KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
PROGRAMŲ INŽINERIJOS KATEDRA

Rytis Stankevičius

Transformacijų tarp konkrečių programavimo kalbų abstrakčios sintaksės medžių ir nuo kalbos nepriklausomų abstrakčios sintaksės medžių tyrimas

Magistro darbas

Darbo vadovas
prof. dr. L. Nemuraitė

Kaunas, 2010

KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
PROGRAMŲ INŽINERIJOS KATEDRA

Rytis Stankevičius

Transformacijų tarp konkrečių programavimo kalbų abstrakčios sintaksės medžių ir nuo kalbos nepriklausomų abstrakčios sintaksės medžių tyrimas

Magistro darbas

Recenzentas
doc. dr. V. Pilkauskas
2010-05-31

Vadovas
prof. dr. L. Nemuraitė
2010-05-31

Atliko
IFM-4/2 gr. stud.
Rytis Stankevičius
2010-05-31

Kaunas, 2010

TURINYS

Summary

Investigation of transformations between abstract syntax trees of a concrete programming language and language-independent abstract syntax trees

Tools for transforming tree-like data structures are used for solving a range of different problems, such as creation of compilers, code optimizers, documentation and code generation tools, etc. But none of these existing tools seem to be specialized in transforming abstract syntax trees to a language-independent format or from it.

This master‘s thesis describes a system that takes concrete abstract syntax trees (ASTs that represent the structure of code written in a real programming language, such a C#, Java, Ruby, etc.) and transforms them into language-independent abstract syntax trees, and vice versa. Rules for mapping one AST to the other are defined in XML-based configuration files that have a simple and clear structure. The system can be used as a stand-alone tool or used as component in third party software. Both the system’s core libraries and tools are written in Java.

The purpose of the system’s investigation was to prove two things: that the system can correctly transform a given specific AST to an equivalent generic AST, and vise versa; and that the transformation execution time is within an acceptable range. Both goals were met. This led to a conclusion that the system is a suitable candidate for use by third party tools that handle code migration, refactoring, model transformation and other types of similar tasks.

Keywords: abstract syntax tree, generic abstract syntax tree, language-independent abstract syntax tree, specific abstract syntax tree, AST, transformation

1. ĮVADAS

CASE priemonės palengvina programinės įrangos kūrimo darbą leisdamos projektuoti sistemas aukštame abstrakcijos lygmenyje. Įvairių diagramų pavidalu (esybių-ryšių, funkcijų ir pan.) specifikuojamos sistemos vėliau automatiškai realizuojamos formų, ataskaitų, programavimo kalbų išeities tekstų ir kitais pavidalais: atliekama aukšto lygio sistemos specifikacijos transformacija į žemesnį, labiau detalų lygmenį. Taip sutaupoma laiko bei pastangų.

Tačiau tokios priemonės turi svarbų trūkumą – žemesnio lygmens sistemos modeliai yra detalesni ir talpina daugiau informacijos, todėl pakeitimus juose sunku sinchronizuoti su aukštesnių lygmenų sistemos modeliais. Tai sukelia ypač didelių nepatogumų, jei reikia rankiniu būdų papildyti sugeneruotus žemesnio lygmens modelius, o vėliau prireikia juos pergeneruoti pakeitus aukštesniojo lygmens modelius.

Siekiant išspręsti informacijos praradimo problemą, pradėtos nagrinėti žemiausio lygmens modelių – programinio kodo – transformacijos. Egzistuojančių kalbų analizės metu iškelta idėja sudaryti bendrą kalbos konstrukcijų aibę, nepriklausančią jokiai konkrečiai programavimo kalbai. Tokios aibės pagrindu būtų bandoma sukurti bendrinio AST medžio duomenų struktūrą, gebančią talpint įvairių konkrečių programavimo kalbų (pavyzdžiui, Java, C#, Ruby ir pan.) AST medžius. Atlikus rinkos analizę šios idėjos atsisakyta, nes buvo rasta jau egzistuojanti bendrinio AST specifikacija.

Rastosios specifikacijos pagrindu buvo sukurta kodo biblioteka, gebanti talpinti bendrinius AST medžius, juos saugoti į failus, užkrauti iš jų, ir atlikti su jais kitus veiksmus. Biblioteką sukūrė Rytis Stankevičius (šio magistro darbo autorius) bei Justas Jokubauskas (darbo autoriaus kolega). Šios bibliotekos pagrindu buvo sukurtos kitos dvi bibliotekos – AST transformavimo ir AST pertvarkymo.

Pirmoji biblioteka skirta atlikti konkrečių programavimo kalbų AST transformacijas į bendrinius AST ir atvirkščiai. Pagrindinis bibliotekos privalumas yra galimybė aprašyti medžių transformacijas taisyklių, talpinamų išoriniuose failuose, pavidalu. Bibliotekos autorius – Rytis Stankevičius. Ši biblioteka (toliau šiame darbe vadinama tiesiog „sistema“) yra šio darbo **tyrimo objektas**, o **tyrimo sritis** yra AST medžių transformavimo algoritmai ir programinė įranga.

Antroji biblioteka skirta atlikti papildomiems bendrinių AST medžių pertvarkymo veiksmams (angl. „refactoring“), taip siekiant geriau struktūrizuoti programų sandarą ir suvienodinti bendrinius AST, sudarytus iš skirtingų konkrečių programavimo kalbų AST. Bibliotekos autorius – Justas Jokubauskas. Ši biblioteka šiame darbe detaliau nenagrinėjama.

Šio darbo **sprendžiama problema** – kokybiškos AST transformavimo sistemos sukūrimas. Tokia sistema galėtų tapti įrankių, leidžiančių konvertuoti vienos programavimo kalbos kodą į kitos kalbos kodą, pagrindu. Tokių įrankių pritaikomumo galimybės labai plačios – liktinių sistemų kodo renovavimas, skirtingomis kalbomis parašyto kodo sujungimas, sistemų portatyvumo galimybės ir t.t. Kitos AST transformavimo sistemos panaudojimo galimybės – kodo pertvarkymo įrankių, specifinių srities kalbų kompiliavimo įrankių bei modelių transformavimo įrankių kūrimas.

Šio **darbo tikslai** – įrodyti, kad AST transformavimo sistema geba korektiškai ir pakankamai greitai transformuoti pasirinktos programavimo kalbos AST į bendrinį AST, ir atgal.

Iškelti tokie **darbo uždaviniai**:

- Ištirti rinkoje egzistuojančių priemonių, panašių į kurtąją sistemą, bendrumus ir skirtumus;
- Surasti bendrumus tarp AST transformavimo sistemos ir išanalizuotų priemonių;
- Identifikuoti ir išspręsti specifines sistemos problemines sritis;
- Suprojektuoti ir sukurti sistemos prototipą;
- Išanalizuoti prototipo privalumus bei trūkumus;
- Pateikti sistemos prototipo patobulinimus ir juos įgyvendinti;
- Ištirti naujosios sistemos veikimo korektiškumą ir greitaveiką.

Tam, kad būtų galima įvertinti, ar patobulintoji AST transformavimo sistemos versija tenkina iškeltuosius tikslus, nustatyti tokie sistemos **kokybės kriterijai**:

- *Sistemos atliekamų transformacijų korektiškumas*. Jei sistemai pateikiamas teisingai suformuotas konkretus AST medis ir medžio transformavimo aprašas, privalo būti sukurta pradinį medį atitinkantis bendrinis AST medis. Tas pats reikalavimas galioja ir atvirkštiniai transformacijai.
- *Sistemos greitaveika*. Ji suvokiama dviem aspektais: ar sistema išvis veikia pakankami greitai su nedideliais duomenų kiekiais ir ar jos spartos priklausomybės nuo duomenų kiekio charakteristika yra pakankamai gera. Jei duomenų kiekis padidinamas nežymia dalimi, sistema neturi sulėtėti daugybę kartų.

Gauti **darbo rezultatai**:

- Ištirta AST transformavimo probleminės sritis;
- Palyginti rinkoje egzistuojantys panašūs sprendimai;

- Suprojektuotas ir sukurtas sistemos prototipas;
- Suprojektuota ir sukurta nauja sistemos versija;
- Įsitikinta, jog naujoji sistemos versija veikia gerai ir greitai.

Magistro **darbą sudaro** devyni skyriai.

Antrajame skyriuje pateikta literatūros šaltinių analizė. Joje apžvelgiamos penkios į sukurtąją sistemą panašios medžio struktūrų transformavimo priemonės – Stratego/XT, TXL, XSTL, Alchemist ir atributinės gramatikos. Priemonės palyginamos, išryškinami jų panašumai, išskiriamos esminės dalys. Apžvelgus šias priemones aptariamos sistemos esminės probleminės sritys ir pateikiami jų sprendimai remiantis literatūros šaltiniais. Sprendimų pasirinkimo priežastys aptariamos ir argumentuojamos.

Trečiajame skyriuje aptariamas sistemos prototipui sudarytas projektas. Jame atliekama sistemos vartotojų analizė, pateikiamas panaudojimo atvejų modelis, specifikuojami reikalavimai sistemai, sudaromas architektūros statinis ir dinaminis modeliai. Galiausiai aprašomas testavimo planas, iškeliantis testavimo tikslus, apžvelgiantis skirtingus testavimo būdus bei pateikiantis testų pavyzdžius.

Ketvirtajame skyriuje įvertinamas sistemos prototipas. Įvertinimas atliekamas tiek teoriniu aspektu pagal viename iš literatūros šaltinių pateiktus kriterijus, tiek remiantis praktiniu prototipo naudojimu. Atskleidžiami prototipo trūkumai ir pateikiami architektūriniai, transformavimo algoritmų bei duomenų failų formatų pakeitimų sprendimai. Kiekvienas iš jų aptariamas detaliau.

Penktajame skyriuje įvertinama naujoji sistemos versija. Pagal iškeltus tikslus vykdomas dviejų etapų eksperimentas: pirmasis etapas tikrina sistemos atliekamų transformacijų veikimo korektiškumą, o antrasis – greitaveiką. Tiriant greitaveiką atkreipiamas dėmesys tiek į absoliučią sistemos spartą, tiek į spartos kitimo dėsningumus priklausomai nuo pateiktų duomenų kiekių.

Šeštajame darbe pateikiamos atlikto darbo išvados.

Septintajame skyriuje pateikiamas sąrašas literatūros, naudotos panašių priemonių ir sprendimų analizei, terminų apibrėžimui bei kitiems tikslams.

Aštuntajame skyriuje pateikiamas trumpas sąrašas naudojamų terminų ir santrumpų.

Devintajame skyriuje pateiktas priedų sąrašas. Jame pateikti programų išeities tekstas, AST medžių aprašai bei AST transformacijų aprašai.

2. AST TRANSFORMAVIMO SISTEMOS ANALIZĖ

2.1. AST transformavimo sistemos apžvalga

2.1.1. Sistemos apibrėžimas ir paskirtis

AST transformavimo sistemos tikslas – suteikti galimybę paversti bet kokios programavimo kalbos AST duomenų struktūrą į bendrinę AST duomenų struktūrą ir šią bendrinę struktūra konvertuoti į bet kokios kitos programavimo kalbos AST struktūrą. AST medžių struktūros gaunamos kai kokios nors programavimo kalbos tekstas išanalizuojamas gramatiškai **[1]**. Tai programos sandarą apibūdinanti duomenų struktūra, kurios dėka galima generuoti programos kodą ir/ar atlikti programos pertvarkymus (angl. „refactoring“, **[2]**).

Šiame darbe nagrinėjamoji sistema suvokiama dvejopai:

- *Siauraja prasme* sistema vadinama programinio kodo biblioteka, atliekanti minėtąsias AST medžių transformacijas pagal konfigūracijos failuose aprašytas medžių transformavimo taisykles. Transformavimo taisyklės rašomos kiekvienai konkrečių AST medžių rūšiai atskirai bent po du kartus: pirmasis transformacijas aprašas skirtas transformuoti konkretų AST į bendrinį, antrasis – atvirkščiai.
- *Plačiąją prasme* sistema – tai ta pati kodo biblioteka bei papildomų įrankių rinkinys, skirtas dirbti su šia biblioteka. Pirmasis įrankis skirtas atlikti transformacijas, o antrasis – kurti transformacijos aprašo failus. Įrankiai apžvelgiami detaliau sistemos projekto apraše.

2.1.2. Sistemos aktualumas ir tikslingumas

Atliekant tiriamąjį darbą buvo nagrinėjamos šiuolaikinės priemonės, leidžiančios greitai ir efektyviai kurti kompiuterines sistemas (tiriamąjį darbą atliko doktorantas Linas Ablonskis, AST transformavimo sistemos užsakovas). Rankinį sistemų kūrimo procesą kai kuriose srityse iš dalies pakeitė CASE priemonių atsiradimas.

CASE priemonės leidžia kurti labiau patikimas ir mažiau žmogiškųjų bei laiko resursų reikalaujančias sistemas **[3]**. Šios priemonės suteikia galimybę įvairių diagramų pavidalu (esybių-ryšių, funkcijų ir pan.) specifikuoti sistemas, kurios vėliau automatiškai realizuojamos formų, ataskaitų, programavimo kalbų išeities tekstų ir kitais pavidalais. Tačiau labai dažnai neužtenka vien automatinio generavimo galimybių, nes sistemos kodą tenka koreguoti ar papildyti rankiniu būdu. O jei sistemos specifikavimo diagramose randama klaidų, kodas generuojamas iš naujo, ir todėl

dingsta rankiniu būdu papildytas kodas. Dėl to nukenčia sistemos patikimumas, eikvojami resursai ir tiesiog atliekama daug perteklinio darbo.

Šiai problemai išspręsti reikalingas būdas išsaugoti rankiniu būdu atliktą darbą net ir tuomet, kai iš naujo atliekamas kodo generavimas. Taigi, reikalingas abipusis ryšys tarp skirtingų sistemos lygių. Tiksliau, reikalingas ryšys tarp skirtingų sistemos modelių.

Sistemos modeliai yra modeliais paremto kūrimo **[4]** pagrindas. Tai tokia sistemų kūrimo metodika, kuri siekia sistemą išreikšti kaip tam tikrą srities problemos abstrakciją. Modelių būna įvairių tipų. Kiekvienas jų skiriasi teikiamos informacijos detalumu ir forma. Aukščiausio lygio modeliai remiasi panaudos atvejų bei funkcinių reikalavimų padengimu. Su šio lygio modeliais dirbama siekiant sudaryti sistemos vaizdą iš kliento perspektyvos. Žemesnio lygio modeliai išreiškia aukštesnio lygio modelių konstrukcijų realizavimą programinio kodo ar panašiu pavidalu.

Skirtingų modelių atskyrimas leidžia skirtingos profesijos atstovams dirbti prie jiems rūpimų sistemos dalių **[5]**. Tačiau čia kyla problema dėl anksčiau minėtųjų skirtingo abstrakcijos lygio modelių, nes aukštesnės abstrakcijos modeliai savyje turi mažiau informacijos, nei žemesnio lygio modeliai.

Tam, kad būtų galima spręsti problemą dėl skirtingo abstrakcijos lygio modelių transformavimo neprarandant informacijos, visų pirma svarbu išnagrinėti transformacijas, atliekamas tame pačiame lygmenyje, pavyzdžiui, programinio kodo lygmenyje. Taip pradėtos nagrinėti skirtingų programavimo kalbų kodo bendrosios savybės.

Atliekant programavimo kalbų lyginimą, buvo iškelta idėja, jog galima sudaryti bendrą kalbos konstrukcijų aibę, nepriklausančią jokiai konkrečiai programavimo kalbai. Tokios konstrukcijos egzistavimas leistų konkrečiomis programavimo kalbomis išreikštas programas konvertuoti į bendrą pavidalą. Kadangi programavimo kalbų kompiliatoriai kodo generavimo proceso metu informaciją talpina AST medžių pavidalu, todėl nuspręsta, jog bendrinė duomenų struktūra taip pat būtų AST medis, tačiau nespecializuotas jokiai programavimo kalbai.

Taigi, galiausiai kilo klausimas, kaip galima atlikti skirtingo kodo transformacijas į/iš bendrinės AST duomenų struktūros. Remiantis egzistuojančių AST medžių generavimo įrankių pavyzdžiu, nuspręsta bandyti sukurti programinio kodo biblioteką, kuri sugebėtų transformuoti AST struktūras naudodamasi transformacijas aprašančiomis specializuotomis transformacijų kalbomis. Kiekvienai konkrečiai transformacijai numatyta turėti atskirą transformacijos aprašą, esantį konfigūravimo failuose.

2.1.3. Praktinio sistemos pritaikymo galimybės

Kadangi sistemos pagrindas yra ne konkreti programa, o kodo biblioteka (pagalbinės programos kurtos šiai bibliotekai išbandyti, tačiau pagrindas vis dėl to yra pati biblioteka), tiesiogiai ji neduoda didelės praktinės naudos. Tačiau ši biblioteka gali tapti pagrindu kuriant daugybę skirtingos paskirties produktų. Šiuo metu numatomi tokie produktai, potencialiai galintys būti sukurti bibliotekos pagrindu:

1) *Programų kodo pertvarkymo įrankiai,* palaikantys daugybę programavimo kalbų. Jie leistų atlikti labiausiai žinomus ir geriausiai ištirtus programų kodo pertvarkymus (kintamųjų ir metodų pervadinimus, metodų iškėlimus, sąlyginių išraiškų išskaidymus ir pan.). Tokie įrankiai veikiausiai būtų integruoti į konkrečias programavimo aplinkas (pvz.: Visual Studio, Eclipse ir t.t.) kaip įskiepiai. Tačiau svarbu paminėti keletą tokių įrankių apribojimų. Pakankamai išvystytos, daugybę programavimo kalbų naudojančios programavimo aplinkos yra retos. Be to, tokio tipo bendriniai kodo pertvarkymo įrankiai veikiausiai nesugebėtų atlikti rečiau pasitaikančių kodo pertvarkymų, kurie būdingi tik konkrečioms kalboms.

2) *Daugybę programavimo kalbų palaikantys kodo transformatoriai*, kurie sugebėtų iš vienos konkrečios programavimo kalbos sugeneruoti kitos konkrečios programavimo kalbos kodą. Tai suteiktų galimybę kurti dideles sistemas atskiroms komandoms, naudojančioms skirtingas programavimo kalbas (panašią galimybę suteikia Visual Studio programavimo aplinka, naudojanti keletą kalbų .NET platformos pagrindu, tačiau ji nepalaiko daugybės kitų, ne .NET platformai skirtų, kalbų). Tokie transformatoriai taip pat leistų modernizuoti liktines sistemas, kurios vis dar pakankamai gerai gali atlikti savo funkcijas.

