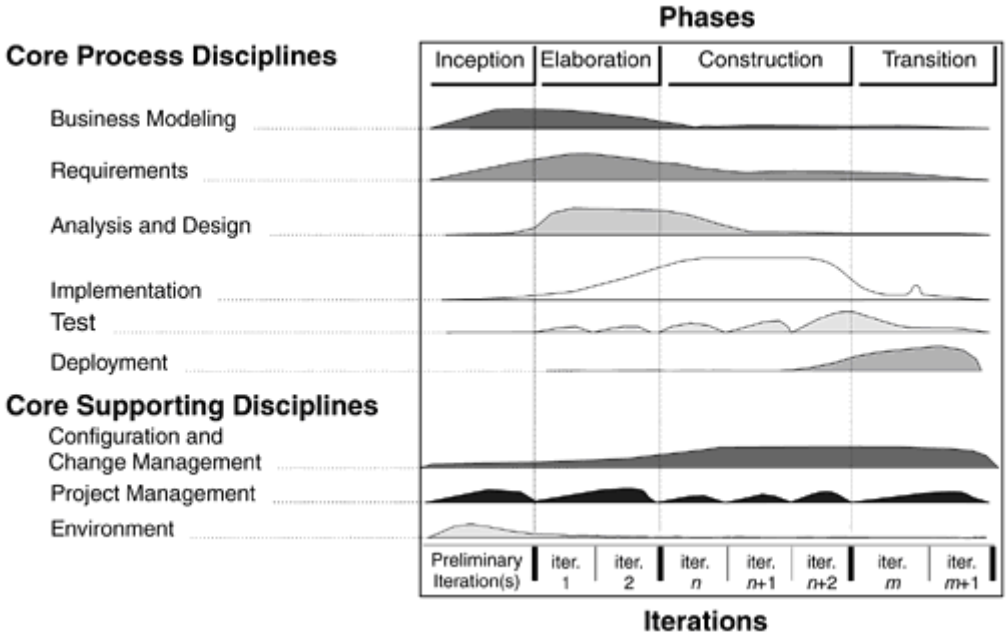
- metodas išskiria veiklas, vaidmenis ir jų darbo produktus - modelius, teigia, kad taikant COMET metodą procesą galima organizuoti iteratyviai, tačiau nepristato kaip tie atskirų veiklų ir proceso iteracijų darbo produktai - modeliai integruojami, kaip užtikrinamas jų korektiškumas, tarpusavio darna ir neprieštaringumas.
- automatizuota transformacija vykdoma tik iš PSM modelių į programinį kodą. Kitos transformacijos vykdomos rankiniu būdu, tačiau jų specifikacija neapibrėžiama ir nepristatoma, nekalbama apie transformacijų testavimą, modelių sinchronizavimą, transformacijų dėka gauto programinio kodo kokybės užtikrinimą ir pan.
- metodas nekalba apie produkto palaikymą, nekalba apie tokiais atvejais vykdomų pakeitimų modelyje paskleidimą į kitus modelius.
- Modeliai ir metamodeliai pristatomi tik iliustruojant kai kurių atskirų veiklų darbo produktus (pvz., Ribų modelis, Naudojimo scenarijų modelis ir pan.). Kitais atvejais išvardijamas tik tų veiklų darbo produktai nedetalizuojant, kaip jie turi būti modeliuojami, kokie turi būti naudojami modeliai ir metamodeliai.
- Architektūrinio projektavimo pristatymas apsiriboja tik UML klasių ir sekų diagramų pavyzdžiais. Nekalbama apie sudėtingesnius atvejus, tokius kaip išskirstytų sistemų arba

lygiagrečių procesų projektavimas ir modeliavimas, ir šių veiklų darbo produktų integravimas.

Įvertinę šiuos COMET metodo trūkumus, šiame darbe iškėlėme tikslą įvertinti MDA požiūrio pritaikymą klasikiniam procesui ir tokio adaptavimo pagrindines kliūtis ir sunkumus. Analizuodami proceso etapus, veiklas ir vaidmenis iš dalies remsimės jų apibrėžimu IBM RUP procese.

4.3 MDA taikymas RUP procese

RUP proceso struktūra apibrėžia dvi požiūrio į sistemų kūrimą dimensijas (perspektyvas): statinę ir dinaminę dimensijas (5 pav.) [Kru03].



5 pav. RUP proceso struktūra [Kru03]

Dinaminė proceso dimensija (horizontali ašis) atspindi dinaminius proceso aspektus. Šie aspektai išreiškiami apibrėžiant proceso etapus, iteracijas ir etapo gaires (*angl.* milestone), žyminčias etapo pabaigą. Statinė dimensija (vertikali ašis) atspindi statinius proceso aspektus: proceso komponentai, veiklos, disciplinos, darbo produktai ir vaidmenys.

Disciplinos yra statinės proceso struktūros dalis. Jos yra savotiški konteineriai, organizuojantys proceso veiklas. Yra 6 techninės (verslo modeliavimas, reikalavimai, architektūrinė analizė ir projektavimas, konstravimas, testavimas ir diegimas) ir 3 papildančios disciplinos, susijusios su sistemų kūrimo projekto valdymu (konfigūracijos ir pakeitimų valdymas, projekto valdymas, aplinka).

IBM RUP techniniai inžinieriai MDA vadina *stiliumi*, kuris gali būti taikomas projektams, ir atspindinčiu naują požiūrį į sistemų kūrimą. RUP jau apima gerąsias sistemų kūrimo praktikas (angl. best practice), tačiau MDA projektai reikalauja naujų praktikų. RUP autorių teigimu, dalį praktikų, kurios tinka MDA projektams, RUP jau siūlo. Alan Brown, vieno iš RUP proceso kūrėjų teigimu, MDA papildytų ar modifikuotų RUP procesą šiais aspektais [BC05]:

- MDA projekte Architekto vaidmuo Įvertinimo (*angl.* Elaboration) fazėje pasipildo naujomis funkcijomis: jis apibrėžia specifines MDA veiklas ir kuriamus artefaktus, kuria transformacijas ir pan. Pagrindiniai MDA artefaktai, už kurių sukūrimą atsakingas architektas yra susiejimų aprašai (*angl.* mapping documents), transformacijos ir UML profiliai.
- MDA taip pat apibrėžia automatizuotą modelių transformaciją. Tai apibrėžia ir tai, kaip MDA ir RUP gali būti naudojami kartu. MDA padėtų automatizuoti veiklas RUP projekte. MDA nekeistų RUP veiklų, bet papildytų jas užduotimis, padėtų automatizuoti tam tikras RUP veiklas. Deja, Alan Brown neapibrėžia, kokios tai užduotys ir kokias RUP veiklas MDA galėtų automatizuoti.
- MDA taikymas įtakotų daugelio proceso vaidmenų funkcijas, tačiau jos būtų ne tiek keičiamos, kiek papildomos naujais veiklos aspektais. Kiekvieno proceso vaidmens funkcijos turėtų būti papildomos tais aspektais, kurie leistų automatizuoti to vaidmens atliekamas veiklas ir MDA technikų naudojimą.
- Žymiai didesnę įtaką procesui turi požiūrio į sistemos kūrimo procesą perspektyvos pokytis. MDA skatina sistemos architektus ir kūrėjus (programuotojus) dirbti aukštesniame abstrakcijos lygyje, nei tai būdavo įprastuose projektuose. Tai yra ypač akivaizdu Konstravimo (*angl.* Construction) fazėje, kuomet MDA programinio kodo kūrimo automatizavimo aspektas reikšmingai keičia realizavimo užduočių atlikimą šioje fazėje. Programuotojai šioje fazėje tęsia darbą su aukštesnio abstrakcijos lygio analizės ir projektavimo modeliais, kurie yra pradiniai modeliai MDA transformacijoms. Tokiu būdu jie daugiau dirba ne su programiniu kodu, bet su verslo proceso sprendimo projektavimu [BC05].

Šie teiginiai bendrais bruožais nusako, kaip MDA keistų RUP procesą. IBM siūlo ir tam tikrą MDA sistemų kūrimo karkasą, kuriuo remiantis galima organizuoti MDA grįstą procesą [Bro08]. Tačiau šis karkasas glaudžiai susietas su RUP naudojamomis technikomis ir komerciniu pagrindu platinamais įrankiais, be to šis karkasas laisvai traktuoja OMG standartus ir daugiau adaptuoja pačią

MDA idėją apie sistemų kūrimą modeliais, neprisirišdamas prie MDA standartų. Pvz., šis karkasas naudoja SysML, UML kalbos atmainą, sistemų modeliavimui ir projektavimui. Be to, prisirišimas prie šio karkaso apsunkina geriausių praktikų pernešimą į kitą MDA sistemų kūrimo kontekstą, kur RUP MDA proceso karkasas nėra ar negali būti naudojamas.

Imdami RUP kaip klasikinio proceso pavyzdį bandysime įvertinti MDA praktinio taikymo problemas ir kliūtis, su kuriomis susidurtų projektų vadovai ir kiti suinteresuoti asmenys, kurdami sistemą pagal MDA paradigmą.

4.3.1 Proceso disciplinos

RUP statinė proceso dimensija apibrėžia proceso disciplinas, kurias labiausiai įtakotų MDA požiūrio taikymas procesui, nes apibrėžia proceso veiklas, vaidmenis ir darbo produktus, kurie MDA grįstame sistemos kūrimo procese yra modeliai. Šių disciplinų principines idėjas apibrėžia ir kiti procesai (MSF, „Agile“ ir pan.), todėl detaliau panagrinėsime kiekvieną šių disciplinų ir kaip jų veiklas įtakotų MDA požiūris.

4.3.1.1 Verslo modeliavimas

Verslo modeliavimo disciplinos tikslai yra šie [Kru03]:

- Perprasti organizacijos, kuriai kuriama sistema, struktūrą ir dinamiką.
- Suvokti sistemos einamąsias problemas, kurias kuriama sistema padėtų spręsti.
- Užtikrinti, kad sistemos galutiniai naudotojai, užsakovai ir kūrėjai turi vienodą supratimą apie organizaciją ir jos tikslus.
- Iš šių tikslų išvesti reikalavimus sistemai, kurių patenkinimas padėtų organizacijai siekti tikslų.

Siekiant šių tikslų verslo modelio disciplina apibrėžia, kaip suformuoti organizacijos verslo viziją ir remiantis šia vizija verslo modelyje apibrėžti verslo procesus, vaidmenis ir atsakomybes. Verslo modelis apima naudojimo scenarijų modelį ir verslo objektų modelį [Kru03].

Modeliavimo technikų naudojimas verslo modeliavime yra panašus į modeliavimo technikų naudojimą sistemų inžinerijoje. Tam naudojama modeliavimo kalba, atitinkanti verslo dalykinės srities kalbą ir terminus, kuri palengvina paties verslo supratimą ir leidžia tarpusavy susikalbėti verslo atstovams ir programų sistemų kūrimo specialistams (sistemų inžinieriams ir architektams). Dalykinės srities kalba leidžia apibrėžti ryšius tarp verslo modelio darbo produktus ir žemesnio abstrakcijos lygio sistemos modelių darbo produktų [Kru03].

MDA gidas teigia, kad verslo modelis ir reikalavimai turi būti modeliuojami nepriklausomais nuo skaičiavimo CIM modeliais [MDA03]. Šie modeliai atspindi sistemą toje verslo aplinkoje, kurioje ji veiks. MDA teigia, kad detalesni PIM ir PSM modeliai turi turėti trasavimo ryšius į CIM modelius, kurie rodytų, iš kokių reikalavimų seka PIM ir PSM modeliai.

Vadovaujantis MDA požiūriu, kuriant sistemą visa projektinė informacija turi būti sudedama į modelius, tačiau MDA nenurodo, kaip tai turi būti daroma. Kyla klausimas, kaip turi būti kuriami verslo modeliai? Verslo modeliuose naudojamos verslo sąvokos, o ryšiai tarp modelių apibrėžiami naudojant verslo taisykles, tačiau UML nesiūlo tam tinkamų verslo modeliavimo technikų ir verslo modelių notacijų, ir OMG nėra apibrėžusi kito standarto, kaip turi būti kuriami verslo modeliai. Egzistuoja 2 sprendimai, kaip modeliuoti verslo modelius [SCG+05]:

- Susikurti specifinį dalykinei sričiai UML profilį
- Susikurti specifinę dalykinei sričiai modeliavimo kalbą (DSL).

Tiek vienas, tiek kitas pasirinkimas turi savų privalumų ir trūkumų. UML profilio naudojimo privalumas:

- UML yra standartizuota modeliavimo kalba. Ją palaiko daugelis modeliavimo įrankių. Net jei tam įrankiui profilis nėra žinomas, modeliavimo įrankis gali dirbti su modeliais ignoruodamas plėtinius (angl. extensions), kuriuos suteikia UML profilis [SCG+05].

UML naudojimo trūkumai:

- UML profiliai suteikia gana ribotą UML pritaikymo dalykinei sričiai laisvę. UML standartas neleidžia sukurti profilius, kuriais būtų galima papildyti UML naujais modeliavimo principais ir išeiti už UML standarto ribų. Pavyzdžiui, neįmanoma sukurti elektrinės schemos modelio ar sukurti jai profilį [SCG+05].
- UML naudojimas reikalauja didelės modeliavimo patirties ir įgūdžių. Modeliai paprastai yra labai sudėtingos struktūros, turinčios daug simbolių ir tekstinių elementų, kuriuos reikia atidžiai analizuoti siekiant suvokti tikrąją jų prasmę. Yra būtinas supratimas, kaip šie elementai veikia, norint sukurti sudėtingą modelį. Iš to seka, kad būtina nemažai investuoti, kad taptum geru modelių kūrėju. Retas dalykinės srities ekspertas geba išreikšti verslo modelį naudojant UML [Coo04; SCG+05].

Šiame kontekste išryškėja DSL naudojimo privalumai [Haa08]:

- DSL kalba turi konkrečią sintaksę ir siūlo sąvokas, atitinkančias tą dalykinę sritį, kuriomis lengvai gali operuoti dalykinės srities ekspertai.
- DSL suteikia galimybę specifiškai optimizuoti transformacijas tam tikros dalykinės srities modeliams.