3) *Specifinių programavimo kalbų kūrimo ir kompiliavimo įrankiai* **[6]**. Jie leistų greitai kurti konkrečiai probleminiai sričiai skirtas specializuotas programavimo kalbas bei kompiliuotų jomis parašytą kodą. Tokie įrankiai galėtų naudoti jau egzistuojančių programavimo kalbų kompiliatoriais, prieš tai specifinės kalbos kodą transformavę į egzistuojančios kalbos kodą. Taip būtų sutaupoma daug pastangų ir laiko, nes nereiktų kurti specializuotų kompiliatorių (panašiu principu veikia daugybė Eiffel programavimo kalbos kompiliatorių, verčiančių Eiffel kalbos kodą į C kalbos kodą **[7]**). Svarbiausia, jog būtų galima naudotis net keliais egzistuojančių kalbų kompiliatoriais.

4) *Sistemos modelių transformavimo įrankiai.* Šie įrankiai leistų kurti įvairiais sistemas skirtinguose abstrakcijos lygmenyse. Vieną sistemą galėtų kurti skirtingi žmonės (pvz.: veiklos srities ekspertas ir programuotojas) skirtinguose lygmenyse: veiklos srities ekspertas galėtų kurti sistemos modelį paremtą vartotojų reikalavimais, o programuotojas sukurtu norimą sistemos

funkcionalumą realizuojantį kodą. Atliekant pakeitimus viename lygmenyje, jie turėtų būti atliekami ir kitame lygmenyje. Labiausiai abstraktus lygmuo galėtų netgi tapti kuriamų sistemų dokumentavimo priemone ir leistų atsisakyti atskirų specifikacijos dokumentų. Tačiau tokių sistemos modeliavimo įrankių kūrimas yra didelis iššūkis, nes, siekiant suteikti galimybę nuolatos keisti sistemos modelį skirtinguose lygmenyse, būtina išsaugoti paties žemiausio lygmens detalumo informaciją. Tikėtina, jog tokio tipo uždavinys nėra išsprendžiamas.

2.2. Panašios programinės įrangos ir priemonių analizė

2.2.1. Transformacijų kūrimo karkasas Stratego/XT

Stratego/XT yra karkasas skirtas platų programų transformacijų spektrą palaikančių transformacijų sistemų kūrimui **[8]**. Karkasą sudaro Stratego transformacijų kalba bei XT transformavimo įrankių kolekcija. Stratego paremtas medžių elementų perrašymu pasitelkiant programuojamas perrašymo strategijas. XT įrankiai suteikia priemones transformacijų sistemų apdorojimui, įskaitant gramatiniam kodo nagrinėjimui bei žmogui įskaitomo kodo generavimui. Karkasas apima visą kūrimo procesą: nuo transformacijų specifikacijos iki transformacijų sistemų kompozicijos.

Pradinis Stratego pritaikymas buvo programų optimizatorių specifikavimas. Vėliau Stratego tapo pritaikoma daugybėje kitų sričių: kompiliatorių kūrime, programų generatorių kūrime, programų migravime, dokumentacijos generavime ir pan.

Standartinė Stratego kalbos realizacija iš Stratego kalbos programų sukuria C programas. Sukurtosios programos yra priklausomos nuo ATerm bibliotekos, kuri skirta medžio struktūrų talpinimui ir apdorojimui, bei specifinio Stratego komponento, naudojamo programų paleidimo metu. Standartinę Stratego biblioteką sudaro gausybė bendrinių ir konkrečių duomenų tipų medžio elementų perrašymo taisyklių ir medžių perėjimo strategijų, kurios skirtos pakartotiniam panaudojimui.

Stratego kalba sukurtas sistemas galima analizuoti keturiais pagrindiniais abstrakcijos lygmenimis:

1) *Transformavimo taisyklių.* Jos aprašo kaip tam tikrus pradinius AST medžių elementus transformuoti į naujus medžių elementus. Taisyklės gali būti aprašomos tiek abstrakčios, tiek konkrečios sintaksės pagrindu. Aprašymai konkrečia sintakse įprastai yra lengviau skaitomi.

2) *Transformavimo strategijų.* Jas sudaro taisyklių rinkiniai, kurie nusako medžių elementų aplankymo eiliškumą. Vykdant tas pačias transformacijas su skirtingomis transformacijų strategijomis įprastai gaunami skirtingi transformuotieji medžiai.
3) *Transformavimo įrankių.* Tai nepriklausomi komponentai, atliekantys medžių elementų transformacijas pagal tam tikrus transformacijų taisyklių ir perėjimo strategijų rinkinius. Šie komponentai gali būti iškviečiami iš komandinės eilutės, kad transformuotų nurodytuosius pradinius medžių elementus į galutinius medžių elementus. Komponentai pasižymi geru pakartotiniu panaudojamu, nes vieno komponento rezultatus gali naudotis kitas komponentas (duomenys ir rezultatai pateikiami ATerm formatu, aprašančiu medžius).
4) *Transformavimo sistemų.* Tai įrankių rinkiniai, kurie atlieka pradinio programinio kodo transformacijas į galutinį kodą. Įprastai sistemas sudaro trys esminės dalys – gramatikos analizatoriai, transformavimo komponentai ir kodo generatoriai. Pirmieji iš programos kodo sukuria medžių struktūras, paskesnieji atlieka įvairius transformacijų žingsnius, o paskutinieji iš medžių struktūrų sugeneruoja žmogui įskaitomą programos kodą.

Toliau pateiktas Stratego programos pavyzdys (1 pav.). Ją sudaro trys pagrindinės dalys: signatūra (angl. „signature“), transformacijos taisyklės (angl. „rules“) ir strategijos (angl. „strategies“). Signatūra aprašo medžių elementų sudarymo taisykles pagal egzistuojančius medžių elementus, o transformacijų taisyklių ir strategijų prasmė atitinka anksčiau aprašytąją.

```
module Example
imports lib
signature
  constructors
    Plus  : Exp * Exp -> Exp
    Minus : Exp * Exp -> Exp
    Times : Exp * Exp -> Exp
    Nat   : String -> Exp
    Id    : String -> Exp
    Real  : String -> Exp

rules
  Inc:
    Nat("0") -> Nat("1")

strategies
  main = io-wrap(topdown(try(Inc)))
```

1 pav. Stratego programos pavyzdys

2.2.2. Medžių transformavimo kalba TXL

TXL (angl. „Tree Transformation Language“ – medžių transformavimo kalba) – tai programavimo kalba ir greitojo prototipų kūrimo sistema, specialiai skirta atlikti taisyklėmis paremtas transformacijas iš pradinio programinio kodo į galutinį **[9]**. Pradiniuose kalbos gyvavimo etapuose ji buvo naudojama kaip įrankis programavimo kalbų dialektams tirti, tačiau vėliau kalba evoliucionavo į bendros paskirties kodo transformavimo sistemą. Per kalbos gyvavimo laikotarpį ji buvo išbandyta gausybėje įvairaus tipo architektūros rekonstravimo, pakartotinės inžinerijos, analizės ir automatinio perprogramavimo uždavinių. Pavyzdžiui, kalba buvo pasitelkta apdorojant milijardus Cobol, PL/1 ir RPG programavimo kalbomis parašyto kodo eilučių. Kodas buvo tvarkomas tam, kad būtų išvengta garsiosios 2000-ųjų metų problemos.

Kalbą sudaro dvi dalys: nuo konteksto nepriklausantis pradinės programavimo kalbos struktūros (gramatikos) aprašas, išreikštas EBNF formoje (2 pav.), bei kalbos elementų transformavimo taisyklių sąrašas (3 pav.).

```
define program
    [expression]
end define

define expression
    [term]
  | [expression] + [term]
  | [expression] - [term]
end define

define term
    [primary]
  | [term] * [primary]
  | [term] / [primary]
end define

define primary
    [number]
  | ( [expression] )
end define
```

2 pav. Programavimo kalbos gramatikos aprašas, parašytas TXL kalba

```
rule vectorizeScalarAssignments
   replace [repeat statement]
       V1 [variable] := E1 [expression];
       V2 [variable] := E2 [expression];
       RestOfScope [repeat statement]

   where not
       E2 [references V1]
   where not
       E1 [references V2]

   by
       < V1,V2 > := < E1,E2 > ;
       RestOfScope
end rule
```

3 pav. Programavimo kalbos transformavimo taisyklės aprašas, parašytas TXL kalba

Pateiktajame pavyzdyje (3 pav.) aprašyta programavimo kalbos, panašios į Pascal, skaliarinių priskyrimo sakinių pavertimo į vektorinius priskyrimo sakinius taisyklė. Taisyklė nurodo, jog radus paprastus priskyrimo sakinius juos reikia paversti į vieną vektorinį, jei tenkinami atitinkami semantiniai apribojimai, t.y. jei priskyrimo įėjimai ir išėjimai nėra tarpusavyje susaistyti neleistinomis nuorodomis (arba rodyklėmis, jei kalba semantine prasme panaši į C++).

TXL kalba lengva naudotis atliekant ganėtinai nesudėtingas transformacijas **[10]**. Tačiau transformacijos gali būti atliekamos tik į vieną pusę, t.y. nėra automatinio atvirkštinių transformacijų generavimo būdo. Transformuojamų medžių perėjimų valdymas taip pat ganėtinai paprastas ir sudėtingesnėse situacijose stokoja lankstumo. Be to, lengva atlikti tik lokalias transformacijas, o globalias atlikti yra arba labai sudėtinga, arba neįmanoma. Čia globali transformacija suvokiama kaip transformacija, kuriai reikalingas globalus kontekstas. Pavyzdžiui, lokali transformacija yra

minėtasis skaliarinių priskyrimo sakinių vektorizavimas, o globali – skirtingose pradinio programos kodo vietose iškviečiamos funkcijos pakeitimas jos kūno turiniu (angl. „inlining“).

2.2.3. XSLT transformacijų kalba

[11] XSLT (angl. „Extensive Stylesheet Language Transformations“ – išplėstinio formatavimo kalbos transformacijos) – tai XML formatu paremta transformacijų kalba, skirta transformuoti vienus XML dokumentus į kitus. Originalus dokumentas nėra keičiamas, o tiesiog iš senojo kuriamas naujas. Įprastai XSLT naudojama konvertuojant XML duomenis į HTML ar XHTML dokumentus, skirtus vaizdavimui naršyklėse.

XSLT procesorius (programa, atliekanti XSL transformacijas) naudoja du įėjimo dokumentus – pradinį XML dokumentą ir XSLT formatavimo dokumentą. Pradinis dokumentas apdorojamas pagal XSLT dokumento taisykles, ir gaunamas galutinis XML dokumentas. Transformacijos pavyzdys pateiktas toliau esančiuose paveikslėliuose (4 pav. – pradinis dokumentas, 5 pav. – transformacijos aprašas, 6 pav. – galutinis dokumentas).

```
<?xml version="1.0" ?>
<persons>
  <person username="JS1">
    <name>John</name>
    <family-name>Smith</family-name>
  </person>
  <person username="MI1">
    <name>Morka</name>
    <family-name>Ismincius</family-name>
  </person>
</persons>
```

4 pav. Pavyzdinis XML dokumentas, skirtas apdoroti atliekant XSL transformaciją

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:output method="xml" indent="yes"/>

  <xsl:template match="/persons">
    <root>
      <xsl:apply-templates select="person"/>
    </root>
  </xsl:template>

  <xsl:template match="person">
    <name username="{@username}">
      <xsl:value-of select="name" />
    </name>
  </xsl:template>

</xsl:stylesheet>
```

5 pav. Pavyzdinis XSL transformacijos dokumentas

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
  <name username="JS1">John</name>
  <name username="MI1">Morka</name>
</root>
```

6 pav. Pavyzdinis XML dokumentas, gautas po XSL transformacijos

XSLT pagrindas yra XPath užklausų kalba, kuri pradiniame dokumente ieško tam tikrų elementų sekų (angl. „patterns“). Rastų sekos elementų reikšmės gali būti panaudotos naujojo dokumento kūrime.

Apdorojant pradinį dokumentą iš eilės bandoma pritaikyti atitinkamas formatavimo taisykles, pradedant nuo šakninio XML dokumento elemento. Jei pradiniame dokumente aptinkama su tam tikra taisykle susieta XPath kalbos seka, tuomet galiniame dokumente generuojami atitinkami XML dokumento elementai (angl. „nodes“) arba toliau bandoma pritaikyti taisykles naujai rastiems elementams.

Vienas iš XSLT trūkumų yra rekursinio gilesnio lygmens elementų lankymas: elementų lankymo eilės tvarka nustatoma tiesiogiai pačioje transformacijoje, todėl nėra atskyrimo tarp atskirų transformacijos taisyklių ir elementų lankymo strategijų **[10]**. Kita vertus, šį XSLT trūkumą iš dalies atsveria plačios XPath sekų radimo kalbos galimybės, nes būtent šios sekos nurodo, kokiems elementams taikyti vienokias ar kitokias transformacijų taisykles. Taip pat XSLT transformacijų stiprybė yra galimybė perdavinėti kintamuosius iš vienos transformacijos taisyklės į kitą bei galimybė naudoti globalius kintamuosius. Šios savybės leidžia panaudoti transformacijas, kurios sugeba pasiekti pradinio dokumento informaciją iš konteksto, o ne tik iš transformacijoje dalyvaujančių pradinių elementų.

2.2.4. Transformacijų kūrimo aplinka Alchemist

Alchemist yra bendrosios paskirties transformacijų kūrimo aplinka, kuri palaiko transformacijų specifikavimą, generavimą ir vykdymą **[12]**. Alchemist leidžia kurti abstrakčias transformacijų specifikacijas naudojant langais paremtą grafinę sąsają ir palaiko transformacijų vykdomojo kodo generavimą ir kompiliavimą. Kitaip nei įprasti kompiliatoriai, Alchemist skirtas automatizuoti transformacijų kūrimą tarp dviejų sudėtingų vaizdavimo formatų ir todėl yra itin tinkamas kuriant transformacijas tarp duomenų bazių įrankių, CASE įrankių, grafinių redaktorių ir teksto formatavimo programų.

Alchemist transformacijų mechanizmą sudaro trys pagrindiniai elementai: pradinės programavimo kalbos gramatikos aprašas, galutinės programavimo kalbos gramatikos aprašas ir gramatikų susiejimo atvaizdas (angl. „map“). Gramatikų susiejimo atvaizdas išreikštas vadinamąja TT-gramatika, kuri nurodo sąryšius tarp pradinės ir galutinės gramatikos medžių. Dėl gramatikų susiejimo struktūros nesudėtinga sukurti atvirkštines transformacijas, nes abi programavimo kalbos aprašomos tokiu pačiu formatu. Žemiau pateiktas transformacijos pavyzdys, kuriame pateikta esybių

aprašo kalba bei jos transformacija į duomenų bazės modelio aprašo kalbą (7 pav. – pradinės kalbos gramatikos aprašas, 8 pav. – galutinės kalbos gramatikos aprašas, 9 pav. – susiejimo atvaizdas).

$$
\begin{array}{lcl}
ER & \rightarrow & Ss \\
Ss & \rightarrow & S\ Ss \\
Ss & \rightarrow & S \\
S & \rightarrow & Entity \\
Entity & \rightarrow & \text{“ENTITY” “(” } Name \text{ “)”} \\
Name & \rightarrow & \text{ID}
\end{array}
$$

7 pav. Pavyzdinis pradinės kalbos gramatikos aprašas, skirtas Alchemist transformavimo aplinkai

$$
\begin{array}{lcl}
Relational\_db & \rightarrow & Schemes \\
Schemes & \rightarrow & \epsilon \\
Schemes & \rightarrow & Scheme\ Schemes \\
Scheme & \rightarrow & Name \text{ “(” } Attributes \text{ “)”} \\
Name & \rightarrow & \text{IDENTIFIER}
\end{array}
$$

8 pav. Pavyzdinis galutinės kalbos gramatikos aprašas, skirtas Alchemist transformavimo aplinkai

$$
\begin{array}{lcl}
ER.Relational\_db & \rightarrow & Ss.Schemes \\
Ss_1.Schemes & \rightarrow & Entity.Scheme \\
 & & Ss_2.Schemes \\
Ss.Schemes & \rightarrow & Entity.Scheme \\
 & & Entity.Schemes \\
Entity.Schemes & \rightarrow & \epsilon \\
Entity.Scheme & \rightarrow & Name.Name \text{ “(”} \\
 & & Entity.Attributes \text{ “)”} \\
Name.Name & \rightarrow & \text{IDENTIFIER.ID}
\end{array}
$$

9 pav. Pavyzdinis transformacijos susiejimo atvaizdo aprašas, skirtas Alchemist transformavimo aplinkai

Transformacijos metu pradinės kalbos gramatikos elementų perėjimai atliekami tiesiogiai, t.y. elementų lankymo eiliškumo strategija yra standartiškai nustatyta ir negali būti keičiama programiniu būdu **[10]**. Beje, transformacijos stokojo funkcionalumo, leidžiančio vykdyti užklausas su pradinės kalbos elementais, todėl transformacijų galimybės yra ganėtinai ribotos.

2.2.5. Atributinės gramatikos

Atributinių gramatikų pagrindas yra AST medžių elementai **[10]**. Kiekvienam iš jų galima priskirti po tam tikrą papildomą informaciją nešantį atributą, kurio reikšmė gali būti suskaičiuojama pasitelkiant informaciją iš medžio elemento vaikinių elementų, tėvinių elementų ar kitų atributų.

Atributų reikšmės gali būti tiek paprasto tipo (pavyzdžiui, sveikasis skaičius), tiek ir sudėtingo (pavyzdžiui, pagal tam tikras transformuotas pradinis AST).

Atributai skirstomi į dvi grupes: *sintetinius* ir *paveldėtuosius* **[13]**. Sintetinių atributų reikšmės apskaičiuojamos pagal atributų reikšmių įvertinimo taisykles (jos nusako atributų reikšmių apskaičiavimo algoritmą ir yra apibrėžiamos kuriant gramatiką) ir taip pat gali naudoti paveldėtųjų atributų reikšmes. Paveldėtieji atributai perduodami į vaikinius medžio elementus iš tėvinių. Įprastai sintetiniai atributai naudojami perduoti semantiniai informacijai aukštyn iš vaikinių elementų į tėvinius, o paveldėtieji – žemyn ir į gretimus medžio elementus.

AST elementų perėjimų strategijos valdymas yra ganėtinai savitas kai kalbama apie atributines gramatikas **[10]**. Strategijos nėra atskirtos nuo transformacijų, tačiau yra nesunkiai valdomos. Kokia eilės tvarka bus aplankyti tam tikri medžio atributai priklauso nuo pačių atributų skaičiavimo taisyklių.

2.2.6. Transformavimo priemonių palyginimas

2.2.6.1. Palyginimo kriterijai

Šaltinis **[10]** išskiria tokius transformavimo priemonių palyginimo kriterijus:

- *Naudojamos sintaksės tipas.* Įvairūs įrankiai leidžia nurodyti skirtingo tipo sintakse aprašytas elementų radimo sekas. Sintaksė gali būti konkreti arba abstrakti. Įprastai konkrečia sintakse aprašytos sekos yra lengviau skaitomos.
- *Medžių elementų lankymo strategijos.* Skirtingi įrankiai suteikia skirtingą strategijų valdymo laipsnį. Vienuose elementų lankymas yra griežtai nusakytas ir negali būti keičiamas, o kituose strategijos gali būti nurodomos atskirai nuo transformacijų taisyklių. Galimybė nurodyti lankymo strategijas lemia galutinį transformacijos rezultatą. Beje, jei strategijos pilnai atskirtos nuo transformavimo taisyklių, jos įgyja aukštesnį pakartotinio panaudojimo laipsnį.
- *Užklausos.* Tai mechanizmas, skirtas išrinkti duomenis iš pradinių medžių pagal tam tikrus elementų struktūros ir reikšmių kriterijus.
- *Duomenų gavimas.* Tai mechanizmas, panašus į užklausas, tačiau orientuotas į globalių pradinio medžio duomenų rinkimą. Įprastai duomenų gavimas būna realizuotas lankant pradinius medžius pagal tam tikras taisykles ir pamažu akumuliuojant duomenis su kiekvieno reikiamo elemento aplankymu.

2.2.6.2. Palyginimų lentelė

1 lentelė. Transformavimo priemonių palyginimo lentelė

Funkcionalumas / Priemonė	Abstrakčios sintaksės palaikymas	Konkrečios sintaksės palaikymas	Lankymo strategijų palaikymas	Užklausų palaikymas	Duomenų gavimo palaikymas
Stratego/XT	Puikus	Puikus	Puikus	Geras	Puikus
TXL	Puikus	Puikus	Prastas	Prastas	Prastas
XSLT	Puikus	Nėra	Geras	Puikus	Puikus
Alchemist	Puikus	Nėra	Prastas	Nėra	Nėra
Atributinės gramatikos	Puikus	Nėra	Geras	Prastas	Puikus

2.3. Esminių uždavinio problemų analizė

Atlikus pasaulyje egzistuojančių panašių transformavimo priemonių analizę ir projektuojant AST medžių transformavimo sistemą, teko susidurti su tokiais klausimais:

- ***Koks turėtų būti konkretaus AST formatas ir kaip turėtų būti kuriami konkretūs AST?*** Konkretaus AST formatas turi būti tinkamas pasirinktų programavimo kalbų elementų, aprašytų tam tikra kalbų gramatikos aprašo kalba (pavyzdžiui, BNF arba EBNF), talpinimui. Taip pat svarbu turėti galimybę kurti konkrečius AST (pačių konkrečių AST kūrimas yra už šio magistro darbo ribų, tačiau siekiant ištestuoti kuriamą sistemą būtina turėti konkrečių AST generavimo įrankį).
- ***Koks turėtų būti bendrinio AST formatas?*** Bendrinio AST formatas turi būti pakankamai lankstus, kad būtų galima atvaizduoti įvairių programavimo kalbų konstrukcijas. Be to, formatas turėtų leisti talpinti ne tik sintaksinę, bet ir semantinę kalbos elementų informaciją.
- ***Kaip pradiniame medyje rasti norimas AST elementų sekas ir koks turėtų būti pradinio medžio informacijos užklausų vykdymo mechanizmas?*** Įprastai tiek elementų sekų aprašymas, tiek paprastų lokalių užklausų vykdymas atliekamas vienu metu. Sekų aprašas lemia, kada sutikus tam tikrą pradinio AST elementų seką kuriama naujojo AST elementų seka. Naujai sukurtajai sekai dažniausiai reikalinga ne tik struktūrinė informacija (t.y., kokie AST elementai sukurti ir kaip jie tarpusavyje sujungti), bet ir pradinio medžio elementų reikšmės (pavyzdžiui, jei naujajame AST kuriamas objektinės programavimo kalbos klasės elementas, reikalinga ir klasės pavadinimo reikšmė).
- ***Kokiu formatu aprašyti medžių transformacijas ir kaip iš nesudėtingų transformacijų sukurti stambesnius transformacijų darinius?*** Analizuotose priemonėse įprastai naudojamos nesudėtingos transformacijų taisyklės bei transformacijų strategijos. Taisyklės

aprašo paprastų kalbos darinių transformacijas (pavyzdžiui, kintamojo deklaravimo), o strategijos lemia taisyklių pritaikymą pilnam pradiniam medžiui.

Tolesniuose skyreliuose aptariami minėtųjų problemų sprendimų būdai bei sprendimų pasirinkimo priežastys.

2.4. Esminių uždavinio sprendimų analizė

2.4.1. Konkretaus AST formato ir generavimo įrankio pasirinkimas

2.4.1.1. Abstrakčių sintaksės medžių generatorius ApiGen

ApiGen yra įrankis, kuris automatiškai generuoja abstrakčių sintaksės medžių realizacijas C programavimo kalbai **[14]**. Jis priima glaustą abstraktaus duomenų tipo aprašą ir sugeneruoja C kodą abstraktiems sintaksės medžiams, kurie griežtai tipizuoti ir maksimaliai išnaudoja galimybę dalintis bendromis išraiškomis, taip siekiant efektyvaus atminties panaudojimo ir greito lygybės tikrinimo. Esminė ApiGen idėja yra ta, kad pilnai funkcionali ir optimizuota AST duomenų struktūros realizaciją gali būti sugeneruota automatiškai, ir turėti labai suprantamą bei saugią tipų atžvilgiu sąsają. Nors ApiGen iš pradžių buvo skirta C programavimo kalbai, vėliau išleista jos versija, skirta Java programavimo kalbai.

Duomenų tipo aprašas glaustai ir tiksliai apibūdina, kaip medžio tipo struktūra turėtų būti sukonstruota. Jame yra tipai ir konstruktoriai. Konstruktoriai apibūdina konkrečių tipų alternatyvias formas pagal jų pavadinimus bei jų vaikų tipus ir pavadinimus. Šio aprašo pavyzdį galima matyti 10 paveiksle.

```
datatype Expressions
  Bool ::= true
         | false
         | eq(lhs:Expr, rhs:Expr)
  Expr ::= id(value:str)
         | nat(value:int)
         | add(lhs:Expr, rhs:Expr)
         | mul(lhs:Expr, rhs:Expr)
```

10 pav. Duomenų tipo aprašas

Iš šio aprašo sugeneruojamos Java kalbos klasės, pavaizduotos žemiau (11 pav.).

```
abstract public class Bool extends ATermAppl {...}
package bool;
  public class True extends Bool {...}
  public class False extends Bool {...}
  public class Eq extends Bool {...}
abstract public class Expr extends ATermAppl {...}
package expr;
  public class Id extends Expr {...}
  public class Nat extends Expr {...}
  public class Add extends Expr {...}
  public class Mul extends Expr {...}
```

11 pav. Java kalbos klasės, sugeneruotos iš duomenų tipo aprašo

Pateiktame sugeneruotų klasių pavyzdyje matomi tik klasių aprašai, tačiau praktiškai ApiGen sukuria detalias klasių realizacijas, kurios leidžia išreikšti įvairias medžių struktūras, suteikia galimybes atlikti greitą norimų elementų paiešką medyje bei įgalina atlikti medžio elementų perėjimą naudojantis „lankytojo šablonu“ (angl. „visitor pattern“).

2.4.1.2. Gramatinės analizės generatorių kūrimo įrankis ANTLR

ANTLR – tai įrankis, skirtas generuoti gramatinės analizės generatorius (angl. „parsers“) pagal norimų programavimo kalbų gramatikos aprašus **[15]**. Gramatinės analizės generatorių paskirtis yra programos kodą paversti į abstraktų sintaksės medį. ANTLR sukuria C, C++ ir Java programavimo kalbomis realizuotus gramatinės analizės generatorius, kurie pakankamai suprantami žmogui, t.y. generatoriaus programos kodas ganėtinai lengvai skaitomas ir yra orientuotas į žmogų, o ne į kompiliatorių, kuris šį kodą kompiliuos. ANTLR pasižymi tokiomis savybėmis:

- Leidžia specifikuoti leksinę bei sintaksinę kodo struktūrą;
- Priima gramatikos konstruktus išreikštus išplėstinės Backus-Naur formos notacija;
- Turi priemones automatiniam abstrakčių sintaksės medžių generavimui;
- Generuoja tiesiogiai atsekamą gramatinės analizės generatorių kodą (matomas ryšys tarp kodo ir gramatikos aprašo);
- Leidžia automatiškai bei rankiniu būdu valdyti analizės metu kylančias klaidas;
- Pats ANTLR yra parašytas lengvai pritaikoma įvairioms platformoms C kalbos realizacija.

ANTLR yra plačiai naudojamas tiek komercinėse, tiek akademinėse bendrijose. Nuo pirmosios programos versijos išleidimo 1992 metais šią programinę įrangą įsigijo daugiau nei 1000 registruotų vartotojų 37 šalyse. Keletas universitetų turi mokymo modulius, skirtus ANTLR pažinimui. ANTLR naudoja daugybė komercinę programinę įrangą kuriantys programuotojai.

2.4.1.3. Pasirinktas konkrečių AST generavimo įrankis

Palyginus ApiGen ir ANTLR buvo nuspręsta pasirinkti paskesnįjį įrankį. Tai lėmė kelios priežastys:

1) ApiGen įrankiu sukurtos konkrečių AST struktūros yra ganėtinai gerai struktūrizuotos, tačiau įrankis nepalaiko medžių generavimo iš kodo, t.y. ApiGen skirtas tik pačių medžio duomenų struktūrų generavimui, bet ne medžių kūrimui iš kodo. Kita vertus, ANTLR skirtas generuoti gramatinės analizės generatorius ir sugeba talpinti jų sukurtus konkrečius AST.
2) ANTLR yra labiau paplitęs įrankis. Atlikus informacijos paiešką apie ANTLR, pavyko surasti ganėtinai daug jį nagrinėjančių informacijos šaltinių. Šis faktorius svarbus, nes yra pateikta daugiau įrankio naudojimo pavyzdžių ir su jo naudojimu susijusių problemų sprendimų.
3) ANTLR įrankiui yra sukurta daugybė realių programavimo kalbų gramatikos aprašų, kurie internete platinami nemokai. Būtent kalbų gramatikos aprašai lemia, kaip bus sugeneruoti konkrečiai programavimo kalbai skirti gramatinės analizės generatoriai.

2.4.2. Bendrinio AST formato pasirinkimas

Renkantis bendrinio AST formatą iš pradžių buvo svarstoma sukurti nuosavą formatą, tačiau šios idėjos buvo greitai atsisakyta, nes tokio masto užduotis galėtų trukti ne vienerius metus.

Vėliau atlikus egzistuojančių sprendimų analizę buvo nuspręsta naudoti AST medžius, kurių sandara yra aprašyta Object Management Group (arba OMG) **[16]** – tarptautinio, atviros narystės, ne pelno siekiančio, kompiuterių industrijos standartus kuriančio konsorciumo – abstrakčios sintaksės medžio metamodelio specifikacijoje (angl. „Abstract Syntax Tree Metamodel Specification“) **[17]**. Ši specifikacija skirta aprašyti bendrinių AST medžių sandarą, kuri skirta lengvam duomenų pasikeitimui tarp skirtingų įrankių, kurie gali būti net skirtingų gamintojų.

Ši specifikacija pasirinkta dėl šių priežasčių:

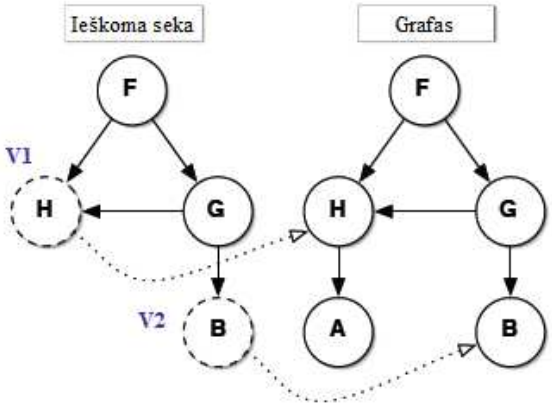
1) Specifikacijos aiškumas. AST medį realizuojančios klasės yra gerai apibrėžtos ir turi aiškią paskirtį.
2) Aukšta specifikacijos kokybė. OMG yra daugybės gerai žinomų ir plačiai informacinėse technologijose taikomų standartų kūrėjai, turintys žymiai didesnę patirtį, nei šiame darbe aprašomos bibliotekos kūrėjas, todėl šios specifikacijos kokybe ir universalumu galima pasitikėti.

3) AST medžių modelio sudarymo sąnaudų sumažinimas. Kadangi naudojamasi jau egzistuojančia specifikacija, sutaupytas bibliotekos autoriaus laikas ir pastangos, kurios būtų reikalingos kuriant savą AST medžių modelį.

2.4.3. AST elementų sekų radimo ir užklausų vykdymo mechanizmo pasirinkimas

Šaltinis **[18]** pateikia paprastus įvairių duomenų struktūrų (grafų, dinaminių sąrašų, masyvų, medžių ir pan.) sekų radimo ir užklausų vykdymo metodus. Pagrindinė metodų idėja remiasi tuo, kad kuriant atitinkamos duomenų struktūros seką, kurią norime rasti, tuo pačiu sekoje pažymime elementus, kurie sutapatinus seką bus naudojami kaip kintamieji.

Tarkime turime tam tikrų objektų grafą ir seką, su kuria bandome sulyginti grafą (12 pav.). Atliekant sekos sutapatinimą su grafu, atliekami visų sekos elementų sulyginimai su grafo elementais pagal elementų tipus. Kadangi tiek seka, tiek grafas turi ta pačia tvarka išdėstytus elementus F, H, G, B, paieška sėkminga. Ir kadangi sekoje H ir B elementai buvo pažymėti kaip kintamieji V1 ir V2, grafe rasti atitinkami elementai grąžinami kaip atitinkamų kintamųjų reikšmės. Šiuo atveju V1 kintamasis įgyja nuorodą į grafo elementą H, kuris tuo pačiu turi ir vaikinį elementą A, o V2 – į elementą B.



12 pav. Objektų grafų sulyginimo principo pavyzdys

Pateiktasis metodas pasirinktas dėl šių priežasčių:

1) Aiškumo. Metodas paprastas ir lengvai suvokiamas.
2) Lankstumo. Grafo pografiai, grąžinami kintamųjų pavidalu, gali būti naudojami ne tik tiesioginių reikšmių išėmimui iš pradinio grafo, bet ir kaip nauji grafai, kuriems bus bandomos pritaikyti naujos sekos (vyks rekursinis sekų sutapatinimas).

2.4.4. AST medžių transformacijos aprašo formato pasirinkimas

Atlikus egzistuojančių įrankių transformacijos aprašų analizę nuspręsta transformacijas aprašinėti ne kokia nors specializuota aprašo kalba, o įvairiems tikslams pritaikomu XML formatu. Formato pagrindas – tai transformacija, sudaryta iš aibės dalinių transformacijų (transformavimo taisyklių). Pagrindinė transformacija nurodo, kurią iš esančių transformavimo taisyklių pritaikyti transformuojant šakninį pateiktojo pradinio medžio elementą. Tolesnės dalinės transformacijos vykdomos tuomet, kai medžio elementai sutampa su tam tikra seka, kuri toliau rekursiškai nurodo, su kokiomis taisyklėmis apdoroti kuriuos rastus medžio elementus. Panašiai aprašomos ir minėtosios XSL transformacijos. Detalesnė formato struktūra aptariama tolesniuose skyriuose.

Formatas pasirinktas dėl šių priežasčių:

1) Aiškumo. Formatas paprastas ir žmogui suprantamas, todėl yra nesunkiai įskaitomas.
2) Universalumo. XML formatu gali būti saugoma įvairaus pobūdžio informacija ir su juo gali dirbti daugybė šiuolaikinės programinės įrangos.

2.5. Analizės išvados

1) Pasaulyje egzistuoja ne viena priemonė, kuri savo pobūdžiu panaši į AST transformavimo sistemą. Tačiau nei viena iš šių priemonių nėra orientuota į konkrečių AST pavertimą į bendrinę formą ir atvirkščiai. Tai požymis, kad AST transformavimo sistema yra šiek tiek kitokia, labiau specializuota, ir todėl ją veikiausia tikslinga kurti.
2) Atliekant egzistuojančių priemonių palyginimą, atsiskleidė bruožai, bendri joms visoms. Taip pat pavyko „suskaidyti“ šias priemones į esminius komponentus, kurie tokioms priemonėms reikalingi ir/ar svarbūs. Kadangi apžvelgta ganėtinai nemažai priemonių (tiksliau, penkios), tai yra pakankamai geras pagrindas AST transformavimo sistemos kūrimui.
3) AST transformavimo sistemos analizės fazėje buvo susidurta ir su specifinėmis problemomis. Jei nebūtų buvus atlikta gera analizė, sistemos kūrimo pastangos būtų galėję gerokai išaugti. Tai tapo ypač akivaizdu analizuojant bendrinio AST formato pasirinkimo klausimą. Jei būtų nuspręsta kurti nuosavą bendrinį formatą, tikėtina, jog pačios transformacijos sistemos kūrimas būtų gerokai užsitęsęs dėl nuolat kuriamo formato.
4) Pasirinktasis AST transformavimo sistemos užklausų realizavimo mechanizmas ganėtinai lengvai suprantamas ir paprastas. Tačiau jis apima tik lokalios informacijos gavimą vykdant dalines transformacijas pagal pateiktas taisykles. Sistemoje nenumatytas mechanizmas, gebantis iš pradinio medžio išgauti informaciją iš globalaus konteksto. Analizuotos literatūros pateiktuose pavyzdžiuose globalaus konteksto informacijos išgavimas naudotas atliekant įvairiais medžių optimizacijas, o ne įprastas transformacijas. Todėl tikimasi, jog šio mechanizmo pakaks. Vis dėlto gali iškilti globalios informacijos gavimo poreikis. Tačiau į šį klausimą gali atsakyti tik nuodugnus AST transformavimo sistemos naudojimo eksperimentas.

3. AST TRANSFORMAVIMO SISTEMOS PROTOTIPO PROJEKTAS

3.1. Vartotojų analizė

3.1.1. Svarbiausias vartotojas

Vartotojo kategorija: informatikos doktorantas (*Linas Ablonskis, projekto užsakovas*);
Vartotojo sprendžiami uždaviniai: programos kodo generatoriaus konfigūravimo metodo, kuris leistų automatiškai sukonfigūruoti kodo generatorių analizuodamas žmogaus parašytą kodą, kūrimas;
Patirtis dalykinėje srityje: srities specialistas;
Patirtis informacinėse technologijose: informatikas;
Papildomos vartotojo charakteristikos:

- *Amžiaus grupė:* 25 – 30 metų;
- *Kalbų išmanymas:* geros anglų kalbos žinios;
- *Požiūris į IT:* palankus;
- *Požiūris į darbą:* palankus.
- *Apsimokymo poreikis:* reikalinga bibliotekos kodo dokumentacija.