DSL naudojimo trūkumai [Haa08]:

- DSL kūrimas yra sudėtingas uždavinys, reikalaujantis tiek gerų tos dalykinės srities žinių, tiek kalbų kūrimo patirties.
- Priklausomai nuo taikymo srities apimties bei naudotojų bendruomenės dydžio, DSL kalbos mokymo medžiagos paruošimas, standartizavimas ir palaikymas gali tapti sunkiai įveikiamu ir daug laiko resursų reikalaujančiu uždaviniu.
- Modeliuojant verslą gali prireikti kelių DSL, o tai dar labiau apsunkina dalykinės srities modeliavimo procesą, nes būtina papildomai apibrėžti modeliavimo rezultatų integravimą ir panaudojimą žemesnio lygmens modeliams gauti.

Papildoma modeliavimo UML ir DSL kalbomis analizė pateikiama skyriuje „UML ir DSL naudojimas sistemų apibrėžimui“.

4.3.1.2 Reikalavimų modelis

Reikalavimų specifikavimo etape paprastai keliami šie tikslai [Kru03]:

- Pasiekti susitarimą tarp suinteresuotų asmenų dėl kuriamos sistemos funkcionalumo;
- Suteikti sistemos kūrėjams aiškią perspektyvą, kokie yra keliami reikalavimai sistemai;
- Apibrėžti sistemos apimtį ir ribas;
- Įvertinti resursus, reikalingus sistemos kūrimui;
- Apibrėžti vartotojo sąsają remiantis vartotojo tikslais ir poreikiais.

Klasikiniame procese (pvz., RUP) reikalavimų etape siekiama apibrėžti, kaip sistemos vizija, apibrėžta verslo modeliu, virsta naudojimo scenarijais, kartu su sistemos specifikacija apibrėžiančiais detalius sistemai keliamus reikalavimus, kurių patenkinimas šią viziją realizuotų. Be to, RUP reikalavimų disciplina apibrėžia reikalavimų atributus, padėsiančius valdyti sistemos apimtį ir pokyčius reikalavimuose [Kru03].

Taigi reikalavimų modelis apibrėžia sistemos specifikaciją per vartotojo sąsajos prototipus bei naudojimo scenarijus, iliustruojančius kuriamos sistemos funkcionalumą.

Reikalavimai sistemai skirstomi į funkcinius ir nefunkcinius reikalavimus. Funkciniai reikalavimai apibrėžia pageidaujamą sistemos elgesį. Nefunkciniai reikalavimai apibrėžia kokybines sistemos charakteristikas [Kru03]:

- Naudojamumas – sistemos naudojimo patogumas ir lengvumas vartotojui, sąsajos, dokumentacijos ir mokymo medžiagos darnumas.
- Patikimumas. Šis atributas atspindi sistemos trikių dažnumą, jų tikimybę, sunkumą ir atliekamų veiksmų tikslumą.
- Našumas. Šis reikalavimas nustato funkcinių reikalavimų sąlygas, pvz., transakcijų greitį, dažnumą, reakcijos laiką ir pan.
- Palaikymas. Šis reikalavimas susijęs su kokybiniais reikalavimais sistemai po jos perdavimo užsakovui ir apibrėžia galimybę sistemą testuoti, palaikyti, modifikuoti ir pan.

Tiek klasikiniai procesai, tiek MDA paradigma nurodo, kad reikalavimai sekami iš verslo modelių ir verslo tikslų. Dalis reikalavimo modelio darbo produktų (pvz., naudojimo scenarijai) gali būti UML modeliai. Bet OMG nesuteikia standarto, kaip turi būti užrašomi reikalavimai ir apibrėžti jų ryšiai su verslo modeliais. UML suteikia tik naudotojo scenarijaus modelį, kuriuo galima modeliuoti ir apibrėžti vaidmenis bei atsakomybes. Tačiau patys reikalavimai sistemos funkcionalumui bei nefunkciniai sistemos atributai, tokie kaip našumas, patikimumas, palaikomumas neturi savo UML notacijų. Dėl nepakankamos UML semantikos šie atributai yra užrašomi laisva forma arba naudojant tam tikrus šablonus (pvz., naudojimo scenarijų užrašymo šablonai). Šie šablonai gali apibrėžti reikalavimų užrašymo struktūrą, tačiau tokia struktūra negali būti panaudojama transformacijose [Tho04].

4.3.1.3 Architektūrinis ir detalus projektavimas

Architektūrinis projektavimas RUP procese skaidomas į architektūros analizės ir detalaus projektavimo disciplinas [Kru03]. Klasikiniame programų sistemų kūrimo procese architektūrinis projektavimas jungia reikalavimus su jų realizacija. Jų dėka šie reikalavimai virsta sistemos specifikacija, kuri detaliai apibrėžia, kaip turi būti realizuojami šie reikalavimai. Be to, sistemos architektūrinis projektas apibrėžia, kiek sistema bus naši, saugi, numato galimybes plėsti, testuoti ir t.t. [Kru03].

Sistemos architektūros analizė didesnį dėmesį skiria reikalavimų ir naudojimo scenarijų transformavimui į klasių ir juos grupuojančių paketų rinkinį. Šiame etape siekiama sukurti projektą, tenkinantį sistemos funkcinius reikalavimus. Paprastumo dėlei architektūros analizės metu dar

ignoruojama daugelis nefunkcinių sistemos reikalavimų bei realizavimo ir diegimo aplinkos ribojimų [Kru03].

Architektūros analizės etapo rezultatas yra analizės modelis, kuris atspindi apytikslį sistemos vaizdą. Tokį apytikslį vaizdą RUP vadina „juodraštiniu“ (*angl.* rough sketch of the system). Būdamas dar ganėtinai abstraktus, jis jau atspindi sistemos funkcionalumą.

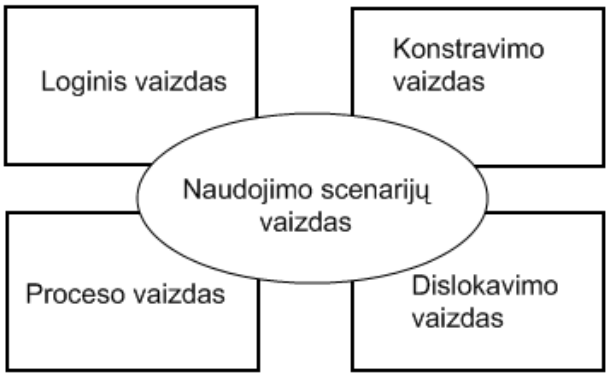
Tuo tarpu sistemos detalaus projektavimo etapas skirtas pritaikyti architektūros analizės modelį prie nefunkcinių sistemos reikalavimų, realizavimo ir diegimo aplinkos ribojimų ir pan. Šis etapas – tai tolimesnis architektūros projekto tobulinimas (*angl.* refinement) ir optimizavimas siekiant, kad būtu atsižvelgiama į visus – tiek funkcinius, tiek nefunkcinius sistemai keltus reikalavimus.

Galutinis architektūrinio projektavimo etapo darbo produktas yra projekto modelis. Šio modelio elementų tarpusavio sąveika (*angl.* collaboration) atspindi sistemos elgesį, apibrėžtą naudojimo scenarijais. RUP procese projekto modelyje sistemos objektai aprašyti klasių pavidalų. Atskleidžiami ryšiai ir sąveika tarp klasių, pagal vykdomas funkcijas klasės grupuojamos į funkcines grupes - paketus (*angl.* package) ir posistemius [Kru03].

Kuriant sistemą pagal MDA paradigmą šiame etape jau būtų kuriami PIM modeliai, kurie abstrakčiame lygmenyje apibrėžia sistemos funkcines savybes. Skirtingai nuo klasikinio proceso, PIM modeliai kuriami atsiribojant nuo platformos, kurioje sistema veiks ir PIM modeliais siekiama apibrėžti visą sistemą ir jos funkcionalumą. Tai įtakotų ir šio proceso etapo pagrindinių vaidmenų vykdomas funkcijas ir etapo darbo produktus. MDA reikalauja ir naujų praktikų, kaip turi būti vykdomas sistemos projektavimas. Kadangi MDA yra aiškus atskyrimas tarp abstraktaus ir konkretaus modelio, Architekto vaidmuo pasipildytų tokiomis funkcijomis, kaip specifinių MDA veiklų apibrėžimas, transformacijų projektavimas, UML profilių, naudojamų modeliavimui, kūrimas, susiejimų (*angl.* mapping) apibrėžimas ir pan. [BC05]. MDA numato automatizuotą modelių transformaciją, todėl proceso vaidmenys turėtų pasipildyti veiklomis, kurios leistų automatizuoti to vaidmens atliekamas veiklas ir MDA technikų naudojimą [BC05]. Tačiau MDA nekalba apie proceso vaidmenis ir jų atliekamas funkcijas, kaip keičiasi klasikinio proceso vaidmenų funkcijos kuriant sistemą pagal MDA paradigmą. Kiekviena organizacija turi pati tai apibrėžti priklausomai nuo jau naudojamo proceso.

4.3.1.3.1 Architektūros vaizdų modeliavimas

Kuriant detalią sistemos architektūrą, jos modelis apibrėžia sistemą iš skirtingų perspektyvų, dar vadinamų vaizdais. Populiariausias sistemos architektūros vaizdų stilius yra 4+1 vaizdų modelis (6 pav.) [Kru95]:



6 pav. 4+1 architektūros vaizdų modelis [Kru95]

Toks modelis leidžia skirtingais aspektais apibrėžti sistemą ir parodo, kaip architektūroje yra atsižvelgiama į vieną ar kelis suinteresuotų asmenų interesus. Pvz., loginis vaizdas apibrėžia sistemos objektinį modelį klasių ir jas grupuojančių paketų pavidalu. Proceso vaizdas apibrėžia sistemoje veikiančius lygiagrečius procesus ir procesų sinchronizavimo aspektus. Konstravimo vaizdas skirtas atsižvelgti į sistemos konstruktorių interesus, apibrėžia programų sistemos modulinę struktūrą, projektavimo ir konstravimo standartus, instrumentus, kurie bus naudojami sistemos konstravimo metu. Dislokavimo vaizdas apibrėžia aplinką, kurioje sistema bus diegiama įtraukiant reikalavimus, kuriuos sistema kelia vykdymo aplinkai [Mas07; Kru95].

MDA taip pat turi vaizdo (požiūrio) sampratą, tačiau ji naudojama kiek kita prasme. Kaip teigia MDA gidas [MDA03], požiūris (*angl.* view) apibrėžiamas per perspektyvą iš tam tikro požiūrio taško (*angl.* viewpoint). Šis požiūrio taškas yra sistemos abstrahavimo technika, naudojanti tam tikrą rinkinį architektūrinių sąvokų ir struktūravimo taisyklių. Taikant šią techniką siekiama atskirti sistemos architektūrą nuo jos realizavimo aspektų ir sutelkti dėmesį ties problemine sistemos vieta [MDA03]. Skirtingai nuo 4+1 vaizdo, MDA žvelgia į sistemos architektūrą per „abstraktus-konkretus modelis“ prizmę (*Vertikalus požiūris*). MDA nesiūlo požiūrio į sistemą (o tuo pačiu ir tokio sistemos modeliavimo), kuriuo galima būti apibrėžti sistemos architektūrą tais aspektais, apie kuriuos kalba 4+1 architektūros modelis (*Horizontalus požiūris*). Šie aspektai ne tik atspindi skirtingą sistemos perspektyvą, bet ir įtakoją patį modeliavimą ir modelių naudojimą. Pavyzdžiui, loginiam sistemos vaizdui kurti gali būti naudojama UML klasių diagrama, o proceso vaizdui gali būti naudojama UML veiklų diagrama. Be to, lygiagrečių procesų vaizdui modeliuoti UML neturi apibrėžtos semantikos [SS07]. Šių vaizdų modelių elementai yra tame pačiame abstrakcijos lygmenyje ir tie patys objektai gali dalyvauti tiek viename, tiek kitame vaizde. Tačiau MDA nekalba apie transformacijas horizontalia kryptimi, kurios dėka galima būtų vykdyti transformacijas iš vieno vaizdo į kitą, integruoti skirtingus modelius, esančius tame pačiame

abstrakcijos lygmenyje. Kadangi sistemos vaizdas pagal MDA vertinamas per abstraktus-konkretus perspektyvą, transformacijos kryptis yra abstrakcijos mažėjimo linkme, t.y. vertikaliai.

Toks MDA požiūris yra kritikuojamas mokslinėje bendruomenėje, nes tai riboja MDA taikymą dideliuose projektuose, kur sistema turi būti projektuojama skirtingais aspektais. Taikant UML yra sudėtinga apibrėžti ryšius tarp skirtingų vaizdų ir net skirtingų modelių ir jų hierarchijos lygmenų [WP03; Haa08], tačiau MDA daugiau kalba apie modelių detalizavimą, bet ne jų integravimą.

4.3.1.3.2 Nefunkcinių sistemos atributų modeliavimas

MDA paradigma yra sunkiai pritaikoma sistemų, reikalaujančių didelio patikimumo, kūrime [Bro08]. Sistemos patikimumas (*angl.* dependability) apima šias kokybines sistemos charakteristikas (atributus) [ALR+04]:

- pasiekiamumas (*angl.* availability) – sistemos pasirengimas veikti korektiškai;
- stabilumas (*angl.* reliability) – sistemos gebėjimas korektiškai veikti be pertraukos;
- saugumas (*angl.* safety) – katastrofinių pasekmių vartotojui ir aplinkai nebuvimas;
- integralumas (*angl.* integrity) – netinkamų sistemos pakeitimų nebuvimas;
- palaikomumas (*angl.* maintainability) – gebėjimas atlikti sistemos modifikacijas ir korekcijas.

Šie patikimos sistemos architektūros atributai yra sunkiai pritaikomi modeliavimo ir metamodeliavimo infrastruktūroje. Būtini geri analitiniai gebėjimai įvertinti šiuos atributus, didelė patirtis kuriant patikimas sistemas bei išsamios MDA standartų, įrankių ir naudojamų technikų žinios. Kadangi nėra priimtų kokybinių charakteristikų modeliavimo standartų, transformacijų įrankiai neatsižvelgia į patikimų sistemų reikalavimus ir dėl to retai naudojami kuriant patikimų sistemų architektūros modelius [Bro08].