3.1.2. Antraeilis vartotojas

Vartotojo kategorija: programuotojas;
Vartotojo sprendžiami uždaviniai: kodo transformacijų, kodo pertvarkymų, srities kalbų bei modelių transformacijų įrankių kūrimas;
Patirtis dalykinėje srityje: įprastas darbuotojas arba srities specialistas;
Patirtis informacinėse technologijose: patyręs arba informatikas;
Papildomos vartotojo charakteristikos:

- *Amžiaus grupė:* 20 – 70 metų;
- *Kalbų išmanymas:* vidutinės arba geros anglų kalbos žinios;
- *Požiūris į IT:* palankus;
- *Požiūris į darbą:* neutralaus arba palankus;
- *Apsimokymo poreikis:* reikalinga bibliotekos arba bibliotekos pagrindu sukurto įrankio programinio kodo dokumentacija.

3.1.3. Nesvarbūs (atsitiktiniai) vartotojai

Vartotojo kategorija: programuotojas;

Vartotojo sprendžiami uždaviniai: kodo transformacijų ir/arba kodo pertvarkymų įrankių naudojimas;

Patirtis dalykinėje srityje: įprastas darbuotojas arba srities specialistas;

Patirtis informacinėse technologijose: patyręs arba informatikas;

Papildomos vartotojo charakteristikos:

- *Amžiaus grupė:* 20 – 70 metų;
- *Kalbų išmanymas:* vidutinės arba geros anglų kalbos žinios;
- *Požiūris į IT:* palankus;
- *Požiūris į darbą:* neutralaus arba palankus;
- *Apsimokymo poreikis:* reikalinga bibliotekos pagrindu sukurto įrankio vartotojo dokumentacija ir/ar video apmokymai (šio poreikio tenkinimo užtikrinimas yra už šio projekto ribų, nes bus naudojamasi įrankiais, sukurtais bibliotekos pagrindu, o ne tiesiogiai pačia biblioteka).

Vartotojo kategorija: informacinių sistemų analitikas;

Vartotojo sprendžiami uždaviniai: srities kalbų naudojimas, modelių transformacijas naudojančių įrankių naudojimas;

Patirtis dalykinėje srityje: įprastas darbuotojas arba srities specialistas;

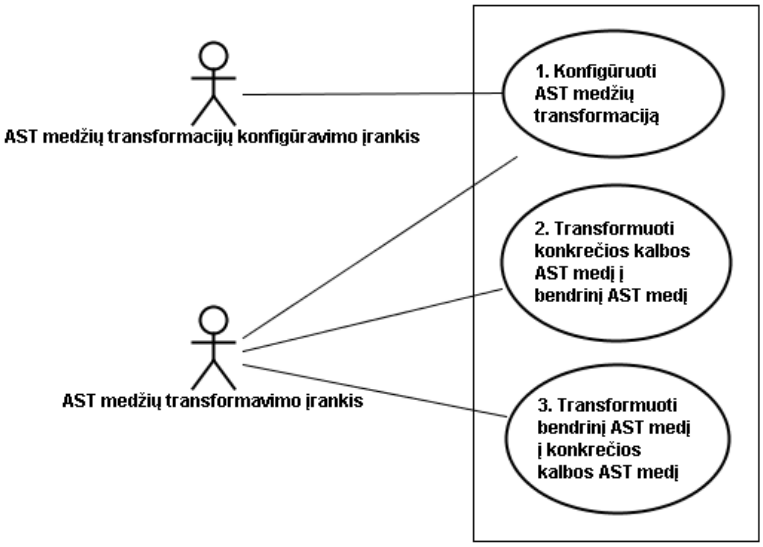
Patirtis informacinėse technologijose: patyręs arba informatikas;

Papildomos vartotojo charakteristikos:

- *Amžiaus grupė:* 20 – 70 metų;
- *Kalbų išmanymas:* vidutinės arba geros anglų kalbos žinios;
- *Požiūris į IT:* neutralus arba palankaus;
- *Požiūris į darbą:* neutralus arba palankaus;
- *Apsimokymo poreikis:* reikalinga bibliotekos pagrindu sukurto įrankio vartotojo dokumentacija ir/ar video apmokymai (šio poreikio tenkinimo užtikrinimas yra už šio projekto ribų, nes bus naudojamasi įrankiais, sukurtais bibliotekos pagrindu, o ne tiesiogiai pačia biblioteka).

3.2. Panaudojimo atvejų modelis

3.2.1. Panaudojimo atvejų diagrama



13 pav. Panaudojimo atvejų diagrama

3.2.2. Panaudojimo atvejų sąrašas

1. PANAUDOJIMO ATVEJIS: Konfigūruoti AST medžių transformaciją
Vartotojas/Aktorius: AST medžių transformacijų konfigūravimo įrankis; AST medžių transformavimo įrankis.
Aprašas: Sukuriama nurodytų AST medžių tipų transformacijų iš pirmojo medžio tipo į antrąjį (ir atvirkščiai) konfigūracija.
Prieš-sąlyga: Pateiktas pilnas bendrojo ASM medžio struktūrinis aprašas; Pateiktas pilnas arba dalinis konkrečios programavimo kalbos AST medžio struktūrinis aprašas.
Sužadinimo sąlyga: Ketinama konfigūruoti AST medžių transformaciją.
Po-sąlygą: Sukurta AST medžių transformacijų konfigūracija.

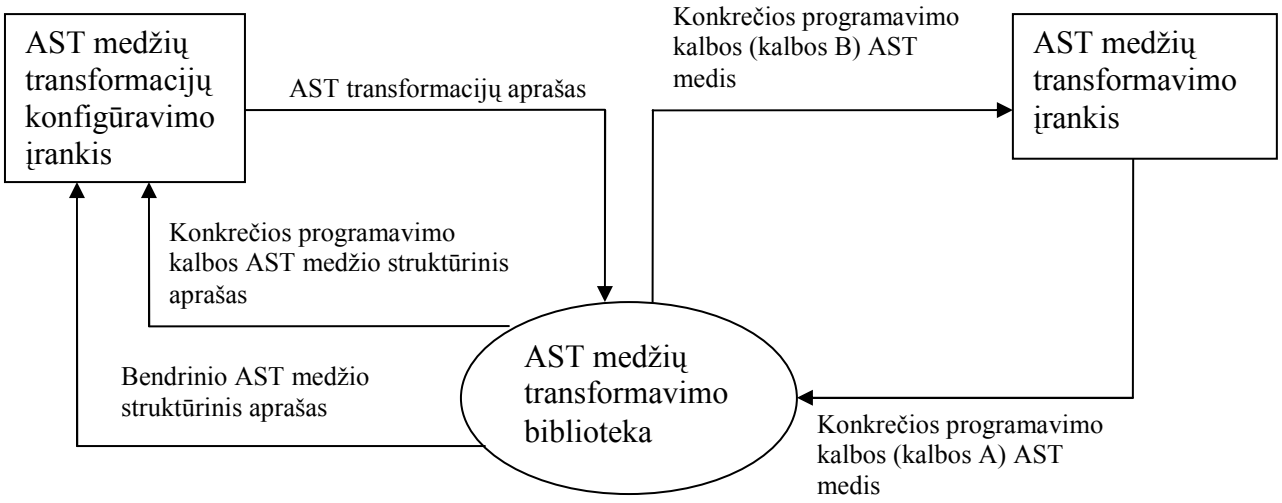
2. PANAUDOJIMO ATVEJIS: Transformuoti konkrečios kalbos AST medį į bendrinį AST medį
Vartotojas/Aktorius: AST medžių transformavimo įrankis.
Aprašas: Pateiktas konkrečios kalbos AST medis transformuojamas į semantiškai ekvivalentų bendrinį AST medį.
Prieš-sąlyga: Pateikta medžių transformacijos konfigūracija; Pateiktas konkrečios kalbos AST medis.
Sužadinimo sąlyga: Iškviečiama medžių transformacijos komanda.
Po-sąlygą: Sukurtas bendrinis AST medis, semantiškai ekvivalentus pradiniam.

3. PANAUDOJIMO ATVEJIS: Transformuoti bendrinį AST medį į konkrečios kalbos AST medį
Vartotojas/Aktorius: AST medžių transformavimo įrankis.
Aprašas: Pateiktas bendrinis AST medis transformuojamas į semantiškai ekvivalentų konkrečios programavimo kalbos AST medį.
Prieš-sąlyga: Pateikta medžių transformacijos konfigūracija; Pateiktas bendrinis AST medis.
Sužadinimo sąlyga: Iškviečiama medžių transformacijos komanda.
Po-sąlygą: Sukurtas konkrečios programavimo kalbos AST medis, semantiškai ekvivalentus pradiniam.

3.3. Reikalavimų specifikacija

3.3.1. Veiklos sfera

3.3.1.1. Veiklos kontekstas



14 pav. Veiklos konteksto diagrama

3.3.1.2. Veiklos įvykiai ir informacijos srautai

Eil. nr.	Įvykio pavadinimas	Įeinantys/išeinantys informacijos srautai
1	AST medžių transformacijų konfigūravimo įrankis sukuria transformacijų aprašą	• AST transformacijų aprašas (*įeinantis*) • Konkrečios programavimo kalbos AST medžio struktūrinis aprašas (*išeinantis*) • Bendrinio AST medžio struktūrinis aprašas (*išeinantis*)
2	Atliekama konkrečios kalbos AST medžio transformacija į kitos konkrečios kalbos AST medį	• Konkrečios programavimo kalbos (kalbos B) AST medis (*įeinantis*) • Konkrečios programavimo kalbos (kalbos A) AST medis (*išeinantis*)

3.3.2. Funkciniai reikalavimai

Reikalavimas #: 1	**Reikalavimo tipas:** 9	**Panaudojimo atvejis #:** 2
Aprašymas:	AST medžių transformavimo biblioteka privalo transformuoti konkrečios programavimo kalbos AST medį į bendrinį AST medį.	
Pagrindimas:	Tai – viena iš kelių esminių funkcijų. Nerealizavus šios funkcijos AST medžių transformavimo biblioteka praranda savo prasmę.	
Šaltinis: Linas Ablonskis – užsakovas		
Tinkamumo kriterijus:	AST medžių pertvarkymo biblioteka transformuoja konkrečios programavimo kalbos AST medį į apibrėžtos formos AST medį neprarasdama semantinės pradinio medžio prasmės.	
Užsakovo patenkinimas: 5		**Užsakovo nepatenkinimas:** 5
Priklausomybės: 3		**Konfliktai:** nėra
Papildoma medžiaga: Nėra		
Istorija: Sukurta 2009 03 25		

Reikalavimas #: 2	**Reikalavimo tipas:** 9	**Panaudojimo atvejis #:** 3
Aprašymas:	AST medžių transformavimo biblioteka privalo transformuoti bendrinį AST medį į konkrečios programavimo kalbos AST medį.	
Pagrindimas:	Tai – viena iš kelių esminių funkcijų. Nerealizavus šios funkcijos AST medžių transformavimo biblioteka praranda savo prasmę.	
Šaltinis: Linas Ablonskis – užsakovas		
Tinkamumo kriterijus:	AST medžių pertvarkymo biblioteka transformuoja bendrinį AST medį į konkrečios programavimo kalbos AST medį neprarasdama semantinės pradinio medžio prasmės.	
Užsakovo patenkinimas: 5		**Užsakovo nepatenkinimas:** 5
Priklausomybės: 3		**Konfliktai:** Nėra
Papildoma medžiaga: Nėra		
Istorija: Sukurta 2009 03 25		

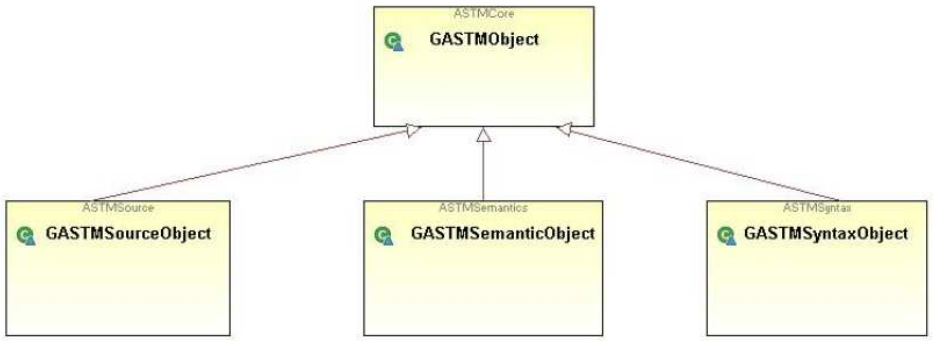
Reikalavimas #: 3	**Reikalavimo tipas:** 9	**Panaudojimo atvejis #:** 1
Aprašymas:	AST medžių transformacijų konfigūravimo įrankis privalo leisti vartotojui sudaryti medžių transformacijų konfigūraciją. Vartotojas turi pasirinkti, kuris vieno medžio elementas atitinka kurį kito medžio elementą, ir kokie yra transformacijos nustatymai.	
Pagrindimas:	Bendrinė AST medžio elementų prasmė gerai žinoma, tačiau nėra aiški konkrečios programavimo kalbos AST medžio elementų prasmė, todėl AST medžių transformavimo biblioteka negali automatiškai atlikti transformacijos. Būtina nurodyti transformacijos konfigūraciją, kurią privalo sudarinėti žmogus.	
Šaltinis: Linas Ablonskis – užsakovas		
Tinkamumo	AST medžių transformacijų konfigūravimo įrankis sukurs medžių transformacijos	

kriterijus: aprašą.	
Užsakovo patenkinimas: 5	**Užsakovo nepatenkinimas:** 5
Priklausomybės: Nėra	**Konfliktai:** Nėra
Papildoma medžiaga: Nėra	
Istorija: Sukurta 2009 03 25	

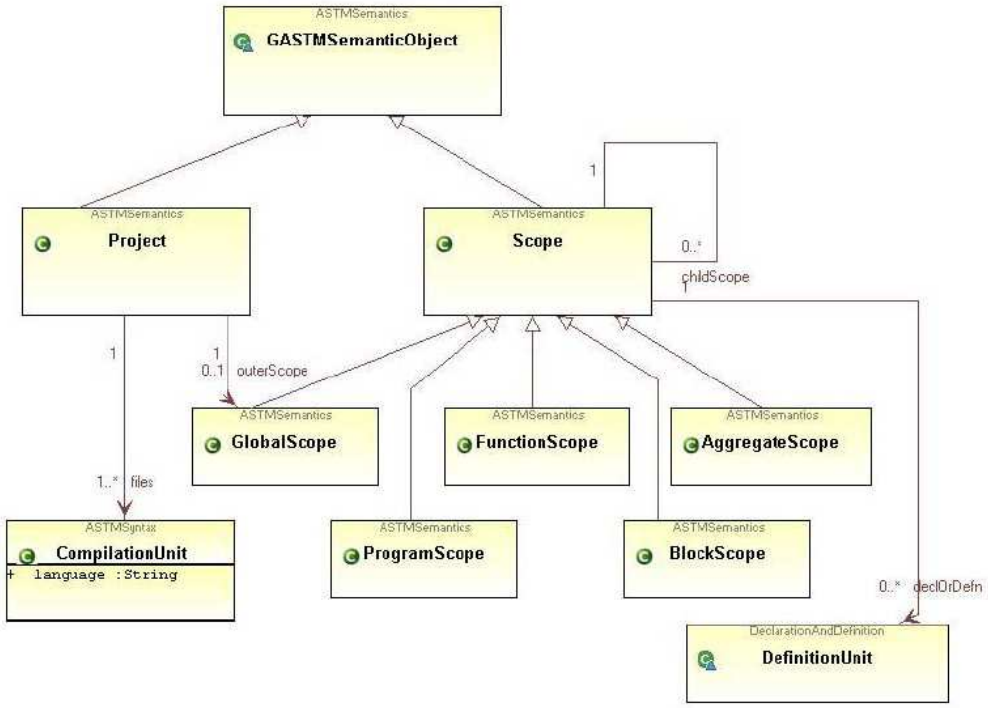
3.3.3. Reikalavimai duomenims

Bendrinio AST formatas privalo būti paremtas Object Management Group abstrakčios sintaksės medžio metamodelio specifikacija. Pagrindinės bendrinio AST modelio klasės:

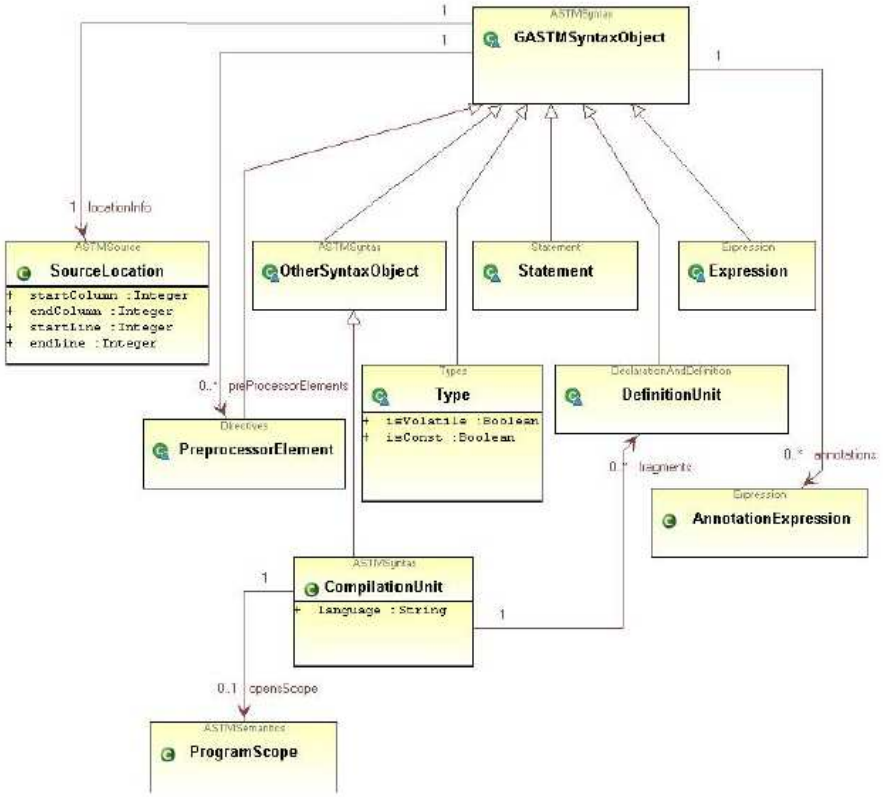
- *GASTMObject* – abstrakti bazinė visų medžio objektų klasė (15 pav.);
- *GASTMSourceObject* – klasė, sauganti programinio kodo šaltinių informaciją (pavyzdžiui, informaciją apie tam tikro kintamojo deklaravimo vietą faile);
- *GASTMSemanticObject* – klasė, išreiškianti kodo semantinę struktūrą (16 pav.);
- *GASTMSyntaxObject* – klasė, išreiškianti kodo sintaksinę struktūrą (17 pav.).