Kitas sunkumas taikant MDA požiūrį kuriant architektūros modelius yra susiję su tokiais nefunkciniais atributais, kaip tinklo paslaugų kokybės atributai (*angl.* Quality of Service - QoS). Nors tam tikslui UML QoS profilis gali padėti praturtinant verslo modelius reikalingais QoS atributais, dabartiniai transformacijų ir integravimo įrankiai dar negali dirbti su šiomis modelio charakteristikomis [Bro08].

Be to, automatizuotos architektūros analizės technikos dažnai turi būti pritaikomos specifinei modeliavimo kalbai ir modeliavimo įrankių infrastruktūrai. Jau egzistuojančių architektūros analizės priemonių (pvz., klaidų ir trikių analizė) pritaikymas konkrečiai architektūros modeliavimo kalbai ir įrankiams dažnai reikalauja daug rutininio darbo. Dėl šios priežasties architektūros analizė

dažniau vykdoma neformaliai peržiūrint architektūros modelius, o jų kokybė vis dar vertinama remiantis asmenine patirtimi [Bro08].

4.3.1.3.3 Sistemos apibrėžimas PIM modeliais

MDA sulaukia kritikos ir dėl paties požiūrio, kad visą sistemą galima apibrėžti ir suprojektuoti naudojant tik nepriklausomus nuo platformos PIM modelius:

- MDA akcentuoja sistemų apibrėžimą ir kūrimą nepriklausomais nuo *platformos* modeliais [MDA03]. Dirbantiems su Java, platforma yra aparatūrinė aplinka ir operacinė sistema. Pagal šią sampratą Java parašyta programa gali dirbti bet kurioje platformoje, kurioje yra įdiegta Java virtuali mašina (Java VM). Tačiau MDA kalba apie *programavimo aplinkos* platformą ir programavimą nepriklausomai nuo įprastų programavimo platformų (Java, .NET). Tam tikslui OMG siūlo visą rinkinį standartų (UML, MOF, XMI, CWM, etc.). Tačiau šie standartai yra ne kas kita, kaip tik dar viena platforma, todėl tampa abejotinas MDA tvirtinimas apie sistemų kūrimą neatsižvelgiant į platformą [Fow03].
- OMG nesiūlo jokių tipinių realizacijų ir bibliotekų, įgalinančių sistemų kūrimą PIM modeliais. Todėl yra keblu sukurti pagal MDA net paprasčiausią programą, kurį atspausdina klasikinį „Hello World“ pranešimą, nes OMG standartuose nėra apibrėžtos įvedimo/išvedimo bibliotekos. Tam tikslui reikia naudoti PSM modelius arba kurti savas bibliotekas [Fow03].
- Yra labai sudėtinga praktiškai pritaikyti idėją apie modeliavimą labai abstrakčiame lygmenyje ir po to naudoti šio modeliavimo darbo produktus generuojant skirtingas sprendimo realizacijas. Detaliai projektuojant sistemą būtina atsižvelgti į specifines platformos savybes, kurios įtakoja galutinį architektūrinį projektą ir realizaciją. Todėl, atsiribojimas nuo platformos dažnai gali neatitikti suinteresuotų asmenų interesų ir sistemai keliamų tikslų ir uždavinių [PA09; Zet06]. Tai ypač aktualu sistemose, kurioms keliama daug nefunkcinių (našumo, patikimumo ir kt.) reikalavimų. Tai reiškia, kad funkcionaliai sudėtingų sistemų (pvz., dirbtinio intelekto sistemų) ar sistemų, kurioms keliama daug nefunkcinių reikalavimų, kūrimas taikant MDA požiūrį gali būti neįmanomas. Kita vertus, verslo informacinės sistemos paprastai nekelia tokių didelių kokybinių reikalavimų ir tokios sistemos sudaro didesnę IT verslo dalį.

4.3.1.4 Konstravimas

Šiame etape architektūrinio ir detalaus projektavimo modeliai transformuojami į programinį kodą.

Taikant MDA paradigmą, šiame etape programuotojai dirba daugiau ne su programiniu kodu, o programuodami transformacijas, kurios bus taikomos analizės ir detalaus projektavimo metu gautiems PIM modeliams ir darbo produktams. Tokios transformacijos leistų automatizuotai transformuoti PIM modelius į PSM modelius ir programinį kodą konkrečiai platformai [MDA03]. Kita vertus, sėkmingi MDA taikymo pavyzdžiai, kai automatizuotomis transformacijomis gaunama pilnai veikianti sistema, dažniau demonstruojami sistemose, kurių yra ganėtinai siaura dalykinė sritis [PA09; PM07]. Tokių sistemų pavyzdys yra įdėtinėms sistemos, kurioms keliama žymiai mažiau kokybinių charakteristikų reikalavimų, t.y., paprastai iš jų nereikalaujama saugumo, našumo, pasiekiamumo. Tada praktiškai lieka uždavinys išreikšti sistemos funkcinius reikalavimus per įdėtinės platformos API[6]. Kadangi šis API santykinai mažas (lyginant su Java EE/.NET), tai čia MDA lydi sėkmė.

Taigi, yra ne tik labai sudėtinga apibrėžti sistemą PIM modeliais, bet įvairių autorių teigimu yra labai sudėtinga sukurti ir išsamų PSM modelį, apie kurį kalba MDA, kad iš jo automatizuotos transformacijos dėka būtų galima gauti pilnai veikiantį programinį kodą. Teigiama, kad yra praktiškai neįmanoma pilnai sukurti ir palaikyti semantiškai korektišką vienos ar kitos specifinės platformos (pvz., Java EE ar .NET) PSM modelį, iš kurių automatizuotai generuojamas kodas. Šios platformos turi tūkstančius API sąsajų, daugelis jų menkai aprašytos ir dokumentuotos. Be to, kiekviena nauja platformos versija vėl privers naujai kurti PSM modelius. Todėl programuotojai po transformacijų iš PSM modelių į kodą, turi dar rankiniu būdu papildyti šį kodą reikiama logika [Tho04; Fow04; PA09].

Jei transformacijos sukuria tik programinio kodo griaučius, kuriuos programuotojas turi užpildyti ranka, prarandamas ir vienas iš svarbiausių MDA taikymo privalumų – sistemų apibrėžimas ir kūrimas modeliais, nes toks apibrėžimas PIM ir PSM modeliais yra nepilnas ir nepakankamas automatizuotai generuoti programinį kodą ir veikiančias sistemas. Be to, programuotojai ne tik turi gerai išmanyti vieno ar kito transformacijos įrankio API ir jo kodo generavimo ypatumus, bet ir turi įsisavinti konkrečios platformos API [Tho04]. Imant kaip analogiją Java kalbą, tai tarsi reikalavimas iš Java programuotojo dar žinoti C++ kalbą ir operacinės sistemos platformos API, kad pilnai realizuoti Java parašytą sistemą ir reikalavimuose apibrėžtą jos funkcionalumą.

[6] Sistemų programavimo sąsaja (*angl.* Application Programming Interface)

4.3.1.5 Testavimas

Testavimas skirtas įvertinti produkto kokybę naudojant šias pagrindines praktikas [Kru03]:

- Identifikuoti ir dokumentuoti programinio produkto defektus ir problemas;
- Patikrinti, kad programinis produktas veikia, kaip numatyta architektūriniame projekte;
- Patikrinti, kad iškelti funkciniai ir nefunkciniai reikalavimai yra tinkamai realizuoti.

Skirtingai nuo kitų etapų (verslo modeliavimo, reikalavimų analizės, architektūrinio projektavimo, konstravimo), kurie kreipia dėmesį į darbo produktų pilnumą, darną ir korektiškumą, testavimo etape kreipiamas dėmesys į šių darbo produktų trūkumus, korektiškumo ir darnos nebuvimą.

MDA gidas nekalba apie modelių testavimą bei modelių korektiškumo užtikrinimą. Kuriant sistemą pagal MDA paradigmą, programuotojai daugiausiai dirbtų ties transformacijų kūrimu, todėl sistemos testavimas turėtų būti orientuojamas į transformacijų korektiškumo tikrinimą. Modeliai iš kitų modelių gaunami transformacijų dėka ir todėl transformacijų testavimas yra labai svarbi dalis kuriant sistemas remiantis MDA. MDA gidas teigia, kad sistemų kūrimas pagal MDA yra patikimesnis ir nuoseklus sistemų kūrimo procesas [MDA03], tačiau taip bus tik tuo atveju, jei modelių transformacijos bus vykdomos taip, kaip numatyta.

Modelių transformacijos irgi yra programos, kurios gali būti užrašomos įprasta bendro naudojimo (*angl.* general purpose) programavimo kalba (pvz., Java), QVT standartą atitinkančia transformacijų kalba ar naudojant transformacijų instrumentą, kur transformacijos apibrėžiamos naudojant instrumento siūlomą taisyklių rinkinį. Taigi, kaip ir kiekvienas programinis komponentas, modelių transformacijos turi atitikti tam tikrus specifikacijos reikalavimus ir funkcionuoti taip, kaip apibrėžta toje specifikacijoje. Standartinis validavimo būdas yra testavimas. Atrodytų, kad transformacijų testavimui galima būtų taikyti įprastas testavimo technikas, tačiau egzistuoja ir tam tikri skirtumai tarp įprastų programinių komponentų testavimo ir modelio transformacijų testavimo, kurie apsunkina įprastų testavimo technikų taikymą transformacijų testavimui. Svarbiausi jų [FSB04]:

- Transformacijos vykdomos modeliams, kurių duomenų struktūra gali būti sudėtinga. Tuo tarpu daugelis įprastų testavimo technikų skirtos darbui su paprastais duomenimis.
- Modelių duomenų struktūra apibrėžiama metamodeliu. Testuojant modelių transformacijas testavimo technikos turi lyginti, kad modeliai generuojami korektiškai ir atitinka metamodelį.

Fleurey et al. [FSB04] tokių duomenų testavimui siūlo taikyti skaidymo techniką, kai pradiniai testuojami duomenys skaidomi į keletą nepersidengiančių subdomenų (arba klasių). Naudojantis šia technika daroma prielaida, kad jei programa teisingai vykdo operacijas su viena duomenų klase, tuomet ji teisingai vykdys su visomis duomenų klasėmis. Tuomet kiekvienai klasei parenkami unikalūs duomenys, su kuriais tos klasės yra testuojamos. Modelių transformacijų pradiniai duomenys yra apibrėžti metamodeliu, tai leidžia suformuoti kriterijus metamodeliams, pagal kuriuos galima išskaidyti pradinius duomenis ir atrinkti testavimui naudojamus duomenų rinkinius [FSB04]. Pavyzdžiui:

- Kiekvieno metamodelio asociacijų ryšiui turi būti abiejų asociacijų ryšio dalyvių duomenų egzemplioriai, kurie bus naudojami testuojant.
- Kiekvienam klasės atributui turi būti testuojamų duomenų egzempliorius
- Kiekvienam generalizacijos ryšiui turi būti testuojami subtipų duomenų egzempliorius.

Testuojant modelių transformacijas taip pat būtina išspręsti šiuos klausimus [LZG05]:

1. *Automatinis modelių lyginimas*. Testuojant modelių transformacijas, turi būti atliekamas modelių (tikėtino ir gaunamo) lyginimas, kad nustatyti, ar egzistuoja skirtumai tarp modelių. Rankinis modelių lyginimas gali būti sudėtinga ir daug laiko reikalaujanti procedūra, kurią atliekant labai lengva suklysti. Efektyviausias būdas būtų automatinis modelių lyginimas, kuriuo turėtų pasižymėti testavimo instrumentas.
2. *Modelio skirtumų vizualizavimas*. Lyginant modelius, turi būti aiški ir intuityvi vizualizavimo priemonė, išskirianti skirtumus tarp modelių pačioje modeliavimo aplinkoje. Pavyzdžiui, grafiniai simboliai, formos ir spalvos gali būti naudojamos nurodyti, jei trūksta kurio nors modelio elemento ar modelio elementas yra perteklinis. Šie požymiai turi būti matomi modelių viduje. Taip pat turi egzistuoti galimybė efektyviai peržiūrėti visų skirtumų sąrašus ir per juos pasiekti problemines vietas. Šios technikos yra esminės vertinant testavimo įrankio rezultatus ir efektyvaus įrankio vaidmuo yra didžiulė atskleidžiant neatitikimus.
3. *Transformacijos specifikacijos derinimas* (*angl.* Debugging of transformation specifications). Nustačius, kad modelių transformacija vykdoma su klaida, turi būti analizuojama transformacijos specifikacija siekiant nustatyti klaidos šaltinį. Transformacijos derinimo įrankis galėtų padėti izoliuoti klaidos šaltinį ir jį nustatyti. Šis įrankis turėtų suprasti modelio reprezentaciją ir suteikti galimybę modeliavimo aplinkoje interaktyviai valdyti transformacijos specifikacijos komandų vykdymą žingsnio rėžimu ir intuityviai pateikti naudotojui modelio duomenis.

Yuehua Lin et al. [LZG05] pasiūlė savo karkasą spręsti 1 ir 2 klausimus (modelių skirtumo ir vizualizavimo). Šis karkasas remiasi modelių lyginimo technika. Modelis ir metamodelis vaizduojamas kaip grafas, sudarytas iš viršūnių ir briaunų. Tokiu būdu skirtumai tarp modelių analizuojant skirtumus tarp grafų. Karkaso autoriai pasiūlė ir algoritmą, kaip toks lyginimas galėtų būti realizuotas. Karkasas siūlo automatizuotą testų vykdymą tikrinant transformacijos specifikacijos korektiškumą ir vizualizuoja gautus neatitikimus. Toks karkasas labai priklauso nuo naudojamo transformavimo įrankio ir transformacijų specifikacijos kalbos, o taip pat ir nuo modeliavimo aplinkos, kadangi modelių grafai, pagal kuriuos bus lyginamas modelis, konstruojami pagal ryšius, kuriuos apibrėžia modeliavimo įrankis.

Taigi, įrankiai didžia dalimi gali palengvinti modelių ir jų transformacijų testavimą. Tinkami įrankiai galėtų užtikrinti tam tikrą modeliavimo ir transformacijų vykdymo valdymą, automatizuotų gerų praktikų naudojimą ir jau pradiniame etape leistų identifikuoti neatitikimus tarp modelių, užtikrintų trasavimą tarp modelio elementų, pakeitimų propagavimą ir modelių tarpusavio darną [PA09].