15 pav. Bazinės bendrinio AST klasės



16 pav. GASTMSemanticObject klasių hierarchija



17 pav. GASTMSyntaxObject klasių hierarchija

3.3.4. Nefunkciniai reikalavimai

3.3.4.1. Reikalavimai sistemos išvaizdai

AST medžių transformavimo bibliotekai reikalavimai sistemos išvaizdai netaikomi, nes tai grafinės sąsajos neturintis produktas. Tačiau taikomi reikalavimai įrankiams:

- Naudojimo paprastumas;
- Programuotojams pritaikyta grafinė sąsaja;
- Anglų kalbos naudojimas.

3.3.4.2. Reikalavimai panaudojamumui

Keliami tokie reikalavimai:

- Bibliotekos kodas ir komentarai privalo būti parašyti anglų kalba;
- Naudojami programuotojams pažįstami terminai ir sąvokos.

3.3.4.3. Reikalavimai valdymo charakteristikoms

Biblioteka privalo atlikti operacijas per „protingą“ laiko tarpą. Pavyzdžiui, jei atliekama konkretaus AST konvertavimas į bendrinį AST, ir jei konkretus AST atitinka maždaug vieno A4 lapo, parašytu 10 šriftu, apimties kodo kiekį, tuomet konvertavimo procesas neturėtų užtrukti ilgiau, nei 1-2 sekundes.

3.3.4.4. Reikalavimai veikimo sąlygoms

Biblioteka privalo veikti ribotas vartotojo teises turinčiame kompiuteryje.

3.3.4.5. Reikalavimai sistemos priežiūrai

Biblioteka privalo laikytis gero komponento principų (žiūrėkite 3.4.1.1 skyrelį).

3.3.4.6. Reikalavimai saugumui

Saugumo reikalavimai netaikytini dėl sistemos pobūdžio.

3.3.4.7. Kultūriniai-politiniai reikalavimai

Bibliotekos programinis kodas bei komentarai privalo būti parašyti anglų kalba. Ši kalba plačiai paplitus, todėl ja galės naudotis daugybė žmonių. Be to, ši kalba turi techninių sąvokų, kurios sunkiai/netiksliai verčiamos į kitas kalbas.

3.4. Sistemos architektūra

3.4.1. Architektūros tikslai ir apribojimai

3.4.1.1. Gero komponento principų laikymasis

Biblioteka privalo laikytis tokių gero komponento principų:

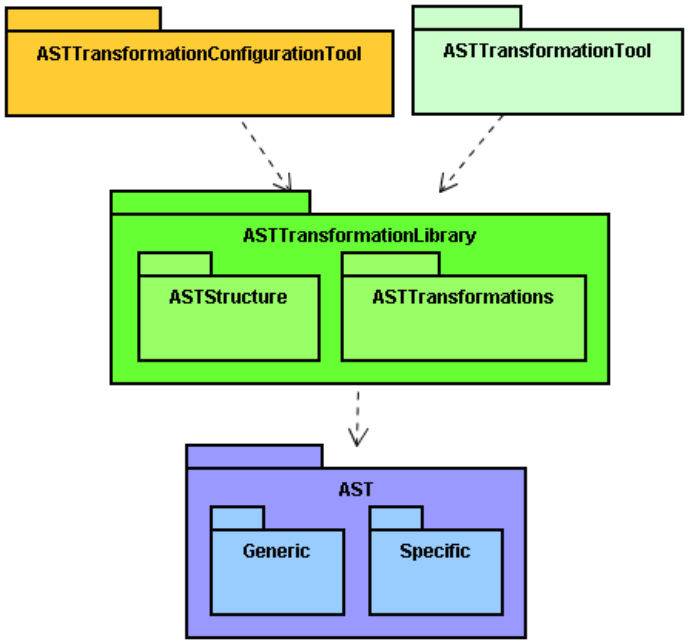
- Aiški ir gerai apibrėžta sąsaja;
- Biblioteka privalo būti savarankiška (pateikta su visu kodu ir kitais failais, reikalingais jos veikimui);
- Biblioteka privalo būti nepriklausoma nuo konkrečios platformos (privalo nenaudoti konkrečiai platformai specializuoto kodo ir/ar įrankių);
- Biblioteka privalo būti gerai inkapsuliuota (realizacijos pakeitimai neturi lemti sąsajos pakitimų).

3.4.1.2. Įvairių platformų palaikymas

Siekiant, jog kuriama biblioteka būtų plačiai pritaikoma, ji įgyvendinama Java programavimo kalba. Kadangi privaloma laikytis nepriklausomumo nuo konkrečios platformos pricipo, todėl kuriama biblioteka naudoja tik standartines Java kalbos galimybes ir nenaudoja specifinių konkrečiai operaciniai sistemai kodo bibliotekų. Biblioteka privalo veikti visose aplinkose, kuriose instaliuota Java virtuali mašina.

3.4.2. Sistemos statinis modelis

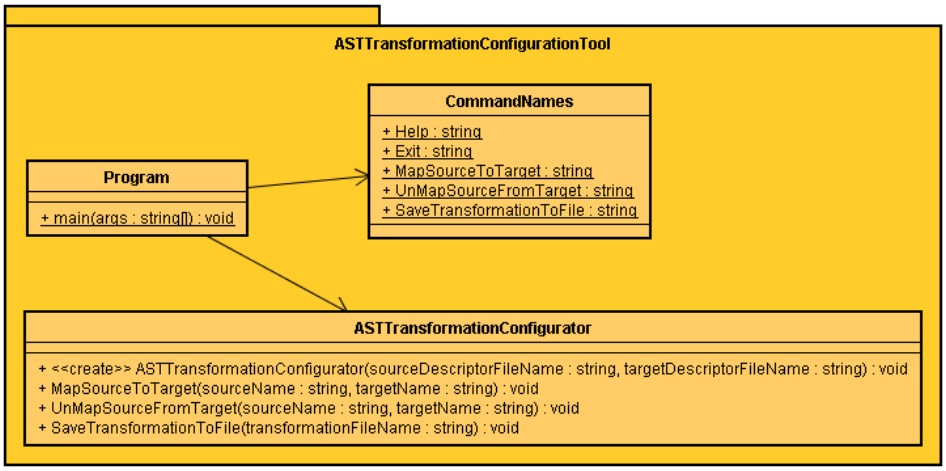
3.4.2.1. Bendra paketų diagrama



18 pav. Bendra paketų diagrama

3.4.2.2. Paketų detalizavimas

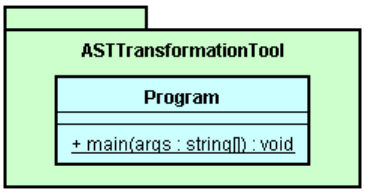
3.4.2.2.1. „ASTTransformationConfigurationTool“ paketas



19 pav. Detalizuota AST transformacijų konfigūravimo įrankio paketo diagrama

Šį paketą sudaro klasės, realizuojančios AST transformacijų konfigūravimo įrankį. `Program` klasė yra programos pradžios taškas.

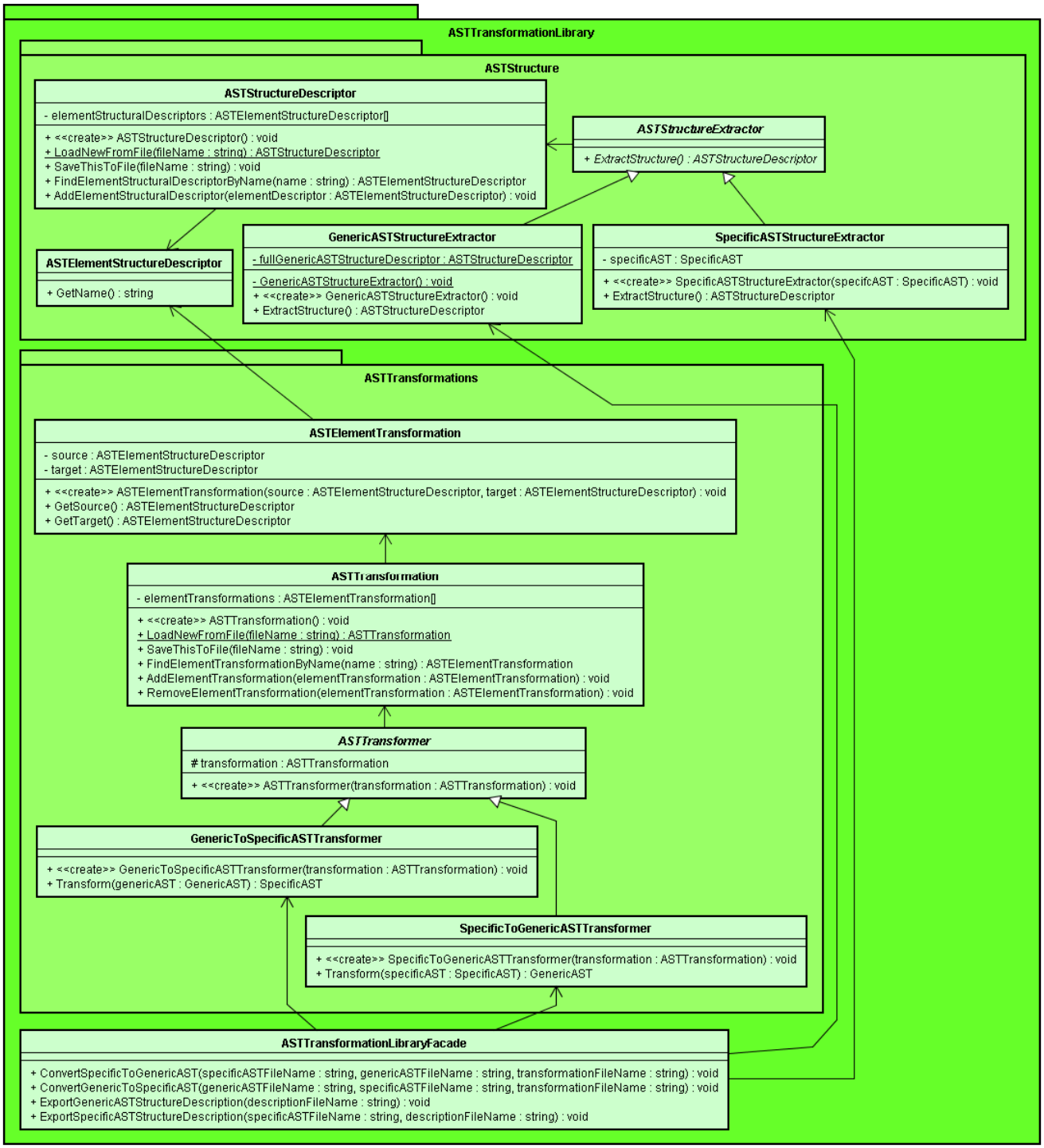
3.4.2.2.2. „ASTTransformationTool“ paketas



20 pav. Detalizuota AST transformavimo įrankio paketo diagrama

Šį paketą sudaro klasės, realizuojančios AST transformavimo įrankį. `Program` klasė yra programos pradžios taškas.

3.4.2.2.3. „ASTTransformationLibrary“ paketas



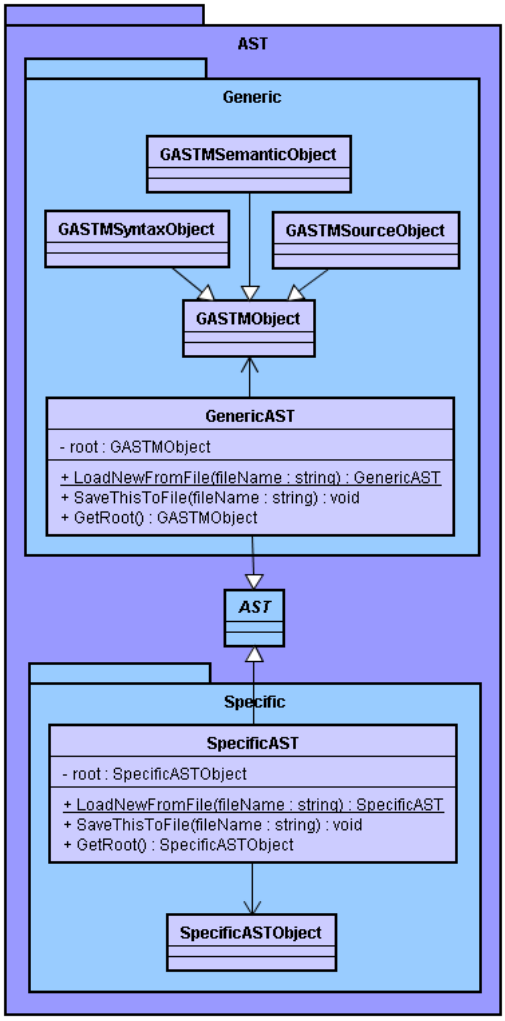
21 pav. Detalizuota AST transformacijų bibliotekos paketo diagrama

Šį paketą sudaro klasės, realizuojančios AST transformacijų biblioteką. Svarbiausios šio paketo sudedamosios dalys:

- Klasė `ASTTransformationLibraryFacade`, kuri pateikia esmines bibliotekos funkcijas per paprastą ir aiškią sąsają;

- Paketas `ASTStructure`, kurį sudaro klasės, apibūdinančios AST medžių struktūrą ir atliekančios įvairius veiksmus su ja;
- Paketas `ASTTransformations`, kurį sudaro klasės, apibūdinančios AST medžių transformacijų nustatymus ir atliekančios įvairius transformavimo veiksmus.

3.4.2.2.4. „AST“ paketas



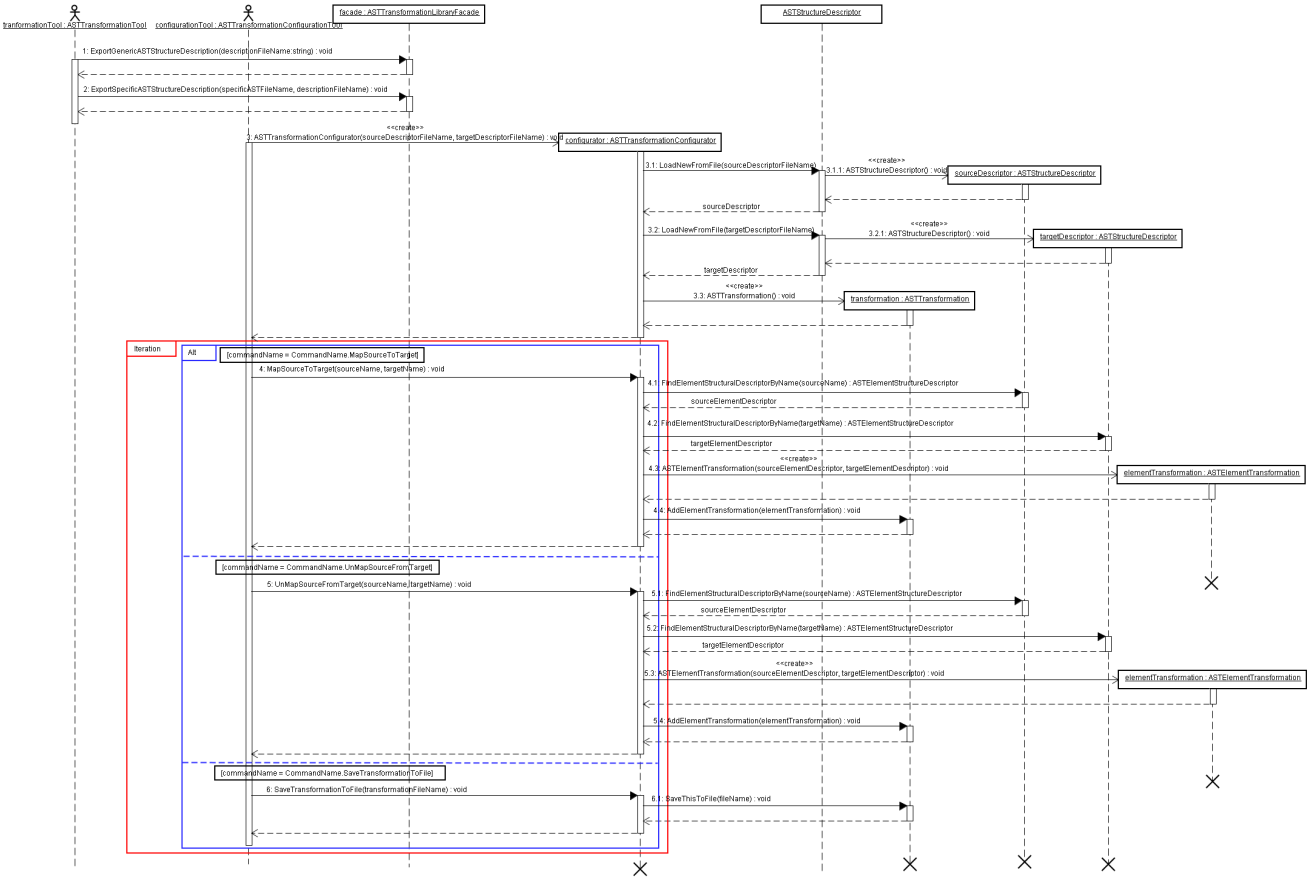
22 pav. Detalizuota AST medžių paketo diagrama

Šį paketą sudaro klasės, apibūdinančios AST medžių sandarą ir realizuojančios medžių išsaugojimo, užkrovimo ir kitus veiksmus. Svarbiausios šio paketo sudedamosios dalys:

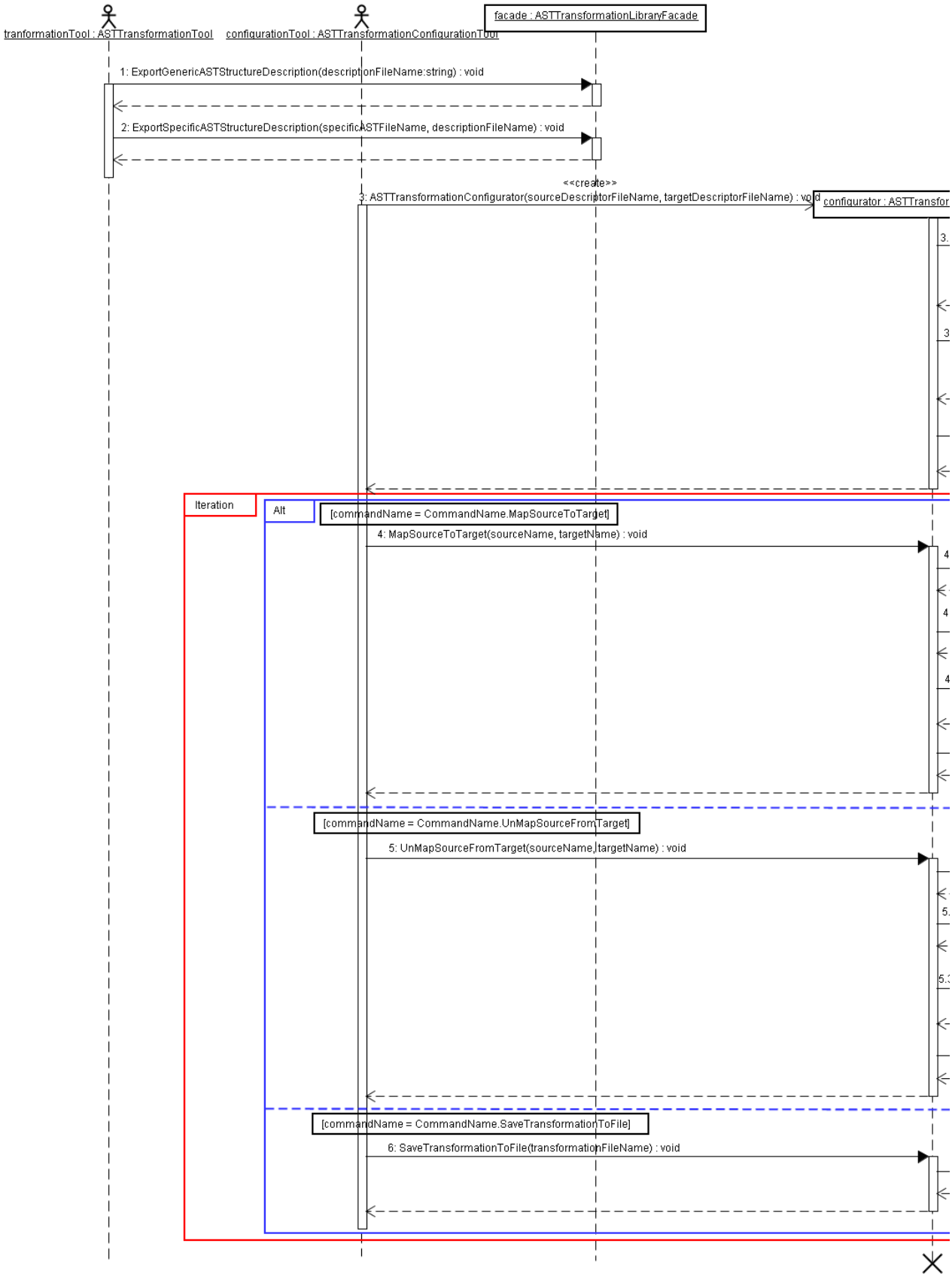
- Klasė `AST`, kuri yra visų AST medžių tėvinė klasė;
- Paketas `Generic`, kurį sudaro klasės, apibūdinančios bendrinių AST medžių sandarą ir realizuojančios įvairius veiksmus su bendriniais AST medžiais;

- Paketas `Specific`, kurį sudaro klasės, apibūdinančios konkrečių AST medžių sandarą ir realizuojančios įvairius veiksmus su konkrečių kalbų AST medžiais.