Deja, mums nepavyko rasti laisvai prieinamų įrankių, teikiančių minėtą funkcionalumą ar realizuojančių Yuehua Lin et al. aprašytą karkasą. Organizacija sistemų kūrimo infrastruktūroje turi pati susikurti tokius įrankius ir pagalbines priemones, palengvinančias transformacijų testavimą ir suteikiančias automatizuotą modelių integralumo ir darnos tikrinimą.

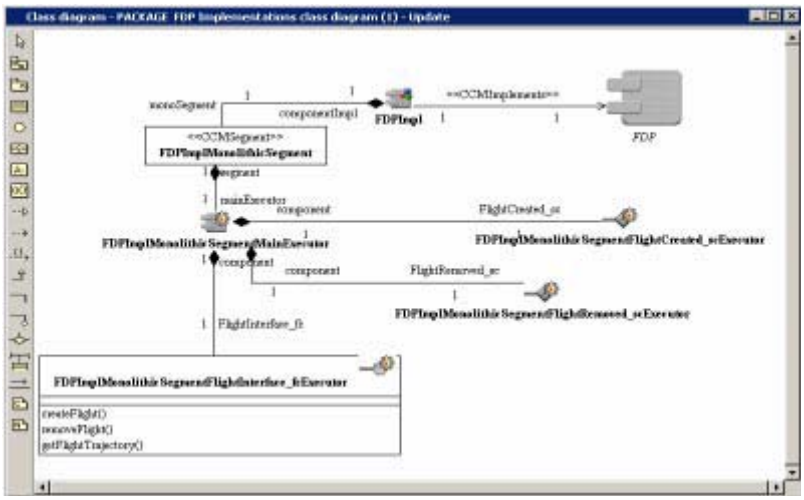
4.3.1.6 Diegimas

Diegimo fazės tikslas yra užbaigto programinio produkto perdavimas galutiniam vartotojui. RUP išskiria šias veiklas, kurios atliekamos šiame etape [Kru03]:

- Produkto testavimas galutinio vartotojo aplinkoje (*beta* testavimas);
- Produkto surinkimas pristatymui;
- Produkto platinimas;
- Produkto diegimas;
- Galutinio vartotojo apmokymas;
- Duomenų migravimas iš ankstesnės sistemos į naują.

Šio etapo pagrindinių vaidmenų (diegimo vadovo, projekto vadovo, techninės dokumentacijos rengėjo ir t.t.) atsakomybės susijusios su aukščiau minėtomis veiklomis ir diegimo darbo produktų ruošimu.

Kadangi kuriant sistemą pagal MDA visa projektinė informacija turėtų būti saugoma modeliuose, informacija apie diegimą, jo organizavimą, galutinio vartotojo aplinką ir pan. turėtų būti sudedama į diegimo modelį. Toks modelis turėtų būti glaudžiai susijęs su PSM modeliu, nes apibrėžia konkrečios platformos ir aplinko savybes, kur sistema turėtų veikti. Deja, nėra standarto, kaip turi būti modeliuojamas diegimas, koks modeliavimo profilis turėtų būti naudojamas, kaip apibrėžti platformos ir diegimo charakteristikas ir sieti jas su PSM modeliais. Mums pavyko rasti tik vieną MDA grįstą projektą, kuriame užsimenama apie diegimo modelį [MAS04]. Nesant standartui, projekto autoriai kūrė savo profilį, kuriuo apibrėžė diegimo modelį per diegimo diagramą ir sistemos komponentų surinkimo vaizdą (7 pav.).



7 pav. Komponentų surinkimo vaizdas [MAS04]

Deja, projekto autoriai nedetalizuoja šio proceso etapo ir pateikia tik trumpą diegimo etapo aprašą nedetalizuodami diegimo modelio profilio ir šio modelio elementų. UML neturi profilio, kaip modeliuoti ir apibrėžti diegimo aplinką ir susieti su PSM modeliais, dėl to MDA grįstame procese tektų kurti labai specializuotą UML profilį arba net atskirą DSL modeliavimo kalbą, kuri leistų susieti tokius elementus, kaip PSM modeliai, aplinkos specifikacija, sistemos reikalavimai, susiję su galutinio vartotojo aplinka, ir pan.

4.3.2 Sistemos palaikymas ir evoliucija

Sistemos palaikymas reikalingas siekiant užtikrinti, kad sistema toliau tenkintų vartotojo reikalavimus. Palaikymas taikomas nepriklausomai nuo to, kokį sistemų kūrimo procesą yra įsisavinusi organizacija ir vykdomas siekiant [Pig09]:

- Ištaisyti sistemos klaidas;
- Patobulinti architektūrinį projektą;
- Patobulinimus sistemos realizaciją;
- Papildyti sistemą naujomis sąsajomis su kitomis sistemomis;
- Adaptuoti sistemą siekiant suderinti ją su skirtinga aparatūrine ir programine įranga.

Programų sistemų gyvavimo ciklo procesų standartas IEEE/EIA 12207 apibrėžia palaikymą, kaip vieną iš pagrindinių gyvavimo ciklo procesų, kurio metu modifikuojamas programinis kodas ir susijusi dokumentacija siekiant išspręsti iškilusią problemą ar siekiant patobulinti produktą, tačiau išsaugant jo integralumą [Pig09]. Palaikymo kontekste sistemos evoliucionuoja ir laikui bėgant keičiasi ir šiuos pokyčius gali paskatinti tokie faktoriai, kaip [Kur05]:

- pasikeitusi verslo aplinka sistemai iškelia naujus reikalavimus. Dėl jų sistema gali būti papildoma nauju funkcionalumu.
- atsiradusios naujos programinės technologijos ir platformos, dėl ko sistema ar jos atskiri komponentai turi migruoti į šias technologijas;
- atskiros sistemos dalys turi būti modifikuojamos siekiant ištaisyti klaidas ar pagerinti jų kokybę (pvz., našumas, adaptacija pokyčiams ir pan.)
- ankstesni reikalavimai sistemai tampa nebeaktualūs ir tam tikros sistemos funkcijos ir jas realizuojantys sistemos komponentai turi būti pašalinti iš sistemos.

Sistemos gebėjimą palaikyti ir evoliucionuoti atspindi sistemos kokybinė savybė, vadinama sistemos adaptyvumu (*angl.* adaptability). Adaptyvumas – sistemos kokybinis gebėjimas, apibrėžiantis sistemos atsparumą aplinkos pasikeitimams nereikalaujant papildomų sistemos modifikacijų, nei numatyta tam tikslui sistemos kūrimo metu [Kur05]. Aplinka apibrėžiama plačiąja prasme apimant aukščiau minėtus sistemos pokyčių ir evoliucionavimo faktorius.

Kaip ir kiekvienas programų sistemų kūrimo procesas, MDA grįstas procesas turi atsižvelgti į šiuos aspektus ir užtikrinti sistemos adaptyvumą. Tačiau MDA grįstas procesas turi žvelgti į minėtus faktorius per modelio ir modelių transformacijų sampratas ir šiame kontekste palaikymas ir evoliucija įtakoja pokyčius modeliuose, modeliavimo kalboje ir transformacijose [Kur05].

4.3.2.1 Pokyčiai modeliavimo kalboje

OMG yra patvirtinusi UML standartą, kuris apibrėžia modeliavimo kalbą, skirtą „vizualizuoti, apibrėžti ir dokumentuoti programų sistemas“ [MDA03]. Kaip jau buvo minėta, MDA modeliai gali būti kuriami UML arba siauros dalykinės srities (DSL) modeliavimo kalba, neturinčia savo standarto. UML palaiko daugelis modeliavimo įrankių ir tai suteikia tam tikrą pasirinkimo laisvę modeliuotojui renkantis sistemų kūrimo įrankius. Tuo tarpu DSL kalba paprastai taikoma siaurai dalykinei sričiai ir paprastai nepritaikoma kitoms sritims. UML 2 versija palaiko 13 diagramų tipų, kuriomis modeliuotojas gali apibrėžti sistemą, tačiau jos plėtimo galimybės yra ribotos, kai reikia apibrėžti dalykinę sritį, kuriai UML neturi notacijų ir semantikos. OMG patvirtino visą eilę papildomų standartų, kurias siekiama standartizuoti MDA požiūrį, pvz.:

- XMI – modelių išsaugojimui ir pernešimui;
- MOF – metamodeliavimo kalbos standartas;
- QVT ir OCL standartai, skirti apibrėžti modelių transformacijas ir taisykles bei išraiškas modeliuose.

Visi šie standartai yra tampriai susiję tarpusavyje ir taikomi drauge (OCL apibrėžia taisykles UML modeliuose, UML apibrėžiamas per MOF, QVT skirta MOF atitinkančia modeliavimo kalba sukurtų modelių transformacijų specifikacijai apibrėžti ir naudoja OCL apibrėžti pradinius ir galutinius modelių elementus, ir pan.).

Standartų gausa ir naujos jų versijos apsunkina sistemos adaptyvumą, kadangi šiais standartais naudojamasi apibrėžti pagrindinius sistemos kūrimo artefaktus – modelius ir jų transformacijų specifikaciją. Be to, didelė standartų ir jų versijų įvairovė apsunkina įrankių, palaikančių šiuos standartus ir skirtingas jų versijas, atsiradimą ir vystymą. Šią problemą gerai iliustruoja sunkumai, kylantys dėl XMI standartų versijų įvairovės ir tarpusavio nesuderinamumų.

OMG pasiūlė naudoti XMI [XMI07] ir XMI-DI (XMI diagramų apsikeitimui) [XDI06] formatą modelių pernešimui iš vieno įrankio į kitą. XMI formatu išsaugotas modelis yra XML dokumentas. XML sėkmingai naudojamas daugelyje dokumentų ir modelių vaizdavimo standartų, yra gerai dokumentuotas, nepriklausomas nuo platformos formatas, turintis didelį palaikančių instrumentų skaičių [AP05]. XMI savo ruožtu naudojamas MOF grįsta modeliavimo kalba (pvz., UML) užrašytų modelių saugojimui. Šiuo metu jau yra išleista XMI 2.1.1 versija. Tai, kad XMI siejamas su MOF ir yra nepriklausomas nuo modeliavimo kalbos lėmė, kad [AP05; JS03]:

1. Įrankiai, naudojantys skirtingus UML standartus (pvz., 1.3 ir 1.4) negali keisti modeliais, net jei modelių saugojimui naudojamas XMI.
2. UML modeliai, išsaugoti XMI formate, praranda tą papildomą informaciją, kuri būna UML modelyje konkrečiame modeliavimo instrumente.

3. UML ir MOF metamodeliai neapibrėžia, kaip diagramose turi būti vaizduojami modeliai. Pavyzdžiui, UML modelis gali nurodyti, kad yra tokia klasė „Asmuo“, tačiau negali nurodyti, kad diagramoje ši klasė vaizduojama stačiakampiu tam tikroje pozicijoje, turinti konkrečius matmenis, spalvą, šriftą ir pan. Šiai situacijai spręsti OMG pasiūlė XMI-DI formato standartą [XDI06]. Šis formatas – tai nebe modelių apsikeitimo tarp instrumentų formatas, o metamodelis, apibrėžiantis diagramos informaciją.
4. XMI nėra nei API, apibrėžiantis, kaip turi būti skaitomas modelis (kaip pavyzdžiui, yra Java Metadata Interface arba Eclipse EMF), nei tarpsisteminis komunikavimo protokolas, koks pavyzdžiui yra HTTP. Tai reiškia, kad nėra jokių standartizuotų programinių komponentų, skirtų skaityti ir rašyti XMI dokumentus, nes tokių komponentų API ir komunikavimo mechanizmas nėra standartizuoti.
5. XMI neturi patikimo mechanizmo identifikuoti, kuris metamodelis buvo naudojamas dokumente. Problemą komplikuoja faktas, kad XMI 1 versija ir XMI 2 versija skirtingai apibrėžia metamodelį [AP05]. XMI 2 versijoje kiekvienas metamodelis susiejamas su viena ar keliomis XML vardų erdvėmis (*angl.* namespace). XML specifikacija apibrėžia vardų erdvę kaip tam tikrą teksto eilutę, kuri „apibrėžia resurso, su kuriuo susieta vardų erdvė, globalų vardą“ [XMI07], bet nėra reikalaujama, kad šis vardas būtų kur nors unikaliai apibrėžtas ir būtų tikrinamas apdorojant XML dokumentus. Tokiu būdu nėra galimybių susieti vardų erdvę su UML versija ar kita modeliavimo kalba. Net jei toks susiejimas egzistuotų, jis būtų suprantamas tik konkrečiam instrumentui. Kitam instrumentui tai tėra dirbtinai sukonstruota teksto eilutė, iš kurios negalima išvesti jokios informacijos, iš jos instrumentas negali nustatyti, kokiu metamodeliu remiantis šiame XMI dokumente užrašytas modelis.

OMG, patvirtindama naują standarto versiją, nepasiūlo tipinio sprendimo, kaip migruoti modelius, apibrėžtus naudojant seną standartą į naują, palikdama tai modeliavimo instrumentų gamintojų nuožiūrai.

Analizuojant šiuo metu prieinamų modeliavimo instrumentų galimybes matyti, kad ši problema iki šiol neišspręsta. Pavyzdžiui, kompanijos „No Magic“ Lietuvoje plačiai žinomas ir Lietuvos dukterinės kompanijos „Baltijos programinė įranga“ kuriamo UML modeliavimo ir projektavimo instrumento „MagicDraw“ paskutinė 16 versija geba įkelti tik UML 2 versijos modelius, bet be diagramos informacijos. Norint įkelti UML 1.3 specifikacijos modelius, reikia naudoti senesnę 9.5-ąją programos versiją[7]. Kompanijos Sparx projektavimo ir modeliavimo

[7] NoMagic Inc. svetainė (2009.03.14) http://www.magicdraw.com/

instrumentas „Enterprise Architect“ 7.31 versija papildomą informaciją, kuria papildo modelius, saugo XMI įrašo specialiose žymėse, kurių kiti modeliavimo instrumentai nesupranta ir neįkelia[8].

Taigi pokyčiai modeliavimo kalboje kuriant sistemas pagal MDA labai apsunkina sistemos adaptyvumą, nes:

- sudėtinga migruoti modelius į naują kalbos versiją, nes OMG nepateikia tipinių migravimo iš vieno standarto versijos į kitą sprendimų;
- Modeliavimo įrankiai nepalaiko visų OMG standartų versijų;
- Nepaisant standartų modeliai skirtinguose instrumentuose užrašomi skirtingai ir nėra tarpusavy pilnai suderinami.