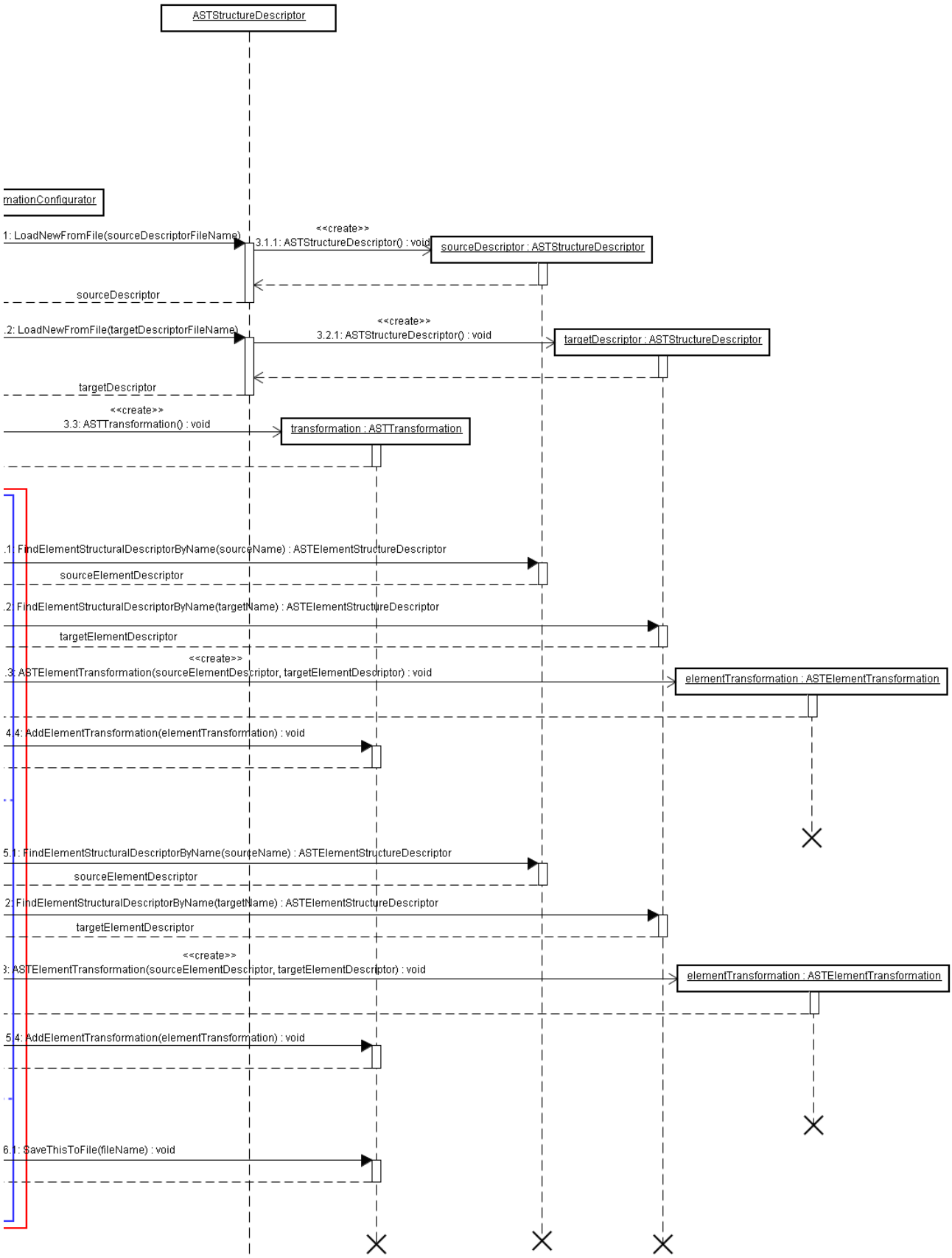
3.5. Sistemos elgsenos modelis



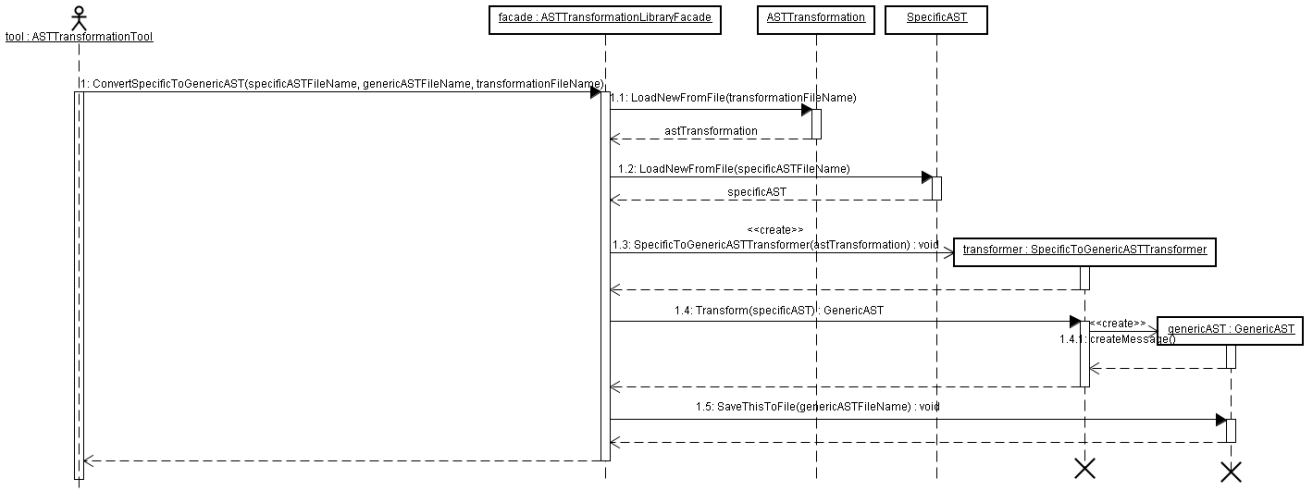
23 pav. „*Konfigūruoti AST medžių transformaciją*“ panaudojimo atvejo sekos diagrama (detalizuota 24 ir 25 pav.)



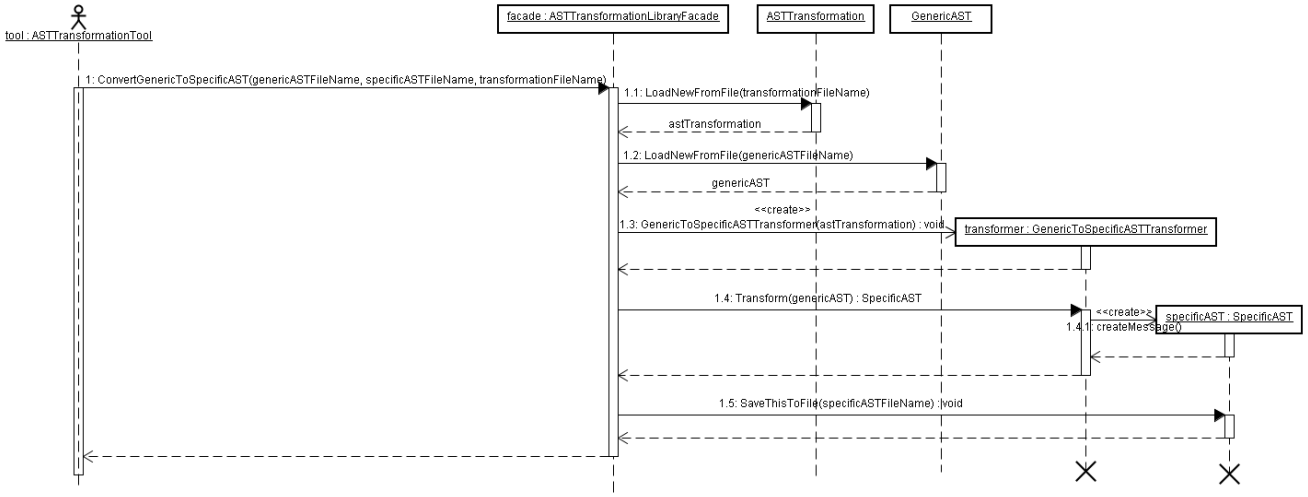
24 pav. „*Konfigūruoti AST medžių transformaciją*“ panaudojimo atvejo sekos diagrama (priartinta, 1 iš 2 dalių)



25 pav. „*Konfigūruoti AST medžių transformaciją*“ panaudojimo atvejo sekos diagrama (priartinta, 2 iš 2 dalių)



26 pav. „Transformuoti konkrečios kalbos AST medį į bendrinį AST medį“ panaudojimo atvejo sekos diagrama



27 pav. „Transformuoti bendrinį AST medį į konkrečios kalbos AST medį“ panaudojimo atvejo sekos diagrama

3.6. Testavimo planas

3.6.1. Testavimo tikslai ir objektai

Sistemos testavimo tikslas yra pašalinti visus programinės įrangos defektus bei kitokius trūkumus. Tačiau šis tikslas pasiektinas tik teoriškai, nes reali programinė įranga visuomet turės bent nedidelį kiekį klaidų. Šio tikslo įgyvendinimą riboja įvairūs veiksniai: ribotas laikas, biudžetas ir pan. Todėl realiai įgyvendinamas tikslas yra sukurti kiek įmanoma geriau ištestuotą ir ištobulintą programinę įrangą.

Pagrindinis siekis yra sukurti pakankamai stabilų programinės įrangos produktą, turintį toleruotiną kiekį loginių klaidų. Taip pat svarbu užtikrinti pakankamą tinkamumą naudoti bei patogumą. Tokie veiksniai, kaip programinės įrangos greitaveika, yra tik antraeilis tikslas, kol kuriama sistema yra prototipo stadijoje.

3.6.2. Testavimo apimtis

Sistema testuojama keliais būdais:

- *Vienetų testavimo*. Šiame etape atliekami baltosios dėžės principu pagrįsti vienetų testai;
- *Integraciniai testais*. Šiame etape atliekami juodosios dėžės principu pagrįsti integraciniai testai;
- *Priėmimo testais*. Šiame etape atliekami testai dalyvaujant programinės įrangos užsakovui;
- *Aukšto lygio testais*. Šiame etape atliekami naudojamumo, apkrovos (stresiniai) bei greitaveikos testai. Atstatymo ir saugumo testai neatliekami, nes jie nėra svarbūs pagal programinės įrangos pobūdį.

Sistema susideda iš dviejų esminių komponentų (AST bibliotekos ir AST transformavimo bibliotekos) ir dviejų pagalbinių įrankių (AST transformacijų konfigūravimo ir AST transformavimo). Vienetų testavimo etape visos dalys testuojamos atskirai, kreipiant ypatingą dėmesį į AST bibliotekos komponentą, nes nuo jo priklauso kitų komponentų darbas. Vėliau visos dalys sujungtos į vieną produktą – tuo metu bus vykdomas integracinis testavimas. Priėmimo bei aukšto lygio testai atliekami sujungtai programiniai įrangai.

Kuriama programinė įranga skirta veikti operacinėse sistemose, palaikančiose Java programavimo kalbą, todėl testavimo darbai atliekami tiek Windows, tiek Linux operacinėse sistemose.

Atliekama bibliotekų bei pagalbinių įrankių dokumentacijos peržiūra, kuria siekiama išsiaiškinti, ar dokumentacija atitinka reikalavimus (Ar aprašytos visos bibliotekų programinio kodo dalys? Ar aprašymai glausti? Ar aprašymai aiškūs? Ar aprašymai pateikti taisyklinga anglų kalba? Ar pateikta informacija, kaip teisingai naudotis įrankiais?).

3.6.3. Pagrindiniai testavimo apribojimai

Sistemos testavimą riboja šie veiksniai:

1) *Laiko apribojimas* – programinės įrangos kūrimui skirtas laiko tarpas, kurio metu vien projektui skiriamas laikas labai ribotas dėl įprastinių studijų bei kitų veiksnių. Dėl šių priežasčių testavimo etapo metu atliekami tik svarbiausi testai, užtikrinant tik ribotą patikimumą.
2) *Programinės įrangos pobūdis* – kuriamas produktas privalo dirbti su įvairių programavimo kalbų AST medžiais, todėl neįmanoma numatyti daugybės su konkrečiomis programavimo

kalbomis susijusių problemų. Testavimai atliekami tik su vienos programavimo kalbos AST medžiais.

3) *Riboti ištekliai* – produkto kūrimui tiesiogiai neskiriamos jokios lėšos, tačiau skiriama programinė įranga bei kompiuteriai. Ilgesniam testavimo periodui prieinamos tik Windows ir Linux operacinės sistemos.

3.6.4. Testavimo strategija

3.6.4.1. Vienetų testavimas

Įprastose situacijose (kai aiški kuriamo kodo realizacija ir yra pakankamai laiko) vienetų testavimas atliekamas testavimais paremto kūrimo (angl. „test driven development“) principais. Atliekami tokie žingsniai:

1) Parašomas testas naujam funkcionalumui;
2) Įsitikinama, jog naujas testas atskleidžia naujo funkcionalumo nebuvimą (kyla testo klaida);
3) Realizuojamas naujas funkcionalumas;
4) Įsitikinama, jog naujas testas patvirtina, jog naujas funkcionalumas išties veikia;
5) Tiek kodas, tiek testai pertvarkomi, kad taptų lengviau skaitomi. Pertvarkymai atliekami mažais žingsniais, po kiekvieno tokio žingsnio iš naujo paleidžiant atitinkamos srities testus.

3.6.4.2. Integravimo testavimas

Pradiniuose etapuose testuojami pagrindinės AST bibliotekos sudedamieji komponentai – bendrinio ir konkretaus AST duomenų struktūros, jų saugojimas į failus ir skaitymas iš jų.

Vėlesniuose etapuose prijungiami AST transformavimo komponento sudedamieji komponentai („subkomponentai“). Galutinis šio etapo rezultatas – sėkmingai atliekamos komponento funkcijos per komponento fasadinę klasę.

Galutiniame etape prijungiami komponentus naudojantys įrankiai.

3.6.4.3. Priėmimo testavimas

Užsakovo ir projekto vadovo akivaizdoje atliekami įvairūs programinės įrangos funkcionalumą tikrinantys testai.

Pirmiausiai AST transformacijų konfigūravimo įrankiu sukuriamas nedidelės apimties konfigūracijos failas. Tuomet AST transformacijų įrankio pagalba naudojamas sukurtasis

konfigūracijos failas ir iš anksto paruoštas AST medis. Tie patys veiksmai atliekami tiek su bendriniu, tiek su konkrečiu AST. Gautieji medžiai palyginami.

3.6.4.4. Aukšto lygio testavimas

3.6.4.4.1. Naudojamumo testavimas

Pirmajame testavimo etape dalyvauja projekto vadovas ir užsakovas. Šių dalyvių paprašoma bandyti naudotis pateiktais įrankiais. Prašoma naudotis įrankių vartotojo dokumentacija, jei kyla naudojimo neaiškumų. Šiame etape stengiami išsiaiškinti, ar įrankiais pakankamai patogu naudotis ir ar pateiktoji dokumentacija pakankamai informatyvi ir tiksli.

Antrajame testavimo etape dalyvauja tik užsakovas. Jis bando integruoti biblioteką į savo programinį kodą ir pateikia atsiliepimus. Kreipiamas dėmesys į naudojimo patogumą, kodo aiškumą bei pateiktos programinio kodo dokumentacijos kokybę.

3.6.4.4.2. Apkrovos (stresinis) testavimas

AST transformacijų įrankiui pateikiami keli ypatingai dideli AST failai ir bandoma atlikti medžių transformacijas. Dalis pateiktųjų medžių failų taisyklingi ir korektiški – jie skirti įvertinti, ar įrankis sugeba dirbti su dideliu informacijos kiekiu. Kita dalis medžių failų turi įvairių klaidų failo gale – jie skirti tikrinti, ar, nuskaičius didelę dalį failo ir užėmus daug sistemos resursų, įrankis teisingai pateikia klaidos žinutę, o ne „nulūžta“.

3.6.4.4.3. Greitaveikos testavimas

Atliekant greitaveikos testavimą tikrinama, kaip greitai įrankių pagalba galima atlikti bibliotekų funkcijas.

Mažiausiai testavimo atliekama su AST transformacijų konfigūravimo įrankiu, nes tikimasi, jog šio įrankio greitis bus ganėtinai pastovus. Taip yra dėl to, kad šis įrankis dirba su ganėtinai nedideliais AST medžių struktūriniais aprašais. Greitaveikos testavimo metu reikia sukurti kuo daugiau AST medžių struktūrinio aprašo elementus jungiančių sąsajų. Stebima, kaip greitai atliekamos AST struktūrinių elementų susiejimo operacijos, tačiau tik po to, kai konfigūracija jau apima apie pusę AST medžių elementų, nes kitu atveju atliekami veiksmai gali būti per greiti, kad juos būtų galima tiksliai išmatuoti.