4.3.2.2 Pokyčiai modeliuose ir transformacijų specifikacijoje

Vykdant pakeitimus vienuose modeliuose dažnai yra būtina, kad šie pakeitimai būtų paskleidžiami (propaguojami) į kitus modelius, nes tai garantuotų modelių taisyklingumą ir tarpusavio darną. Modelio pakeitimas gali sukelti pakeitimų bangą kituose modeliuose ir, jei tai būtų daroma ranka, toks procesas taptų sunkiai valdomas, nes programuotojui rankiniu būdu tektų tikrinti kiekvieną modelį ir paskleisti kitame modelyje padarytus keitimus. Modelių testavimo įrankiai padėtų atskleisti prieštaravimus, bet visgi yra būtina, kad modeliavimo įrankis galėtų automatizuotai paskleisti pakeitimus viename modelyje į kitus modelius bei paties modelio viduje. Paprastai keliami šie pakeitimų modelyje paskleidimo uždaviniai [AP04]:

- *Pašalinimo propagavimas*. Modelio transformacija gali suardyti ryšius tarp modelio elementų. Modelio elementai paprastai sudaro sudėtingą ryšių grafo medį, todėl būtina paskleisti pakeitimus, susijusius su elemento pašalinimu, į kitas modelio struktūrines dalis [AP04]. Pavyzdžiui, jei klasės diagramoje pašalinama klasė A, iš kitų klasių turi būti pašalinti atributai, kurių tipas yra A.
- *Pakeitimų propagavimas į XMI-DI diagramas*. Pakeitimai UML modelyje turi atsispindėti ir modelį vaizduojančioje diagramoje, kuri paprastai saugoma XMI-DI formate, kuris apibrėžia, kaip turi būti vizualizuojamas modelis. Ištrinti elementai turi būti taip pat pašalinti iš diagramos, nauji elementai turi atsirasti. Dėl to, kai kurie diagramos elementai ir objektai gali pakeisti savo mastelį (pvz., klasių diagramoje klasės objektas papildomas nauju elementu) [AP04].
- *Pakeitimų propagavimas į korektiškumo modelį*. Šis modelis saugo informaciją apie grupės modelių atitikimą funkciniams ir nefunkciniams reikalavimams. Esant modelių pakeiti-

[8] Enterprise Architect User Guide, Version 7.3.3

mams, pakeitimai turi būti paskleidžiami ir šiame modelyje ir patikrinamas modelio korektiškumas [AP04].

Nors šie uždaviniai atspindi bazinę pakeitimų propagavimo sampratą, tačiau jie neatskleidžia tų kliūčių, kuriuos reikia įveikti realizuojant pakeitimų propagavimo mechanizmą. Kadangi MDA gidas nesuteikia jokių gairių ir sprendimų, kaip užtikrinti pakeitimų propagavimą ir modelių sinchronizavimą, siekiant automatizuoti šį procesą, tenka spręsti šiuos uždavinius [XLH07; Tra05; Tra08]:

- Būtina aiškiai apibrėžti, ką reiškia modelių sinchronizavimas, kada modeliai yra sinchronizuoti, o kada – ne. Dabartinės transformacijų kalbos (pvz., ATL) neturi sinchronizavimo semantikos ir tam skirtų kalbos konstruktų, todėl siekiant užtikrinti modelių sinchroniškumą tektų plėsti esamas kalbas [XLH07].
- Realizuojant pakeitimų paskleidimo (propagavimo) mechanizmą, modelių sinchronizavimą turi būti įmanoma atlikti tiek tarp to paties abstrakcijos laipsnio modelių, tiek tarp modelių, esančių skirtinguose abstrakcijų lygmenyse (pvz., PIM, detalesnio PIM ir PSM). Pvz., jei modifikuojama programinio kodo lygmenyje, kaip įsitikinti ir užtikrinti, kad šie pakeitimai nepažeidė abstraktesnių modelių darnos? Jei to neįmanoma padaryti, tokiu atveju tie modeliai tėra sistemos dokumentacija, o ne pirminiai sistemos kūrimo artefaktai. Problemą gali apsunkinti ir tai, kad skirtingo abstrakcijos lygio modelių ir jų transformacijų užrašymui naudojamos skirtingos kalbos ir saugojimo formatai. Kaip jau minėta aukščiau, net ir naudojant tą patį XMI formatą modelių saugojimui, įrankiai skirtingai saugo jame informaciją.
- Pakeitimų paskleidimas turi būti nedestruktyvus: paskleidžiant pakeitimus iš pradinio modelio, turi būti išsaugoti bet kokie rankiniu būdu padaryti modelio pakeitimai;
- Ar pakeitimai turi vykti interaktyviai informuojant vartotoją apie būtinus pakeitimus galutiniame modelyje, ar pakeitimai daromi neinformuojant?
- Rankinis ar automatinis pakeitimu propagavimas? Kada yra įmanoma atlikti automatizuotai pakeitimų pasklidą, kada reikia juos apibrėžti rankomis?
- Betarpiško ir paketinio pakeitimų paskleidimo realizacija. Paketinis pakeitimų propagavimas paskleidžia vieno modelio pakeitimus į kitus modelius tik vartotojui tai nurodžius. Betarpiškas pakeitimų propagavimas paskleidžia pakeitimus vartotojui redaguojant modelį . Paketinis propagavimas galėtų paskleisti visą rinkinį pakeitimų, tuo tarpu betarpiškas –

vienu metu paskleistų pakeitimus po vieną ir šie pakeitimai savo apimtimi yra maži ir atominiai savo prigimtimi [Tra08].

- Pradinio ir galutinio modelio elementų susiejimas (trasavimas). Kiekvienas pakeitimų propagavimas reikalauja mechanizmo, kuris susieja (ir tam tikra prasme - atskiria) specifinius galutinio modelio elementus, atsiradusius pradinio modelio elementams pritaikius nurodytą transformacijos taisyklę. Atskyrimo funkcionalumas yra esminė propagavimo mechanizmo savybė, leidžianti užtikrinti galutinio modelio elemento pakeitimą, sukūrimą ar pašalinimą vykdant pakeitimų propagavimą [Tra05; Tra08].
- Pašalinimo aptikimas. Pradinio modelio elementų pašalinimas turi atsispindėti ir galutiniame modelyje jame pašalinant susijusius elementus (su sąlyga, kad pašalinimo paskleidimas nepažeis rankiniu būdu įvestų modifikacijų – pakeitimų, papildančių modelio turinį, bet ne struktūrą, sąlygojančią pakeitimų propagavimą).
- Korektiškumo patikrinimas ir prieštaravimų sprendimas. Dalis pakeitimų pradiniame modelyje negali būti sėkmingai paskleidžiami į kitus susijusius modelius, jei, pavyzdžiui, pakeitimai pradiniame modelyje nesuderinami su galutiniame modelyje esančiu elementu ar elementais [Tra08; MA08].
- Kaip apibrėžti ir automatizuoti transformaciją, kad ji ne tik galėtų paskleisti pakeitimus pradiniame modelyje į galutinį modelį, bet ir atvirkščiai - pakeitimai galutiniame modelyje atsispindėtų ir pradiniame? Šiuo metu egzistuojantys transformavimo įrankiai nesuteikia tokios galimybės, nes programuotojas šalia transformacijų specifikacijos turi programuoti modelių validavimą ir sinchronizaciją tikrinantį kodą [XLH07]. Tai apsunkina sistemos adaptyvumą, nes yra sudėtinga užtikrinti darnumą tarp sinchronizavimo kodo ir transformacijų specifikacijos, nekalbant jau apie modelių, kuriuos sieja ši transformacija, tarpusavio darnumo užtikrinimą.

4.4 Įrankių palaikymas

MDA taikymo praktikoje sėkmė didele dalimi priklauso nuo atitinkamų įrankių, leidžiančių vykdyti veiklas, susijusias su modeliavimu ir transformacijų apibrėžimu. Pastaruoju metu atsiranda vis daugiau įrankių, skirtų, skirtų palengvinti ir taikyti MDA požiūrį sistemų kūrimo proceso veiklose. Keletą jų čia pristatysime.

Rinkoje jau egzistuoja tiek komerciniai, tiek atviro kodo modelių transformavimo į modelius ar kodą įrankiai. Dauguma jų jau gali automatiškai transformuoti žemiausio lygio PSM modelius į kodą, tačiau kol kas neradome tokių laisvai prieinamų įrankių, kurie apimtų kokio nors programų

sistemų kūrimo metodo modelių transformacijų grandinę ir galėtų vykdyti pradedant abstrakčių PIM modelių transformacijas į detalesnį modelį ir baigiant – PSM modelių transformaciją į programinį kodą. Tačiau egzistuoja keletas atviro kodo įrankių, kuriais, detaliai apibrėžus transformacijas, atskiroje tos grandinės dalyje galima vykdyti modelių transformacijas į kitus modelius ar programinį kodą:

1. Eclipse metamodelių kūrimo karkasas
2. openArchitectureWare
3. Fornax platforma

4.4.1 „Eclipse“ metamodelių kūrimo karkasas

Eclipse.org yra atviro kodo bendruomenė, kurios projektai orientuoti į programinių produktų kūrimo platformos, žinomos „Eclipse“ vardu, kūrimą ir plėtojimą. Dauguma programuotojų žino ir naudoja „Eclipse“ kaip Java integruotą programavimo terpę (IDE), tačiau su „Eclipse“ vardu šiuo metu taip pat siejama daugiau nei 60 vykdomų projektų, apimančių tokias sritis, kaip taikomųjų programų sistemų karkasų kūrimas (*angl.* Application Frameworks), programų sistemų gyvavimo ciklo valdymas (*angl.* Application Lifecycle Management), į paslaugas orientuota architektūra (SOA) ir pan. Vienas iš projektų, susijusių su MDA ir kuriamų „Eclipse“ platformos pagrindu yra „Eclipse“ modeliavimo karkaso projektas (*angl.* Eclipse Modeling Framework Project – EMF). EMF – tai metamodeliavimo karkasas ir programinio kodo generavimo priemonė, skirta kurti kitus įrankius ir taikomąsias sistemas, grįstas struktūrizuotais duomenų modeliais. EMF buvo pradėtas kurti kaip MOF standarto realizacija, šiuo metu palaiko MOF 2.0 standartą ir yra vienas iš MDA įrankių kurti ir aprašyti metamodelius. Naudojantis EMF, naują metamodelį galima kurti šiais būdais [Ste08; BSM+03]:

1. Naudoti grafinę EMF IDE, kuri metamodelį vizualizuoja kaip medį ir leidžia kurti to medžio mazgus bei apibrėžti ryšius tarp mazgų.
2. Metamodelį apibrėžti Java kalbos interfeisu. Šio interfeiso metodai privalo turėti atitinkamas anotacijas.
3. UML klasių diagrama.
4. XML Schema dokumentu.

Kiekvienas EMF metamodelis aprašo tokias jį reprezentuojančio modelio savybes [Ste08]:

- Objekto atributai;
- Ryšiai (asociacijos) tarp objektų;

- Operacijos, kurias galima atlikti su kiekvienu objektu;
- Paprasčiausi apribojimai (*angl.* constraints) objektams ir jų tarpusavio ryšiams.

EMF karkasas metamodelį saugo XMI dokumente – standartizuotame MOF modelių užrašymo formate. Turėdamas modelio specifikaciją, aprašytą XMI, EMF suteikia įrankius iš šio metamodelio generuoti grafinį redaktorių, kuriuo galima redaguoti ir modifikuoti šiuos modelius. Redaktorius – tai „Eclipse“ IDE pagrindu veikiantis vizualus redaktorius (*angl.* Eclipse Graphical Modeling Framework – GMF), kuriame pateikiami grafiniai modelio redagavimo įrankiai ir priemonės.

Kaip jau minėta, visas EMF karkasas ir jo komponentai veikia virš „Eclipse“ platformos. Kiti įrankiai, tokie, kaip žemiau pristatomi openArchitectureWare ar Fornax platformos karkasai, naudojasi EMF karkasu modelių kūrimui, redagavimui ir kodo generavimui.

4.4.2 openArchitectureWare

openArchitectureWare (oAW) yra Eclipse platformos pagrindu sukurtas atviro kodo įrankių rinkinys, skirtas padėti kurti MDA grįstas programų sistemas bei kitus MDA įrankius [OAW08]. Jis gali tiek importuoti įvairių tipų modelių formatus, tiek rašyti į įvairių tipų formatus. oAW įrankiai (redaktoriai ir kodo generatoriai) veikia virš Eclipse platformos. Veikdamas virš Eclipse EMF karkaso, oAW palaiko ir dirba su EMF karkaso modeliais, užrašytais XMI formatu, o taip pat gali priimti modelius, užrašytus UML ar XML dokumentais. Naudodamas Eclipse platformą, oAW suteikia priemones redaguoti ir aprašyti modelius, metamodelius, šablonus, transformacijų eigą ir taisykles [OAW08]. oAW karkasas grįstas moduline struktūra, kuri leidžia išplėsti bet kurį karkaso modulį, keisti ir papildyti jo funkcionalumą. Kiekvienas iš šių modulių atsakingas už atskirą oAW teikiamą funkcionalumą. Svarbiausi oAW moduliai yra Xtext karkasas, Xtend transformacijų kalba ir redaktorius, Xpand kodo generavimo šablonų kalba ir redaktorius, bei oAW proceso eigos modulis [Vol07; HVE+07]. Šie moduliai žemiau yra detaliau aptariami.

4.4.2.1 Xtext karkasas

Xtext yra tekstinių dalykinės srities kalbų (DSL) kūrimo karkasas, kuris integruojasi į Eclipse IDE redaktorių ir leidžia kurti DSL kalbas turint Bakus-Nauro forma (BNF) užrašytą kalbos sintaksę. Iš jos Xtext karkasas gali generuoti:

- šios kalbos redaktorių. Šis redaktorius veikia virš Eclipse IDE ir palaiko redaguojamo kodo reikšminių žodžių paryškinimą (*angl.* syntax highlighting), automatinį žodžio užbaigimą (*angl.* auto-completion), ribojimų tikrinimą, navigaciją ir t.t.

- kalbos sintaksinį analizatorių (*angl.* parser), kuris leidžia modeliuojant realiu laiku tikrinti sintaksę. Jį oAW proceso eigos modulis taip pat naudoja kaip metamodelį vykdant modelių transformacijas.

4.4.2.2 Xtend transformacijų kalba ir jos redaktorius

Xtend yra su ATL suderinama kalba, kuri naudojama:

- aprašyti modelių transformacijas sukuriant naujus modelius iš jau esančių;
- modelių validacijai. Modelių apribojimai ir validacija vykdoma remiantis taisyklėmis, užrašytomis Xtend kalba;
- aprašyti jau esančių modelių modifikacijas;
- metamodelių išplėtimui. Tai leidžia dinamiškai papildyti metamodelį naujomis savybės fiziškai nemodifikuojant paties metamodelio.

4.4.2.3 Xpand šablonų kalba ir jos redaktorius

oAW gali generuoti bet kokios programavimo kalbos išeities tekstus (o taip pat ir paprastus tekstinius failus) naudodamas Xpand šablonų kalbą. Kiekvienas šia kalba užrašytas sakinys nurodo atskirą formuojamą išeities teksto fragmentą. Xpand šablonai aprašomi naudojant šios kalbos redaktorių, integruotą į Eclipse IDE, ir tokiu būdu palaiko tokias redagavimo pagalbines funkcijas, kaip reikšminių žodžių paryškinimas (*angl.* syntax highlighting), automatinis žodžio užbaigimas (*angl.* autocompletion), ribojimų tikrinimas ir pan.

4.4.2.4 oAW proceso eigos modulis

oAW šerdis yra proceso eigos (*angl.* workflow) modulis, apjungiantis aukščiau minėtus modulius į integruotą sistemą ir naudojantis tų modulių teikiamą funkcionalumą. Proceso eiga aprašoma XML faile, kurį vykdydamas proceso eigos modulis gali skaityti įvairių tipų formatų modelius, kurti jų egzempliorius, patikrinti, ar jie atitinka metamodelyje apibrėžtus apribojimus, vykdyti modelių-į-modelį bei modelių-į-kodą transformaciją ir rezultatą išsaugoti įvairių formatų išeities tekstuose.

4.4.3 Fornax platforma

Fornax platforma yra atviro kodo platforma, skirta kurti komponentus ir įrankius, naudojamus kuriant MDA paradigma grindžiamas programų sistemas [FOR08]. Naudodama ir praplėsdama openArchitectureWare karkasą, Fornax platforma siūlo infrastruktūrą ir priemones šių komponentų kūrimui. Vienas iš projektų, vykdomų naudojant Fornax platformą, yra taip vadinamos „UML2

kasetės“ (*angl.* UML2 Cartridges). Šios kasetės – tai oAW modulių plėtiniai, kuriais oAW kodo generatoriaus karkasas papildomas modelių transformacijomis į specializuotą Java programinį kodą, naudojantį tokių Java taikomųjų sistemų karkasų, kaip Spring, Hibernate ir EJB, teikiamą funkcionalumą. Kiekviena kasetė turi UML2 stereotipų ir oAW šablonų rinkinį, naudojamą generuoti specializuotą konkrečiam karkasui programinį kodą. Šiuo metu yra sukurti šių Java taikomųjų sistemų karkasų kasetės:

- Hibernate objektų – reliacinių duomenų bazių susiejimo karkaso kasetė;
- EJB2 verslo sistemų komponentų karkaso kasetė;
- Spring2 taikomųjų sistemų karkaso kasetė.

Programinio kodo generavimą vykdo openArchitectureWare įrankis, į kurį integruota atitinkama kasetė. UML modeliai turi būti pažymėti naudojant tos kasetės siūlomu UML stereotipų rinkiniu. Stereotipai yra susieti su šablonais, kurių dėka oAW įrankis gali generuoti konkrečiam karkasui specializuotą programinį kodą ir konfigūracinius failus. Pvz., naudojant Hibernate kasetę, generuojamas su Hibernate karkasu suderinamos Java klasės – esybės bei atitinkami konfigūraciniai Hibernate XML failai.

Fornax nesiūlo modelio transformacijų į modelį funkcionalumo. Pradiniai modeliai, iš kurių generuojamas programinis kodas, turi būti paties žemiausio PIM abstraktumo lygio, nes, pavyzdžiui, naudojant Hibernate kasetę, jos UML2 stereotipų rinkiniu reikia modelyje nurodyti, kur esybė, kuris atributas yra pirminis raktas, kokiais asociacijų ryšiais susijusios klasės, kokį klasių paveldėjimo metodą taikyti ir pan. [Joc08]. Todėl Fornax platforma tinkama galutiniame modelių transformacijų grandinės dalyje, kai reikia modelius transformuoti į programinį kodą.

4.4.4 Sistemų kūrimo karkasai

Verslo IT organizacijai reikia viso rinkinio įrankių ir metodų, kuriuos programų sistemų inžinieriai galėtų lengvai taikyti ir naudoti. Nors, kaip minėta, jau egzistuoja visa eilė vertingų technologijų ir įrankių, kurie galėtų būti naudojami, jie turi būti glaudžiai integruoti į sistemų kūrimo procesą, pasižymėtų plečiamumu ir orientuoti į specifines dalykines sritis ir architektūrinius stilius.

Šiuo metu organizacijos informacinės sistemos (IS) architektūros (*angl.* Enterprise Architecture) modeliai retai naudojami kaip pradiniai duomenys kuriant žemesnio abstrakcijos lygio modelius. Dabartiniai transformacijų įrankiai nesuteikia galimybių iš šių architektūros modelių gauti programinis kodą, dokumentaciją ar diegimo modelius. Be to, transformacijų įrankiai paprastai geba dirbti tik su vieno metamodelio modeliais, tuo tarpu verslo modelių informacija gali

apimti kelis metamodelius [Bro08]. Šiuos klausimus reikia išspręsti renkantis ir integruojant įrankius į programų sistemų kūrimo procesą. Tam tikslui reikia viso rinkinio įrankių, praktikų, pagalbos žinynų, specializuoto turinio (šablonų, karkasų, dalykinės srities modelių), surinkto ir skirto verslo organizacijai projektuojant, kuriant, diegiant ir valdant verslui skirtus programų sistemų sprendimus [Bro08]. Tai gali būti taip vadinamas organizacijos IS architektūros MDA karkasas (*angl.* MDA Framework for Enterprise Architecture). Šiuo metu plačiausiai žinomi yra:

- IBM MDA karkasas, grindžiamas IBM Rational Software Modeler (RSM) programine platforma,
- bei Microsoft Software Factories, grindžiama Microsoft Visual Studio programine platforma.

Abu šie karkasai grindžiami komerciniu būdu platinama programine įranga. Be to, Microsoft Software Factories karkasas netaiko ir nesilaiko OMG patvirtintų modeliavimo standartų (UML, XMI, OCL ir pan.) [Coo04], tuo tarpu IBM MDA karkasas taiko taip vadinamą „pragmatinį“ požiūrį į MDA ir OMG patvirtintus standartus perima tiek, kiek tai yra naudinga jų siūlomam karkasui ir produktams [Bro08]. Pavyzdžiui, modeliavimui RSM naudoja SysML modeliavimo kalbą, kuri viena vertus yra UML poaibis, kita vertus naudoja modeliavimo koncepcijas, kurių UML nepalaiko: reikalavimų modeliavimas, elgesio ir struktūros modeliavimas (naudojant skirtingą, nei apibrėžta UML, semantiką) ir pan. [SCG+05; NBB+08].

Bet kuriuo atveju, jei organizacija jau yra adaptavusi sistemų kūrimo procesą ir instrumentinių priemonių karkasą, MDA požiūris verčia jį keisti ir adaptuoti bei į procesą integruoti naujus įrankius. Tokiu atveju MDA sistemų kūrimo karkasai gali būti tas sprendimas, kuris palengvintų MDA požiūrio taikymą įmonėje adaptuotam procesui, nes integruoja visą eilę įrankių, euristinių metodų, praktikų, gairių, padedančių IT verslo įmonei projektuoti, kurti, diegti, valdyti ir palaikyti programų sistemų sprendimus taikant MDA požiūrį.

4.5 UML ir DSL naudojimas sistemų apibrėžimui

Sistemos apibrėžimui modeliais MDA grįstame procese turi būti naudojama modeliavimo kalba. Tam tikslui OMG patvirtino UML modeliavimo kalbos standartą, skirtą „vizualizuoti, apibrėžti ir dokumentuoti programų sistemas“ [cit. pgl. MDA03].

UML yra naudojamas labai plačiai, nes leidžia vizualiai pavaizduoti sprendimų idėjas ir tokiu būdu UML padeda šias idėjas aptarti ir suprasti ties programų sistemų projektu dirbantiems skirtingų profesijų atstovams ir suinteresuotiems asmenims. UML tokiu atveju naudojamas kaip

eskizas, paprastai piešiamas ant lentos ar popieriaus lapo, ir toks jo naudojimo būdas yra matyt labiausiai paplitęs [Fow03].

Kita UML taikymo sritis – detalus sistemos projektavimas (*angl.* blueprint). Tokiu atveju sistemos architektas naudoja UML detaliai apibrėžti sistemos komponentų sąsajas ir tų komponentų elgesį, ir pagal šį apibrėžimą programuotojai rašo programinį kodą. Skirtingai nuo eskizinių modelių, modeliais apibrėžtas detalus projektas turi būti pakankamai pilnas, kad programuotojai galėtų juo vadovautis ir daug negalvodami rašyti programinį kodą [Fow03]. Tokiems modeliams kurti jau reikalingi specializuoti įrankiai, suteikiantys galimybę norimu detalumu apibrėžti sistemą, jos komponentus, sąsajas, elgesį.

Vadovaujantis MDA požiūriu, modeliuose turi būti sudedama visa projektinė informacija ir sistema turi būti apibrėžiama ir kuriama modeliais. Tokiu būdu jau patys UML modeliai taptų programavimo kalba, o veikianti sistema gaunama vykdant transformacijas iš modelių į veikiantį programinį kodą. Architektai ir programuotojai dirba ties modeliais ir transformacijomis, o ne su vykdomu programiniu kodu, nes šis gaunamas modelių transformacijų dėka. UML užrašyti modeliai ir jų transformacijos tampa pirminiais sistemos kūrimo artefaktais. Skirtingai nuo UML naudojimo eskizams ar detaliam sistemos apibrėžimui, naudojant UML kaip programavimo kalbą reikia griežtai laikytis OMG patvirtinto UML standarto ir UML diagramose apibrėžtos semantikos, nes priešingu atveju nebus įmanoma sėkmingai kompiliuoti modelių, vykdyti transformacijų, užtikrinti modelių pernešimo tarp įrankių ir pan. Kadangi UML standartas neapima skirtingų dalykinių sričių sąvokų ir koncepcijų, UML specifikacija leidžia naudoti profilius, kuriais galima išplėsti modelių apibrėžimą papildant juos naujais stereotipais, žymėmis (*angl.* Tag) ir ribojimais (*angl.* Constraint). Pavyzdžiui MOF metaklasė „Verslo objektas“ gali būti susiejama (*angl.* Mapped) su UML klasės elementu, turinčiu stereotipą „verslo objektas“. Tokiu būdu kiekvienas MOF metamodelio elementas gali būti siejamas su UML elementais naudojant šias išplėtimo technikas, aprašytas profilyje. Tai leistų metamodelio lygyje apibrėžti visas sąvokas, reikalingomis skirtingose abstrakcijos lygiuose aprašyti modelius, bei veiksmus su šiomis sąvokomis – modelių transformacijas.

Kita vertus, MDA grįstame programų sistemų kūrimo procese, naudojant UML kaip programavimo kalbą, neišvengiamai tektų susidurti su šiomis kliūtimis:

- UML yra sudėtinga kalba, turinti 13 skirtingų diagramų tipų, kiekviena diagrama turi skirtingą semantiką ir notacijas. O ir patys modeliai paprastai yra labai sudėtingos struktūros, turinčios daug simbolių ir tekstinių elementų, kuriuos reikia atidžiai analizuoti siekiant suprasti tikrąją jų prasmę. Yra būtinas supratimas, kaip šie elementai veikia,

siekiant sukurti sudėtingą modelį. Iš to seka, kad būtina nemažai investicijų, kad tapti geru modelių kūrėju [Coo04].

- UML modelis turi turėti pakankamai informacijos, kad leistų sėkmingai vykdyti transformacijas į veikiantį programinį kodą, tačiau UML standartas neleidžia papildyti modelio elementais ir diagramų tipais, kurių standartas neapibrėžia. Pvz., laikantis UML standarto nėra galimybės UML diagrama apibrėžti elektrinę schemą arba pavaizduoti vartotojo sąsajos langų tarpusavio ryšius ir vaizdavimo seką [Fow03].
- UML, kaip programavimo kalbos, naudojimas turi būti efektyvesnis ir produktyvesnis būdas kurti sistemą, nei kiti. Deja, įvairių autorių teigimu, tai nėra akivaizdu [Coo04; Fow03; Tho04]. Be to, trūksta sėkmingo jos taikymo pavyzdžių ir technikų, demonstruojančių šios kalbos pranašumą prieš kitas.

Didelis UML sudėtingumas iš dalies lėmė dalykinės srities kalbų (*angl.* Domain Specific Language – DSL), naudojamų dalykinės srities modeliavimui, atsiradimą. Specifinis dalykinės srities modeliavimas (*angl.* Domain-Specific Modelling – DSM) turi du pagrindinius tikslus [KT08]:

1. Iškelti sprendimo kūrimą į aukštesnį abstrakcijos lygį, kur sprendimo specifikacija yra išreiškiama kalba, kurios sąvokos ir taisyklės yra specifiškos tai dalykinei sričiai;
2. Generuoti galutinį produktą iš šios aukšto lygio specifikacijos.

Modeliuojant sistemą naudojant UML, dalykinės srities elementus reikia pirmiausiai susieti su UML objektais, o šiuos – transformuoti į kodą. Gautas UML modelis labiau atitinka sprendimo specifikaciją, o ne dalykinės srities specifikaciją.