Vėliau testuojamas AST transformacijų įrankis. Šio įrankio testavimui pateikiamas skirtingo dydžio AST medžių rinkinys. Medžiai transformuojami eilės tvarka ir stebima kiekvienos transformacijos trukmė.

Veiksmų atlikimo trukmes pateikia patys įrankiai. Ši informacija kartais gali būti naudinga ne tik testavimo, bet ir įprastinio naudojimo metu, todėl laiko matavimo funkcionalumą prasminga integruoti.

3.6.4.5. Testavimo resursai

3.6.4.5.1. Techninė įranga

Testavimo darbams naudojamas asmeninis programinės įrangos kūrėjo kompiuteris. Kilus nesklandumams galima atlikti testavimo darbus naudojant universiteto suteikiamus kompiuterius. Labiausiai tikėtini nesklandumai yra nesėkmingas testavimo darbų atlikimas dėl netinkamai sukonfigūruotos Linux operacinės sistemos.

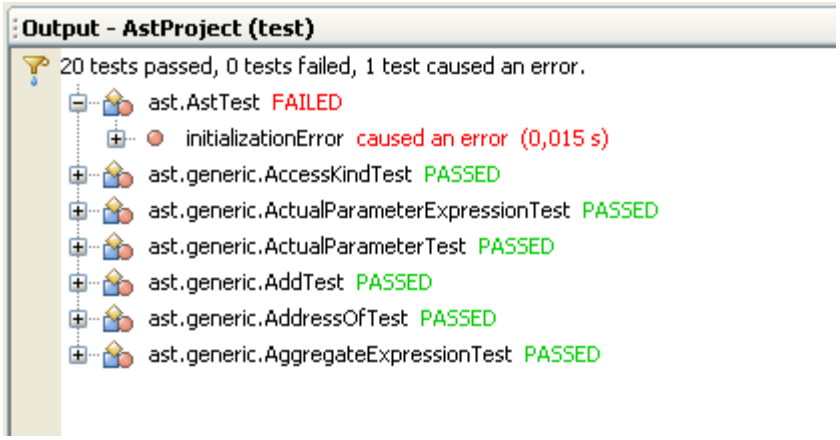
3.6.4.5.2. Programinė įranga

Vienetų ir integravimo testavimui naudojamas JUnit atviro kodo testavimo karkasas. JUnit suteikia galimybę kurti vienetų bei integracinius testus, juos leisti ir stebėti jų rezultatus.

Kadangi JUnit karkaso gražinami rezultatai gali būti pateikiami įvairiais būdais, patogiam testų rezultatų stebėjimui naudojama NetBeans 6.5.1 programavimo aplinka. NetBeans turi specialiai testų rezultatams stebėti skirtą langą (28 pav.), kuris hierarchiškai išdėsto rezultatus pagal kodo išdėstymą paketuose ir leidžia greitai atidaryti testus, kurie buvo nesėkmingi.

Testų rezultatų kaupimui naudojama Excel skaičiuoklė ir Word teksto redaktorius. Rezultatų failai žymimi laiko žyma.

Testavimas vykdomas Windows Vista ir Linux Ubuntu aplinkoje.



28 pav. Testavimo rezultatų pavyzdys

3.6.4.6. Testavimo rezultatai

Vienetų testavimo rezultatai nekaupiami, nes šis testavimas vykdomas lygiagrečiai sistemos kūrimui. Sistemos vienetai testuojami ir testavimo metu aptiktos klaidos taisomos iš karto, sistemos kūrimo proceso metu.

Integravimo ir greitaveikos testavimo rezultatai kaupiami Excel failuose (pavyzdinis integracinio testo rezultatų failas pateiktas 29 pav.).

Priėmimo ir panaudojamumo testų dalyviams skirti klausimai bei jų pastabos kaupiami Word tekstų redaktoriaus failuose.

Testo Nr.	Data	Testo aprašymas	Testo rezultatas	Patabos
1	2009 11 01	Pagalba: pateikiama informacija apie programą	Informacija pateikiama sėkmingai	
2	2009 11 01	Pagalba: pateikiama informacija apie autorių	Blogai pateiktas autoriaus vardas	
3	2009 11 01	Pagalba: pateikiama programos versija	Informacija pateikiama sėkmingai	
4	2009 11 01	Išeiti	Programa sėkmingai baigia darbą	
5	2009 11 01			
6	2009 11 01			
7	2009 11 01			
8	2009 11 01			
9	2009 11 01			
10	2009 11 02			

29 pav. Pavyzdinis integracinių testų rezultatų failas

3.6.5. Vienetų testavimo procedūra

Atliekami tokie pagrindiniai vienetų testavimo etapai:

1) **AST medžių elementų testavimas (tiek bendrinio, tiek konkretaus AST)** – visi medžių elementai išbandomi siekiant išsitikinti, kad pavyksta jiems teisingai priskirti susijusius medžių elementus;

2) **AST medžių elementų užkrovimo/saugojimo į failus testavimas (tiek bendrinio, tiek konkretaus AST)** – visi medžių elementai atskirai tikrinami siekiant įsitikinti, jog jų užkrovimo/saugojimo į failą metodai iškviečia teisingus failų klasės metodus su teisingais parametrais;
3) **AST struktūrinio aprašo elementų užkrovimo/saugojimo į failus testavimas** – struktūriniai AST medžių elementai tikrinami siekiant įsitikinti, jog jų užkrovimo/saugojimo į failą metodai iškviečia teisingus failų klasės metodus su teisingais parametrais;
4) **AST struktūrinio aprašo elementų formuotojų testavimas** – tikrinama, ar AST struktūrinio aprašo elementus kuriantys formuotojai veikia teisingai su pateiktais pavyzdiniais AST medžio elementais, t.y. ar pateikus tam tikrą AST medžio elementą sukuriamas teisingas jį atitinkančio elemento aprašas;
5) **AST transformacijos aprašo testavimas** – tikrinama, ar į transformacijos aprašą galima pridėti papildomus aprašo elementus, juos pašalinti ar rasti.
6) **AST transformacijos aprašo užkrovimo/saugojimo į failus testavimas** – tikrinama, ar atliekant užkrovimą/saugojimą į failą iškviečiami teisingi failų klasės metodai ir su teisingais parametrais;
7) **AST transformatorių testavimas (tiek bendrinių į konkrečius, tiek konkrečių į bendrinius AST)** – tikrinama, ar pateiktus vienokio AST medžio elementus teisingai transformuoja į kitokio AST medžio elementus;
8) **AST medžių transformavimo bibliotekos fasado testavimas** – tikrinama, ar iškvietus atitinkamus fasado metodus kreipiamasi į teisingus kitų klasių metodus su teisingais parametrais;

4. SISTEMOS PROTOTIPO KOKYBĖS ANALIZĖ IR PATOBULINIMAI

4.1. Sistemos prototipo analizė

4.1.1. Prototipo įvertinimas remiantis gero komponento principais

Biblioteką pasirinkta vertinti pagal 5 balų vertinimo sistemą, kiekvienam jos požymiui suteikiant proporcingą įvertinimą, kur 1 yra žemiausia vertinimo reikšmė, o 5 – aukščiausia. Kadangi kuriama biblioteką galima įvardinti kaip komponentą, todėl ją tikslinga įvertinti pagal **[19]** šaltinyje minimus komponentų požymius. Įvertinimas pateiktas 2 lentelėje.

2 lentelė. Sistemos prototipo įvertinimas pagal komponento kokybės kriterijus

Komponento vertinimo kriterijus	**Įvertinimas**	**Įvertinimo pagrindimas**
Galimybė panaudoti pakartotinai	5	Komponentas naudoja daugybę platformų palaikančią programavimo kalbą, turi plačias pritaikymo galimybes (gali tapti daugybės įrankių pagrindų).
Nepriklausomumas nuo konteksto	5	Komponentas kurtas neatsižvelgiant į kitus programinius produktus.
Suderinamumas su kitais komponentais	2	Kuriant biblioteką laikytasi paprastumo, aiškumo ir uždarumo principų. Tačiau komponento kūrėjas neturi rimtos patirties kuriant komponentus.
Stipri inkapsuliacija (realizacijos detalės neatskleidžiamos už komponento ribų).	3	Komponentas nebuvo derinamas prie kitų konkrečių komponentų, todėl suderinamumas priklausys tik nuo atsitiktinumo (t.y. su kokiu kitu komponentu bus bandoma derinti).
Nepriklausomas vystymas ir versijų valdymas	4	Kadangi produktas paremtas Java programavimo kalba, kuri jau pakankamai gerai įsitvirtinusi, todėl jos pokyčiai veikiausiai neturės didelės įtakos produkto vystymo procesui.

4.1.2. Prototipo vertinimas remiantis realiu naudojimu

Pradėjus naudoti sistemos prototipą iškilo žemo sistemos patikimumo problema. Ją lėmė netinkamas medžių transformavimo algoritmas. Dalinės medžių transformacijos (transformacijų taisyklės) veikė gerai, tačiau dažnai šių dalinių transformacijų rezultatus į vieną pilną transformaciją sujungdavo netinkamai.

Algoritme nebuvo numatyta patikimo būdo konfigūracijos failuose tinkamai nurodyti dalinių transformacijų sujungimo taškus. Sujungimas būdavo atliekamas analizuojant jau sukurtą galutinio AST dalį ir joje ieškant elementų pagal reikiamą tipą, t.y. būdavo ieškomi elementai, kurie sugebėdavo talpinti naujai sukurtas AST dalis kaip vaikinius elementus.

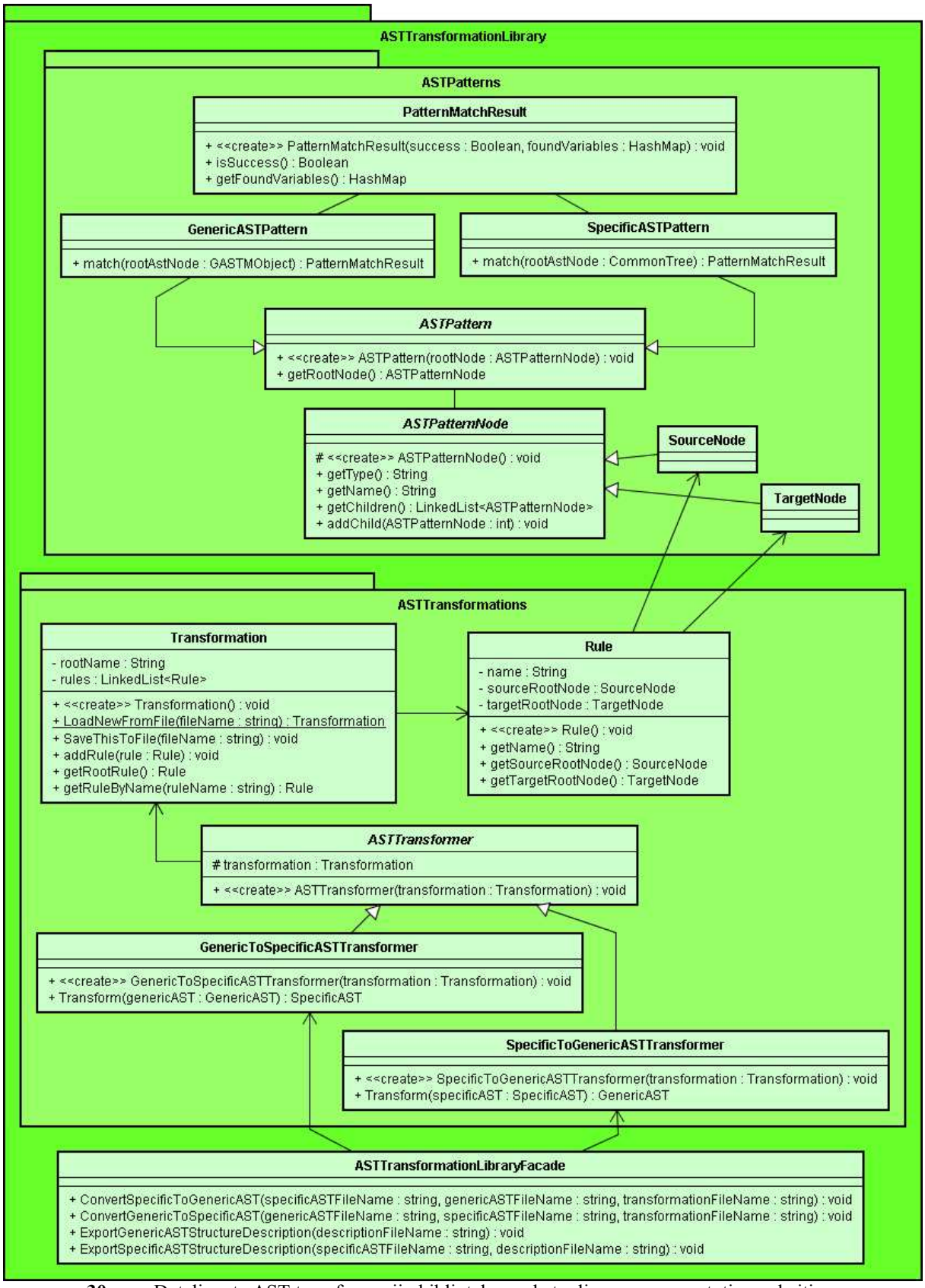
4.2. Sistemos prototipo patobulinimai

Dėl netinkamo sistemos veikimo (prasto transformavimo algoritmo) teko atlikti daugybę pakeitimų.

Buvo atlikta detalesnė panašių sistemų algoritmų analizė. Analizė, kuri buvo atlikta prieš kuriant sistemos prototipą, buvo labiau orientuota į egzistuojančių panašių priemonių savybių palyginimą, per mažai dėmesio skiriant jų realizacijos detalėms. Pasirinktasis prototipo algoritmas nebuvo pakankamai išanalizuotas.

Analizės metu aptikta informacija padėjo suformuoti naujosios sistemos versijos projektą. Sistemoje buvo atlikti ganėtinai nemaži architektūriniai, algoritmų realizacijų bei transformacijų failų formato pakeitimai. Kiekvienas iš šių pakeitimų detaliau aptariamas tolesniuose skyreliuose.

4.2.1. Sistemos architektūros patobulinimai



30 pav. Detalizuota AST transformacijų bibliotekos paketo diagrama po prototipo pakeitimų

Naujai perdarius `ASTTransformationLibrary` paketą (30 pav.), iš jo buvo išmestas vidinis `ASTStructure` (21 pav.) paketas, kuris jau nebereikalingas, ir buvo pridėtas `ASTPatterns` paketas. Trumpai apie paketus:

- `ASTPatterns`. Jį sudaro klasės, aprašančios AST elementų sekas bei realizuojančios sekų radimo medžiuose algoritmus. Pagrindinės paketo klasės:
    - `ASTPattern`. Tai bazinė AST elementų sekų aprašymo klasė. Pagrindinis jos metodas `match` atlieka sekų radimo funkciją ir grąžina `PatternMatchResult` klasės objektą, nurodantį bandymo sutapatinti rezultatus.
    - `GenericASTPattern`. Tai bendrinio AST elementų sekų aprašymo klasė.
    - `SpecificASTPattern`. Tai konkretaus AST elementų sekų aprašymo klasė.
    - `PatternMatchResult`. Tai klasė, aprašanti rezultatą bandymo rasti tam tikrą AST elementų seką. Klasė grąžina loginę reikšmę, parodančią, ar paieška sėkminga, bei grąžina rastų kintamųjų reikšmių lentelę.
    - `ASTPatternNode`. Tai klasė, kuri talpina informaciją apie AST sekos elementą. Joje laikomas elemento vardas, tipas bei vaikiniai `ASTPatternNode` klasės elementai.
    - `SourceNode`. Tai klasė, talpinanti informaciją apie AST sekos elementą, kurio ieškoma pradiniame AST medyje.
    - `TargetNode`. Tai klasė, talpinanti informaciją apie AST sekos elementą, kuris sukuriamas po to, kai randama ieškota `SourceNode` elementų tipo seka.
- `ASTTransformations`. Kaip ir prototipe, šį paketą sudaro AST transformacijas aprašančios bei transformacijas atliekančios klasės. Pagrindinės paketo klasės:
    - `Rule`. Tai transformavimo taisyklę aprašanti klasė. Klasė turi pavadinimą bei dviejų sekų šakninius elementus. Pirmosios sekos šakninis elementas bei jo vaikiniai elementai žymi seką, kurią reikia rasti pradiniame AST medyje, o antrosios – seką (kuriamo AST medžio duomenų struktūrą), kurią reikia sukurti, jei buvo rasta pradinė seka.
    - `Transformation`. Tai transformavimo taisyklių rinkinį talpinanti klasė. Klasė turi nuorodą į šakninę transformacijos taisyklę bei metodus taisyklių pridėjimui į transformaciją bei paieškai pagal pavadinimą.
    - `ASTTransformer`. Tai bazinė AST transformacijas atliekančių algoritmų realizavimo klasė.

- `GenericToSpecificASTTransformer`. Tai klasė, atliekanti transformacijas iš bendrinių AST į konkrečius AST.
- `SpecificToGenericASTTransformer`. Tai klasė, atliekanti transformacijas iš konkrečių AST transformacijas į bendriniu AST.

4.2.2. Sistemos algoritmų patobulinimai

Naujojoje sistemos versijoje buvo pakeisti sekų paieškos, užklausų vykdymo bei transformavimo algoritmai. Pagrindinis dėmesys skirtas sekų paieškos ir užklausų vykdymo algoritmui, nes būtent jis suteikia galimybę patogiai aprašyti AST transformacijas. Esminis naujojo algoritmo privalumas – galimybė aiškiai nurodyti, kurie pradinio medžio elementai bus naudojami naujojo medžio kūrime kaip kintamieji.

Nors transformavimo algoritmo efektyvumas naujoje sistemos versijoje ganėtinai išaugo, tą labiausiai lėmė tik vienintelis pakeitimas – algoritme pridėta galimybė sujungti dalines transformacijas. Tai realizuota panaudojus rastuosius kintamuosius kaip dalinius pradinius AST medžius, kuriem rekursiškai taikomos vis naujos dalinės transformacijos.

4.2.2.1. Sekų paieškos ir užklausų vykdymo algoritmo patobulinimai

Naujasis sekų paieškos ir užklausų vykdymo algoritmas paremtas **[18]** šaltinyje nurodytu metodu. Pateiktasis metodas skirtas sulyginti ieškomas sekas su pateiktąja duomenų struktūra (šiuo konkrečiu atveju – AST medžiu) ir, jei sulyginimas sėkmingas, grąžinti atitinkamą sėkmės žymę ir sąrašą duomenų struktūros elementų, kurie sekoje buvo pažymėti kaip kintamieji (žiūrėkite 2.4.3 skyrelyje pateiktą pavyzdį). Sistemai pritaikyta algoritmo versija pateikta pseudo-kodo formatu pateikta žemiau (31 pav.), o pilna realizacija Java programavimo kalba – 1-ajame priede.