DSM šalininkų teigimu, UML modeliai, sukūrus sprendimą, kitur jau nebepanaudojami, nes jie menkai atitinka dalykinės (probleminės) srities sąvokas ir elementus [KT08]. DSM paradigma grindžiama sistemos, modeliavimu naudojant dalykinės srities sąvokomis ir kodo generavimu būtent iš šių modelių. Programinio kodo generatorius nebūtų visą apimantis – jis būtų specifiškas tai dalykinei sričiai ir rašomas tos dalykinės srities eksperto [KT08; Eva04]. DSM paradigma teigia, kad tokiu būdu sistemą galėtų kurti dalykinės srities ekspertas, kuris pats net nėra programuotojas. Naudodamas savo srities terminus jis galėtų apibrėžti sistemos specifikaciją, o kodo generatorius sugeneruotų veikiančią sistemą.

Kita vertus, norint, kad DSM būtų veiksmingas ir efektyvus, būtina, kad dalykinė sritis būtų kuo siauresnė [KT08]. Kompanijos viduje DSM sprendimas apima tik dalį jos verslo. Pavyzdžiui banko ar automobilių gamybos dalykinės sritys būtų per plačios, kad jas modeliuoti su viena DSL. Banko atveju, DSM sprendimas galėtų apimti paskolų išdavimo arba investavimo sritis. Automobilių gamybos atveju – šviesų sistemos ar informacinės panelės gamybos sritis. Kuo

siauresnė dalykinė sritis, tuo lengviau yra apibrėžti efektyvią srities kalbą ir automatizuoti programinio kodo generavimą [KT08]. Remiantis DSM paradigma programų sistemų kūrimas – tai modelių generatorių kūrimas. Kadangi transformacijas iš modelio į modelį DSM paradigma atmeta [KT08], toks generatorius turi būti:

a) arba labai sudėtingas, kad galėtų vienu šuoliu įveikti didžiulę abstrakcijos spragą tarp dalykinės srities modelių, neapibrėžiančių sistemos architektūros ir realizacijos, ir programinio kodo. Bet tada kyla klausimas, kuo generatoriaus kūrimas skiriasi nuo pačios sistemos kūrimo įprastu klasikiniu būdu?
b) arba taikomas ypač siaurai dalykinei sričiai, kaip pvz. įdėtinės (*angl.* embeded) technologijos, kur šiuo metu DSM gali pasigirti tam tikrais pasiekimais kuriant programinę įrangą mobiliems telefonams [KT08].

Be to, DSL kūrimas yra sudėtingas uždavinys, nes reikalauja tiek gerų tos dalykinės srities žinių, tiek kalbų kūrimo patirties. Priklausomai nuo taikymo srities apimties bei naudotojų bendruomenės dydžio, DSL kalbos mokymo medžiagos paruošimas, standartizavimas ir palaikymas gali tapti sunkiai įveikiamu ir daug laiko resursų reikalaujančiu uždaviniu [Haa08].

5 Apibendrinimas

MDA paradigmą OMG iškėlė reaguodama į vis augantį sistemų kūrimo sudėtingumą, susijusį su technologinių platformų evoliucija, augančiais sistemų kūrimo ir palaikymo kaštais, augančiais ir nuolat besikeičiančiais reikalavimais sistemoms ir sistemų klasėms. Didelė dalis sistemai keliamų reikalavimų apibrėžiami verslo dalykinės srities lygmenyje, kuris yra aukštesniame, nei platforma, abstrakcijos lygmenyje. Tuo tarpu esami šių reikalavimų sprendimai pernelyg stipriai susipynę su konkrečia platforma. Nors įprastinis sistemų kūrimas komponentais ir karkasų naudojimas padeda siekiant padaryti verslą efektyvesnį, nes leidžia pakartotinai panaudoti komponentus, tačiau šis panaudojimas apsiriboja tik konkrečia platforma. Be to, platformų evoliucija ir naujų technologijų atsiradimas labai padidina tokių komponentų palaikymo kaštus, nes komponentai pernelyg priklausomi nuo egzistuojančių sistemų ir platformų realizacijų, o tai savo ruožtu apsunkina pakartotinį komponentų panaudojimą.

Komponentų pakartotinis panaudojimas yra vienas iš kertinių principų siekiant efektyvinti verslą ir leidžiantis sumažinti sistemos kūrimo kaštus. Sparti platformų (Java EE, .NET) evoliucija šiuo metu apsunkina pakartotinio ankstesnių sprendimų panaudojimo problemą, todėl siekiant sumažinti komponentų jautrumą platformų evoliucijai, o tuo pačiu ir platesnį jų panaudojimą, sistema ir jos komponentai turi būti apibrėžiami tame abstrakcijos lygmenyje, kuriame

neatsižvelgiama į platformą, o MDA požiūriu – nuo platformos nepriklausomais modeliais. MDA paradigma teigia, kad sistemos apibrėžimas nepriklausomais nuo platformos modeliais padeda spręsti pakartotinio žinių ir sprendimų panaudojimo kuriant sistemas problemą. Siekdama standartizuoti šį MDA požiūrį, OMG patvirtino visą eilę standartų, bet esminiai MDA principai ir praktikos glūdi modeliais grįstame sistemų kūrimo stiliuje, kuris yra fundamentalus sistemų inžinerijoje.

Tačiau pirminis entuziazmas MDA idėjomis, įvertinus jos taikymo praktikoje sunkumus, didžia dalimi išsisklaidė ir, kaip matyti šiame darbe, daug autorių pradėjo skeptiškai vertinti MDA požiūrį. Kai kurių jų tvirtinimu [Zet06] vargiai rasime organizacijų, kurios pilnai adaptavo MDA požiūrį griežtai laikydamiesi OMG patvirtintų su modeliais ir modeliavimu susijusių standartų. Kaip teigia Microsoft kompanijos verslo sistemų architektas, vienas iš OCL kalbos kūrėjų Steve Cook [Coo04], MDA negali vadintis architektūra, nes nesprendžia problemų, susijusių su modelių, šablonų, karkasų ir įrankių naudojimu programų sistemų inžinerijos procese. Cook teigimu, MDA tėra *standartizuotas požiūris* į modeliais grįstą sistemų kūrimą remiantis abstrakčiame lygmenyje apibrėžtais platformų panašumais.

Nepaisant šiame darbe cituojamų autorių skepticizmo, požiūris į efektyvų automatizuotą sistemos kūrimą negali apsieiti be sistemos apibrėžimo nepriklausomais nuo platformos modeliais ir jų transformacijų, ir jie yra kertiniai tokio požiūrio komponentai. Reikia atsižvelgti į tai, kad MDA paradigma yra dar gana jauna ir, skirtingai nuo klasikinių procesų, dar nėra nusistovėjusių praktikų kaip taikyti MDA požiūrį programų sistemų kūrimo procese. MDA neapibrėžia jokių specifinių sistemų kūrimo metodikų, nesuteikia jokių proceso gairių ir nenurodo proceso veiklų, etapų (fazių), vaidmenų ir jų atsakomybių. Kiekviena organizaciją, norėdama taikyti MDA požiūrį kuriant sistemas, turi pati atrasti tas praktikas, kaip kurti sistemą pagal MDA paradigmą bei pritaikyti šį požiūrį organizacijoje jau adaptuotam procesui, nes daugeliui proceso etapų ir veiklų nėra standartų, pripažintų gerųjų praktikų ir pavyzdinių sprendimų.

Tačiau MDA grįstas procesas savo ruožtu pasipildytų papildomomis veiklomis ir proceso veikėjų atsakomybėmis, susijusiomis su modeliavimo kalbų ar jos plėtinių apibrėžimu, modeliavimu, transformacijų apibrėžimu ir t.t. Tai reikalauja didelių investicijų, resursų ir laiko sąnaudų, nes siekiant sėkmingai sukurti programų sistemą, architektai ir programuotojai turi turėti didelę modeliavimo kalbų kūrimo, modeliavimo ir transformacijų apibrėžimo patirtį ir žinių. Kai kurie autoriai, atsižvelgdami į UML sudėtingumą, abejoja, ar toks sistemos kūrimas modeliais vadovaujantis OMG patvirtintais standartais, yra efektyvesnis būdas, nei kiti [Fow03; Zet06; Tho04]. Kadangi šiuo metu dar nėra įmanoma pilnai modeliais apibrėžti sistemą, dėl to tampa

sunkiau pritaikomas mažose kompanijose, neturinčios nusistovėjusio programų sistemų kūrimo proceso, adaptuoto metodo ir negalinčių tiek investuoti į MDA požiūrio pritaikymą procese.

MDA požiūris labiau tinka stambioms IT verslo įmonėms, kuriančioms konkrečios verslo dalykinės srities programų sistemų sprendimus. Tokiose įmonėse paprastai būna nusistovėjęs brandus programų sistemų inžinerijos procesas bei adaptuotos sistemų kūrimo metodikos, jos paprastai vykdo daug užsakymų, orientuotų į tam tikrą verslo dalykinę sritį, todėl pakartotinis komponentų (modelių) panaudojimas ir automatizuotas sistemų, orientuotų į skirtingas platformas, kūrimas siekiant efektyvinti verslą yra ypatingai aktualus.

Siekiant palengvinti MDA požiūrio adaptavimą programų sistemų kūrimo procese, kai kurių autorių teigimu [Coo04; Bro08; Ski08] būtinas tam tikras *pragmatinis* požiūris į pačią MDA paradigmą bei su ja susijusius OMG standartus, perimant tas gerąsias idėjas, kurias organizacija gali pritaikyti savo procese. Sėkmingam ir efektyviam MDA požiūrio taikymui sistemų kūrimo procese turi egzistuoti MDA sistemų kūrimo karkasai, apibrėžiantys ir į sistemų kūrimo procesą integruojantys technologijas, gebėjimus, instrumentus, veiklas, metodus ir darbo produktus (turinį) [Bro08]. Toks karkasas suteiktų organizacijai dalykinės srities platformą, kurią būtų galima naudoti kuriant specifinius įmonės verslui į paslaugas orientuotus sprendimus.

Šiuo metu egzistuoja tokie komerciniai MDA sistemų karkasai, kaip Microsoft Software Factories, IBM RSM grįsta MDA platforma, MetaEdit [9], Mendix[10] ir kt. Tokie karkasai integruoja visą eilę automatizuotų įrankių ir technologijų, kaip kurti modeliavimo kalbas, kurti modelius, šiuos modelius integruoti ir testuoti, apibrėžti modelių transformacijas ir pan. Jie taip pat suteikia tam tikros srities modeliavimo kalbą (Mendix, IBM) arba suteikia priemones susikurti DSL kalbą (MetaEdit, Microsoft DSL Tools, openArchitectureWare). Šie karkasai skirtingu lygmeniu apibrėžia OMG standartų palaikymą arba visai to nedeklaruoja, tačiau jos visos remiasi MDA idėjomis suteikdami galimybę organizacijoms projektuoti sistemas nepriklausomais nuo platformos modeliais, kurie gali būti pakartotinai panaudojami kuriant kitus tos pačios dalykinės srities sprendimus. Naudojant tokį karkasą, MDA gali būti apibrėžiama kaip verslo informacinių sistemų kūrimo ir integravimo *stilius*, grindžiamas automatizuotų instrumentų naudojimu kuriant nepriklausomus nuo platformos sistemos modelius [Bro08].

Kita vertus, MDA sunkiai pritaikomas kuriant sistemas, kurioms keliami dideli kokybiniai ir nefunkciniai reikalavimai, nes šie reikalavimai paprastai glaudžiai susiję su naudojama programine ir aparatūrine platforma (pvz., reikalavimai programos dydžiui, našumui, patikimumui ir pan.).

[9] http://www.metacase.com
[10] http://www.mendix.com

Tačiau daugumai verslo informacinių sistemų nekeliami tokie reikalavimai ir šių sistemų kūrimas sudaro didžiąją daugumų visų kuriamų sistemų. Į šių verslo sistemų kūrimą ir orientuojasi MDA siekdama apibrėžti požiūrį, kaip tokias sistemas būtų galima kurti efektyviau.

Rašydami šį darbą dar neradome tyrimų, kurie pristatytų, palygintų ar įvertintų sistemų kūrimo efektyvumą naudojant šiuos karkasus. Kita vertus, šie karkasai yra kuriami siekiant palengvinti MDA taikymą procese programų sistemų kūrimo procese atsižvelgiant į tuos sunkumus, apie kuriuos buvo rašoma šiame darbe. MDA karkasų, instrumentinių modeliavimo ir modelių transformavimo priemonių, kuriamų tokių kompanijų, kaip IBM, Microsoft, Borland ir kt., atsiradimas ir vystimas liudija, kad IT rinka, nepaisant kiek išblėsusio entuziazmo, priėmė OMG pristatytą MDA požiūrį ir palaipsniui jį pritaiko verslo sistemų kūrimo procese.

6 Rezultatai

Atlikus MDA praktinio taikymo programų sistemų kūrimo procese tyrimą, gauti šie rezultatai:

- Identifikuoti pokyčiai, kuriuos PS kūrimo procese iššaukia MDA grįstas požiūris;
- Suklasifikuotos MDA taikymo galimybės ir kliūtys skirtinguose PS kūrimo proceso etapuose.
- Identifikuotos ir pristatytos programų sistemų klasės, kurioms kurti MDA grįstas procesas gali būti sėkmingai taikomas ir turi pasisekimą, bei probleminės vietos, kur MDA grįstas požiūris yra sunkiai pritaikomas.

7 Išvados

- MDA neapibrėžia jokių specifinių programų sistemų kūrimo metodikų, nesuteikia jokių proceso gairių ir nenurodo proceso veiklų, etapų (fazių), vaidmenų ir jų atsakomybių, todėl MDA negali būti laikomas PS kūrimo procesu, o yra PS kūrimo proceso *stilius*
- MDA grįstas PS kūrimo procesas turėtų pasipildyti naujomis veiklomis ir proceso veikėjų atsakomybėmis, tačiau OMG standartai apibrėžia tik nedidelę dalį proceso veiklose naudojamų technologijų (MOF, UML, OCL, QVT ir kt.). Tuo tarpu kitos technologijos (pvz., nefunkcinių atributų, lygiagrečių procesų modeliavimas, modelių trasavimas ir t.t.) nėra standartizuotos ir todėl procese turi būti atskirai apibrėžiami ir kuriami jiems sprendimai.

- MDA požiūrio, vadovaujantis OMG standartais, taikymas programų sistemų kūrimo procese reikalauja daug resursų, pastangų ir investicijų, tuo tarpu šio taikymo nauda ir efektyvumas, skirtingų autorių, teigimu dar nėra akivaizdus.
- Dėl didelių investicijų, reikalingų adaptuojant procese MDA požiūrį, MDA taikymą gali sau leisti tik didelės įmonės, turinčioms nusistovėjusį procesą, vykdančių daug užsakymų, orientuotų į tam tikrą verslo dalykinę sritį, ir kuriančias sprendimus skirtingoms platformoms.
- MDA paradigma, grindžiama OMG standartais, yra sunkiai pritaikoma programų sistemų kūrimo procese. Reikalingi MDA sistemų kūrimo karkasai, apibrėžiantys ir į sistemų kūrimo procesą integruojantys technologijas, geriausias praktikas ir MDA idėjas, instrumentus, veiklas, metodus ir darbo produktus, ir tokiu būdu palengvinantys MDA požiūrio taikymą sistemų kūrimo procese.
- MDA požiūris didžiausią pasisekimą turi kuriant įdėtines (*angl*. embedded) sistemas, kurių platformos modelis (API) yra nedidelis ir nėra keliami dideli kokybinių charakteristikų reikalavimai.
- MDA požiūris sunkiai pritaikomas sistemų, kurioms keliami aukšti kokybiniai ir nefunkciniai reikalavimai, susiję naudojama technologine ir aparatūrine platforma, kūrimui.

8 Literatūros šaltinių sąrašas

[ALR+04] Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, Carl Landwehr “Basic Concepts and Taxonomy of Dependable and Secure Computing”, IEEE Transactions On Dependable And Secure Computing, Vol. 1, No. 1, January-March 2004.

[AMM08] „The AMMA Project home page“. Prieiga internete (2008.05.28): http://www.sciences.univ-nantes.fr/lina/atl/

[AP04] M. Alanen and I. Porres, "Change Propagation in a Model-Driven Development Tool," in Presented at WiSME part of UML 2004, 2004.

[AP05] Marcus Alanen, Ivan Porres “Model Interchange Using OMG Standards”, Proceedings of the 2005 31st EUROMICRO Conference on Software Engineering and Advanced Applications, 2005.

[ATL07] „Atlas Transformation Language“. Prieiga internete (2007.12.10): http://www.sciences.univ-nantes.fr/lina/atl/

[ATL08] „ATL presentation sheet“. Prieiga internete (2008.05.28): http://www.eclipse.org/m2m/atl/doc/ATL_PresentationSheet.pdf

[BC05] Brown A., Conallen J. An introduction to model-driven architecture. Part III: How MDA affects the iterative development process. May 2005. Prieiga internete (2009.03.27): http://www.ibm.com/developerworks/rational/library/may05/brown/index.html

[Bio07] „Biography of David John Wheeler“. Prieiga internete (2007.12.03): http://www.thocp.net/biographies/wheeler_david.htm

[Bro08] Alan W. Brown “MDA Redux: Practical Realization of Model Driven Architecture”, Seventh International Conference on Composition-Based Software Systems, 2008.

[BSM+03] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, Timothy J. Grose „Eclipse Modeling Framework: A Developer's Guide“, 2003.

[CH03] Krzysztof Czarnecki, Simon Helsen „Classification of Model Transformation Approaches“, OOPSLA’03 Workshop on Generative Techniques in the Context of Model-Driven Architecture, 2003.

[COM07] „COMET - Component and model-based development methodology“. Prieiga internete (2007.12.10): http://www.modelbased.net/comet/index.html

[Coo04] Steve Cook „Domain-Specific Modeling and Model Driven Architecture“, MDA Journal, January 2004.

[Eva04] Eric Evans „Domain-Driven Design Quickly“, 2004.

[FOR08] „Fornax Platform“. Prieiga internete (2008.05.28): http://www.fornax-platform.org

[Fow03] Martin Fowler „Platform Independent Malapropism“, 2003. Prieiga internete (2009.04.02): http://martinfowler.com/bliki/PlatformIndependentMalapropism.html

[Fow04] Martin Fowler, „Model Driven Architecture“, 2004. Prieiga internete (2009.04.02): http://martinfowler.com/bliki/ModelDrivenArchitecture.html

[FSB04] Franck Fleurey, Jim Steel, Benoit Baudry „Validation in Model-Driven Engineering: Testing Model Transformations“, First International Workshop on Model, Design and

Validation, 2004. Proceedings. 2004.

[GBP+04] Gavras, A. and Belaunde, M. and Ferreira Pires, L. and Andrade Almeida, J.P. „Towards an MDA-based development methodology for distributed applications“. In: 1st European Workshop on Model-Driven Architecture with Emphasis on Industrial Applications (MDA-IA 2004), 17-18 Mar 2004.

[Haa08] Johan de Haan “DSL and MDE, necessary assets for Model-Driven approaches”, 2008. Prieiga internete [2008.11.21]: http://www.theenterprisearchitect.eu/archive/2008/08/11/dsl-and-mde-necessary-assets-for-model-driven-approaches

[HVE+07] Arno Haase, Markus Völter, Sven Efftinge, Bernd Kolb „Introduction to openArchitectureWare 4.1.2“, 2007. Prieiga internete (2008.05.28): http://www.dsmforum.org/events/MDD-TIF07/oAW.pdf

[Joc08] Darius Jockel „Hibernate Cartridge Manual“, 2008. Prieiga internete (20008.06.12): http://de.geocities.com/darius.jockel/manual/Hibernate_Manual.html

[JS03] Juanjuan Jiang, Tarja Systä “Exploring Differences in Exchange Formats – Tool Support and Case Studies”, Proceedings of the Seventh European Conference On Software Maintenance And Reengineering, 2003

[Kru03] Philippe Kruchten “The Rational Unified Process: An Introduction, Third Edition”, 2003.

[Kru95] Philippe Kruchten “Architectural Blueprints – The “4+1” View Model of Software Architecture”, IEEE Software 12 (6), November 1995, pp. 42-50.

[KT08] Steven Kelly, Juha-Pekka Tolvanen „Domain-Specific Modeling. Enabling Full Code Generation“, 2008.

[Kur05] Kurtev, I. „Adaptability of Model Transformations“. University of Twente. Enschede: Febodruk BV., 2005.

[KWB03] Anneke Kleppe, Jos Warmer, Wim Bast „MDA Explained: The Model Driven Architecture™: Practice and Promise", 2003.

[LE04] Langlois B., Exertier D. “MDSoFa: a Model-Driven Software Factory” Thales Research & Technology France. October 2004. Prieiga internete (2009.03.27): www.softmetaware.com/oopsla2004/langlois.pdf

[LZG05] Yuehua Lin, Jing Zhang, and Jeff Gray “A Testing Framework for Model Transformations”, in Model-Driven Software Development - Research and Practice in Software Engineering. 2005.

[MA08] Parastoo Mohagheghi, Jan Aagedal “Evaluating Quality in Model-Driven Engineering”, International Workshop on Modeling in Software Engineering (MISE'07), 2007.

[MAS04] “Process Model to Engineer and Manage the MDA: Model-driven Architecture inSTrumentation, Enhancement and Refinement Approach (MASTER)”, European Software Institute, 2004. Prieiga internete (2008.11.23): http://modeldrivenarchitecture.esi.es

[Mas07] Saulius Maskeliūnas “Programų sistemų architektūra ir projektavimas”, 2007.

[MCG05] Tom Mens, Krzysztof Czarnecki, Pieter Van Gorp „A Taxonomy of Model Transformations“, 2005. Prieiga internete (2008.11.05):

http://drops.dagstuhl.de/opus/volltexte/2005/11/pdf/04101.SWM2.Paper.pdf

[MDA03] „MDA Guide Version 1.0.1“. © 2003 OMG. Prieiga internete (2007.12.08): ftp://ftp.omg.org/pub/docs/omg/03-06-01.pdf

[MDA06] „MDA Guide Revision Draft–Version 0.1.5“, OMG, 2006. Prieiga internete (2008.12.22): http://ormsc.omg.org/mda_guide_working_page.htm

[MOF02] “MOF® Specification. Version 1.4 April 2002“. Prieiga internete (2007.12.03): http://www.omg.org/mda/specs.htm#MOF. 2007 m. gruodis 3 d.

[MSU+04] Stephen J. Mellor, Kendall Scott, Axel Uhl, Dirk Weise „MDA Distilled: Principles of Model-Driven Architecture“, 2004.

[NBB+08] Brian Nolan, Barclay Brown, Laurent Balmelli, Tim Bohn, Ueli Wahli “Model Driven Systems Development with Rational Products”, IBM Redbooks, 2008.

[NN08] Vladimirs Nikulsins, Oksana Nikiforova „Adapting Software Development Process towards the Model Driven Architecture“, The Third International Conference on Software Engineering Advances, 2008.

[OAW08] „openArchitectureWare“. Prieiga internete (2008.05.28): http://www.openarchitectureware.org, http://www.eclipse.org/gmt/oaw/

[PA09] Bertrand Portier, Lee Ackerman „Model Driven Development Misperceptions and Challenges“, 2009. Prieiga internete (2009.03.12): http://www.infoq.com/articles/mdd-misperceptions-challenges

[Pig09] Pigoski, T.M. „Software Maintenance. In Swebok (Software Engineering Body of Knowledge), IEEE version 1.00, Chapter 6“. Prieiga internete (2009.04.16): http://www.swebok.org/ch6.html

[PM07] Oscar Pastor, Juan Carlos Molina „Model-Driven Architecture in Practice“, 2007.

[QVT07] „SmartQVT documentation“, 2007. Prieiga internete (2008.05.22): http://smartqvt.elibel.tm.fr/doc/fr.tm.elibel.smartqvt.doc/SmartQVT_documentation.html

[QVT08] “Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. Version 1.0”, 2008.

[SCG+05] Peter Swithinbank, Mandy Chessell, Tracy Gardner, Catherine Griffin, Jessica Man, Helen Wylie, Larry Yusuf “Patterns: Model-Driven Development Using IBM Rational Software Architect”, IBM Redbooks, 2005.

[Ski08] Cameron Skinner „DSL + UML = Pragmatic Modeling“, Skinner‘s Blog, 2008. Prieiga internete (2009.04.30): http://blogs.msdn.com/camerons/archive/2008/06/25/dsl-uml-pragmatic-modeling.aspx

[SS07] Jagadish Suryadevara, Shyamasundar R.K. “UML-based Approach to Specify Secured, Fine-grained Concurrent Access to Shared Resources”, Journal Of Object Technology, Vol.6, No.1, January-February 2007.

[Ste08] Dave Steinberg „Fundamentals of the Eclipse Modeling Framework“, 2008. Prieiga internete (2008.05.18): http://www.eclipse.org/modeling/emf/docs/presentations/EclipseCon/EclipseCon2008_309T_Fundamentals_of_EMF.pdf

[Tho04] Dave Thomas, "MDA: Revenge of the Modelers or UML Utopia?", IEEE Software, vol. 21, no. 3, pp. 15-17, May/June 2004.

[Tra05] L. Tratt, "Model transformations and tool integration," Journal of Software and Systems Modelling, vol. 4, no. 2, pp. 112-122, 2005.

[Tra08] L. Tratt, "A change propagating model transformation language," Journal of Object Technology, vol. vol. 7, no. no. 3, pp. 107-126, Mar. 2008.

[UML08] „Introduction to OMG's Unified Modeling Language™ (UML®)“. Prieiga internete (2008.05.24): http://www.omg.org/gettingstarted/what_is_uml.htm

[Vol07] Markus Völter „openArchitectureWare 4.2 Fact Sheet“, 2007. Prieiga internete (2008.05.28): http://www.eclipse.org/gmt/oaw/oAWFactSheet.pdf

[WP03] Alain Wegmann, Otto Preiss “MDA in Enterprise Architecture? The Living System Theory to the Rescue…”, Proceedings of the Seventh IEEE International Enterprise Distributed Object Computing Conference (EDOC’03), 2003.

[XDI06] OMG XMI Diagram Interchange Specification, 2006.

[XLH07] Yingfei Xiong, Dongxi Liu, Zhenjiang Hu, Haiyan Zhao, Masato Takeichi and Hong Mei „Towards Automatic Model Synchronization from Model Transformations“, Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, 2007.

[XMI07] OMG XMI specification „MOF 2.0/XMI Mapping, Version 2.1.1“, 2007.

[Zet06] C. Zetie, “MDA is DOA, Partly Thanks to SOA”, Forrester Trends Report, March 22, 2006. Pgl. [Bro08].

9 Santrumpos

ATL	Prancūzijos nacionalinio kompiuterijos mokslo instituto (*pranc.* Institut National de Recherche en Informatique et en Automatique – INRIA) sukurta modelių transformavimo kalba, skirta apibrėžti transformacijas remiantis QVT specifikacija.
CIM	Nuo algoritmizacijos nepriklausomas aukščiausio abstrakcijos laipsnio modelis (*angl.* Computation Independent Model).
COMET	Į modelius orientuotas ir užduotimis (*angl*. Use Case) grindžiamas programų sistemų kūrimo metodas.
DSL	Dalykinės srities kalba (*angl.* Domain Specific Language).
EMF	„Ecipse" platformos pagrindu sukurtas metamodeliavimo karkasas ir programinio kodo generavimo priemonė, skirta kurti kitus įrankius ir taikomąsias sistemas, grįstas struktūrizuotais duomenų modeliais.
MDA	Modeliais grįsta sistemų architektūra – OMG konsorciumo iniciatyva apibrėžti naują požiūrį į programų sistemų kūrimą remiantis modeliais ir automatizuota jų transformacija į programinį kodą.
MOF	OMG patvirtintas metamodelių kūrimo standartas.
OCL	OMG standartizuota modelių išraiškų kalba, kuria galima rašyti užklausas modelių elementams, atitikimo ir validavimo sąlygas ir pan. (*angl.* Object Constraint Language)
OMG	„Object Management Group“ konsorciumas, vienijantis daugiau, nei 500 organizacijų.
PIM	Nuo platformos nepriklausomas modelis.
PSM	Specifinis platformai modelis.
QVT	OMG konsorciumo patvirtinta modelių transformacijų kalbos specifikacija (*angl.* Queries/Views/Transformations).
RUP	Programinės įrangos kūrimo proceso karkasas, sukurtas įmonės „Rational Software“, nuo 2003 m. priklausančios IBM.
UML	Standartizuota bendros paskirties modeliavimo ir specifikacijų kūrimo kalba, naudojama programų sistemų inžinerijoje.
XMI	Standartizuotas modelių saugojimo XML dokumente formatas.