Pradedant sekos lyginimą su pateiktu AST medžiu iškviečiamas metodas `Sulyginti`. Perduodami argumentai: šakninis naudojamos sekos elementas (`SekosElementas`), šakninis pradinio AST elementas (`ASTElementas`) bei naujai sukurta lentelė, skirta saugoti randamus kintamuosius (`RastiKintamieji`). Sulyginimas vyksta pagal tipą. Po sėkmingo elemento sulyginimo atliekamas patikrinimas, ar AST elementas turi būti išsaugotas kaip kintamasis, ir, jei reikia, atliekami atitinkami išsaugojimo veiksmai.

Vėliau atliekamas perėjimas per vaikinius tiek sekos, tiek paties AST elementus. Sulyginimas laikomas sėkmingu, kol kiekvienam sekos elementui pavyksta rasti atitinkamą AST elementą. Jei tam tikru momentu tikrinimas sekos elementas yra sąrašo tipo, ieškoma bent vieno

AST elemento, atitinkančio šį sekos elementą. Priešingu atveju tuo momentu nagrinėjama AST ir sekos elementų pora toliau rekursiškai sulyginama iškviečiant tą patį metodą `Sulyginti`.

Metodui baigus darbą grąžinama loginė reikšmė, kuri rodo, ar sulyginimas sėkmingas. Sulyginimo metu aptikti kintamieji saugomi parametre `RastiKintamieji` (parametro reikšmės grąžinti iš metodo nereikia, nes pateiktasis argumentas yra tik nuoroda į parametro turinį, angl. „reference“).

```
function Sulyginti (SekosElementas, ASTElementas, RastiKintamieji)
  if SekosElementas.Tipas = ASTElementas.Tipas then

    if SekosElementas.YraKintamasis then
      RastiKintamieji.Prideti(SekosElementas.KintamojoVardas, ASTElementas)
    end

    if SekosElementas.TuriVaiku then
      if ASTElementas.TuriVaiku then
        ArSulyginimasSėkmingas := true

        while ArSulyginimasSėkmingas = true and SekosElementas.TuriDarNeaplankytuVaiku do
          if ASTElementas.TuriDarNeaplankytuVaiku then
            VaikinisSekosElementas := SekosElementas.ImtSekantiNeaplankytaVaika()
            VaikinisASTElementas := ASTElementas.ImtSekantiNeaplankytaVaika()

            if VaikinisSekosElementas.YraSarasoTipo then
              while ASTElementas.TuriDarNeaplankytuVaiku do
                if VaikinisSekosElementas.Tipas = VakinisASTElementas.Tipas or
                   VaikinisSekosElementas.ArSvarbusTipas = false
                then
                  if RastiKintamieji.Turi(VaikinisElementas.KintamojoVardas) then
                    SąrašoKintamasis := RastiKintamieji.
                                        ImtKintamaji(VaikinisElementas.KintamojoVardas)

                    SąrašoKintamasis.Pridėti(VaikinisASTElementas)
                  else
                    SąrašoKintamasis := SukurtiTuščiąSąrašoKintamąjį()
                    SąrašoKintamasis.Pridėti(VaikinisASTElementas)
                    RastiKintamieji.Prideti(VaikinisSekosElementas.KintamojoVardas,
                                            VaikinisASTElementas)
                  end
                  return true
                else
                  break // Nutraukiamas vidinis ciklas
                end

                VaikinisASTElementas := ASTElementas.ImtSekantiNeaplankytaVaika()
              end
            else
              ArSulyginimasSėkmingas := Sulyginti(VaikinisSekosElementas,
                                                  VaikinisASTElementas,
                                                  RastiKintamieji)
            end
          else
            return false
          end
        end
        return ArSulyginimasSėkmingas
      else
        return false
      end
    else
      return true
    end
  else
    return false
  end
end
```

31 pav. Konkretaus AST sekos radimo algoritmas, išreikštas pseudo-kodu

4.2.2.2. Transformacijų algoritmo patobulinimai

Naujasis AST transformavimo algoritmas nuo senojo skiriasi papildoma galimybe tiesiogiai nurodyti transformavimo taisyklių sujungimo taškus. Pilna algoritmo realizacija Java kalba pateikta 2-ajame priede.

Pradedant transformavimo procesą iš transformacijos aprašo (prieš pradedant transformaciją šį aprašą reikia užkrauti iš pateiktojo XML failo) paimama šakninė taisyklė. Ji nusako šakninio AST elemento transformaciją. Jei taisyklėje aprašyta pradinio medžio seka sutampa su pateiktuoju AST, sukuriama taisyklėje aprašyta galutinio AST dalis.

Kuriant šią naująją dalį, kiekvienam naujojo AST elementui gali būti priskiriamos atributų (AST elementą aprašančios klasės kintamųjų) reikšmės. Atributų reikšmės gali būti nurodytos tiesiogiai arba panaudojant išraiškas (detalesnė informacija pateikta 4.2.3 skyrelyje).

Be įprastų naujojo AST elementų kūrimo – nurodant jų tipą, pavadinimą ir atributus – yra galimybė nurodyti, kad tam tikrą naujojo AST atšaką sukurtų kita taisyklė. Nurodant taisyklę tereikia pateikti jos vardą ir kintamąjį, kurio turinys bus naudojamas kaip pradinis AST šios taisyklės transformavimo veiksmams.

Transformavimo procesas baigiamas po to, kai atliktos visos įmanomos dalinės transformacijos, pradedant šaknine, kuri rekursiškai kreipiasi į kitas. Proceso rezultatas – pilnai sukurtas naujas AST arba tuščia reikšmė (`null`), jei medžio nepavyko sukurti dėl pradiniame AST nerastos elementų sekos.

4.2.3. AST transformacijų aprašų failų formato pakeitimai

Senąjį AST transformacijų aprašo formatą (32 pav.) sudarė dviejų tipų elementai:

- `ASTTransformation` – tai šakninis XML elementas, apimantis naudojamas transformavimo taisykles.
- `Map` – tai transformavimo taisyklės aprašo XML elementas. Jis turi du atributus: `from` ir `to`. Pirmasis atributas parodo pradinio AST elementų seką, kurią radus kuriama seka, aprašyta antrajame atribute. Sekos abiejuose atributuose aprašytos atskiriant kiekvieną medžio elementą taškais. Pirmiau einantis elementas yra toliau einančio elemento tėvinis elementas. Aprašymo formatas šiek tiek panašus į CSS stilių aprašo formatą **[20]**.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<ASTTransformation>
    <Map from="JAVA_SOURCE" to="CompilationUnit"/>
    <Map from="CLASS" to="AggregateTypeDefinition.ClassType@{AggregateType}"/>
    <Map from="CLASS.IDENT.#value" to="AggregateTypeDefinition.Name@{Name}.NameString"/>
    <Map from="CLASS_TOP_LEVEL_SCOPE" to="AggregateScope"/>
    <Map from="VOID_METHOD_DECL" to="FunctionDefinition.TypeReference(*)@{ReturnTypes}.UnnamedTypeReference.GastmVoid@{Type}"/>
    <Map from="VOID_METHOD_DECL.MODIFIER_LIST.PUBLIC" to="FunctionDefinition.Public@{AccessKind}"/>
    <Map from="IDENT.#value" to="Name.NameString"/>
    <Map from="FORMAL_PARAM_LIST" to="FormalParameterDefinition(*)"/>
    <Map from="VOID_METHOD_DECL.BLOCK_SCOPE" to="FunctionDefinition.FunctionScope@{OpensScope}"/>
    <Map from="VAR_DECLARATION" to="DeclarationOrDefinitionStatement.VariableDefinition@{DeclOrDefn}"/>
    <Map from="EXPR.STRING_LITERAL.#value" to="StringLiteral.Value"/>
</ASTTransformation>
```

32 pav. AST transformacijų aprašo failo, skirto sistemos prototipui, pavyzdys

Naująjį AST transformacijų aprašo formatas (33 pav.) sudaro tokie elementų tipai:

- `Transformation` – tai šakinis XML elementas, apimantis naudojamas transformavimo taisykles. Elementas turi atributą `root`, kuris nusako, kokią taisyklę pritaikyti šakiniam pradinio medžio elementui.
- `Rule` – tai transformavimo taisyklės aprašo XML elementas.
- `Source` – tai šakninis elementas, skirtas aprašyti pradinio medžio sekai, kuri yra ieškoma.
- `Target` – tai šakinis XML elementas, skirtas aprašyti galutinio medžio sekai, kuri yra kuriama, jei pradinė seka buvo rasta pateiktame medyje.
- `Node` – tai AST medžio elementą aprašantis XML elementas. Jis turi atributus, žyminčius AST elemento tipą (`type`) ir pavadinimą (`name`). Nurodant tipą galima pateikti tikslų tipo pavadinimą specialią žymę („*“), reiškiančią bet kokį tipą. Jei elementas naudojamas kaip `Source` elemento vaikas, papildomai jis gali turėti atributą, rodantį, jog nurodytasis AST elementas bus naudojamas kaip kintamasis (`var`). Tuomet šio atributo reikšmė yra suteiktasis kintamojo pavadinimas.
- `NodeList` – tai AST medžių elementų sąrašą aprašantis XML elementas. Jo sandara panaši į `Node` elementą. Įprastai šis XML elementas naudojamas kartu su vaikiniais elementais, kurie reiškia AST elementų sąrašą, tačiau galima nurodyti, jog sąrašo vaikinius AST elementus sugeneruos tam tikrą kitą transformavimo taisyklė. Tuomet pridedami du atributai: `useRule` ir `sourceVar`. Pirmasis rodo, kokią transformavimo taisyklę naudoti kuriant sąrašo elementus, o antrasis – kurio kintamojo turinį pateikti taisyklei. Beje, `useRule` atributo reikšmė gali būti sudaryta iš kelių taisyklių pavadinimų, atskirtų vertikaliais brūkšniais („|“). Tuomet atliekant transformaciją paeiliui bandoma pritaikyti po taisyklę iš pateiktojo atributo tol, kol bus sutikta taisyklė, kurią pavyko pritaikyti.
- `Property` – tai AST medžio elemento atributo (AST elementą aprašančios klasės kintamojo) reikšmę nusakantis elementas. Jis turi pavadinimo (`name`) bei reikšmės (`value`) XML atributus. Reikšmės atribute nurodoma tiesioginė reikšmė arba tarp specialių žymių („${“ ir „}“) pateiktą išraiška. Išraiškoje nurodomas kintamojo vardas bei kintamojo atributo pavadinimas (čia kintamasis yra AST medžio elemento tipo, o jo atributas – klasės kintamasis), abu atskirti tašku.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Transformation root="SourceRootRule">
    <Rule name="SourceRootRule">
        <Source>
            <Node type="JAVA_SOURCE">
                <Node type="ANNOTATION_LIST"/>
                <NodeList type="CLASS" var="var_class_definition_list"/>
            </Node>
        </Source>
        <Target>
            <Node type="CompilationUnit">
                <NodeList name="Fragments" sourceVar="var_class_definition_list" useRule="ClassRule" type="DefinitionObject"/>
                <Property name="Language" value="Java"/>
            </Node>
        </Target>
    </Rule>
    <Rule name="ClassRule">
        <Source>
            <Node type="CLASS">
                <Node type="MODIFIER_LIST"/>
                <Node type="IDENT" var="var_class_name"/>
                <Node type="CLASS_TOP_LEVEL_SCOPE"/>
            </Node>
        </Source>
        <Target>
            <Node type="AggregateTypeDefinition">
                <Node type="Name">
                    <Property name="NameString" value="${var_class_name.Text}"/>
                </Node>
                <Node name="AggregateType" type="ClassType"/>
            </Node>
        </Target>
    </Rule>
</Transformation>
```

33 pav. AST transformacijų aprašo failo, skirto naujai sistemos versijai, pavyzdys

4.3. Idėjos tolesniam įrankių tobulinimui

4.3.1. Automatinė transformacijų konfigūravimo funkcija

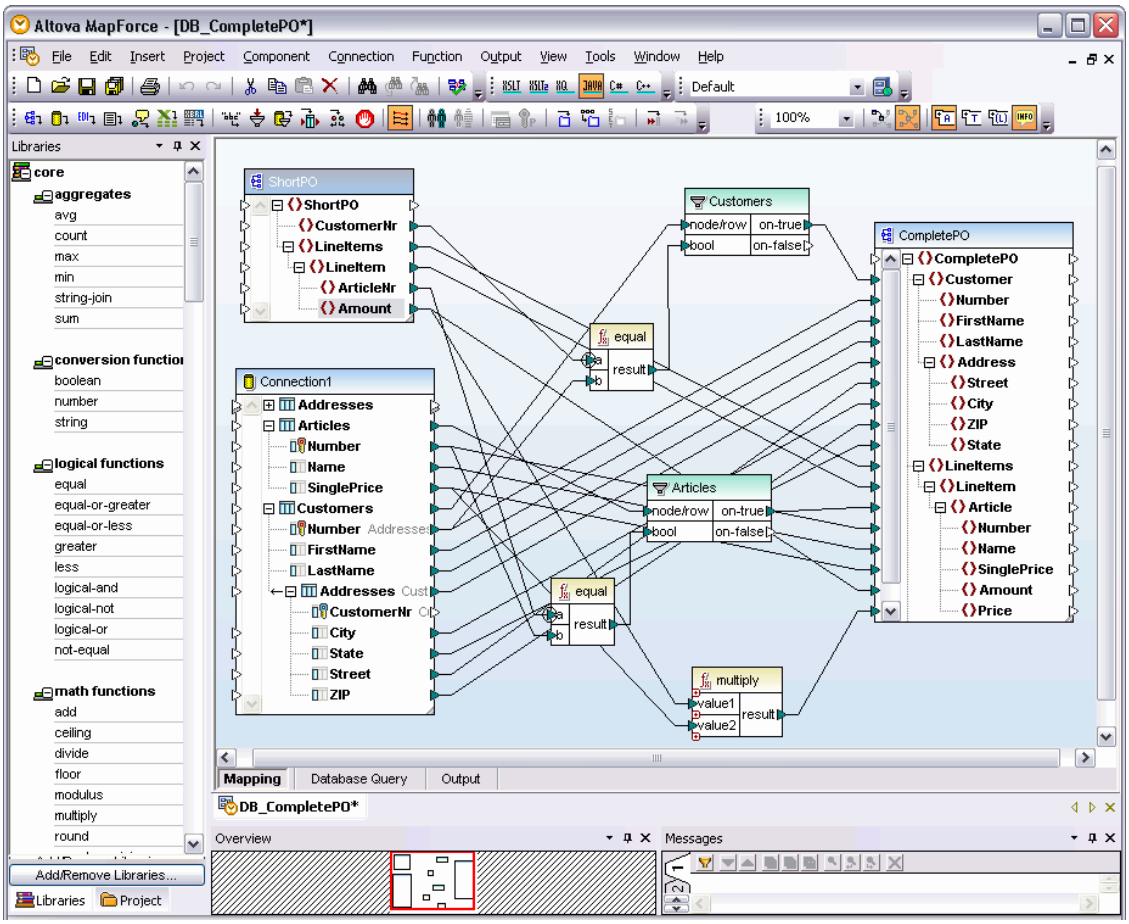
AST medžių transformacijų konfigūravimo įrankis turėtų leisti automatiškai nustatyti transformacijų parametrus tiems medžių elementams, kurie vienas kitą visiškai atitinka. Jei bendriniame medyje būtų elementas, kuris savo forma identiškas konkrečiam medžiui, tuomet būtų nustatoma, jog abu šie elementai atitinka.

Kadangi atitikimas aptinkamas tik pagal elemento formą, todėl tai nebūtinai turėtų būti semantine prasme atitinkantys elementai. Todėl būtina leisti vartotojui redaguoti šiuos transformacijų nustatymus.

4.3.2. Grafinės sąsajos patobulinimas

Ateityje galima bandyti sukurti tobulesnę AST medžių konfigūravimo įrankio versiją, kuri leistų grafiškai atlikti medžių konfigūravimą. Šiame įrankyje būtų vaizduojami bendrinis ir konkrečios kalbos medžiai kaip tam tikri elementai su įėjimais ir/ar išėjimais. Tuomet būtų galima nurodyti, kuris vieno medžio išėjimas atitinka kurį kito medžio įėjimą. Būtų nurodoma, kuri vieno medžio struktūra atitinka kurią kito medžio struktūrą. Tai turėtų būti kažkas panašaus į Altova

firmos MapForce įrankį (34 pav.), leidžiantį atlikti duomenų transformacijas iš vieno šaltinio į kitą **[21]**.



34 pav. Altova MapForce įrankio grafinė sąsaja

5. PATOBULINTOS SISTEMOS ĮVERTINIMAS EKSPERIMENTU

5.1. Eksperimento tikslai

Šiuo eksperimentu siekiama parodyti, jog AST transformavimo biblioteka tinkama kasdieniniam naudojimui ir pasižymi pakankamai geromis charakteristikomis, jog galėtų tapti kitų įrankių pagrindu. Dėl to būtina įrodyti, kad:

1) Biblioteka veikia teisingai;
2) Biblioteka veikia pakankamai greitai.

Pirmasis tikslas yra pats svarbiausias. Jei pateikiamas taisyklingai suformuotas pradinis AST medis ir taisyklingai suformuotas transformacijos aprašas, biblioteka privalo sugeneruoti naują AST medį. Naujasis medis turi būti sukurtas laikantis transformacijos aprašo taisyklių. Transformacijos privalo veikti tiek transformuojant konkretų AST į bendrinį, tiek atvirkščiai.

Antrasis tikslas svarbus dviem aspektais. Pirmiausia svarbu įsitikinti, jog net ir veikdama korektiškai, biblioteka grąžins rezultatus per „protingą“ laiko tarpą, t.y. ar neveiks taip lėtai, jog tą akivaizdžiai matys vartotojas, naudojantis nedidelės apimties medžius (dėl papildomo paaiškinimo žiūrėkite 3.3.4.3 skyrelį). Tuomet svarbu ištirti dėsnius, pagal kuriuos kinta operacijų vykdymo trukmė kai didėja duomenų kiekis. Jei nedidelis duomenų kiekio prieaugis bibliotekos darbą sulėtina šimtus ar net tūkstančius kartų, ji praktiškai beveik nenaudinga.

5.2. Eksperimento atlikimo metodika

Eksperimentą sudaro du etapai – po vieną abiems eksperimento tikslams pasiekti (įrodyti veikimo teisingumą ir pakankamai gerą greitaveiką).

Pirmajame etape AST transformavimo bibliotekai pateikiamas konkretus AST, sudarytas iš nedidelio Java kalba parašyto išeities teksto fragmento, ir transformacijos aprašas, kuriame aprašytos transformacijos taisyklės, sugebančios transformuoti visus pradinio medžio elementus. Įvykdžius AST transformaciją gautasis bendrinis medis išsaugomas į rezultatų failą tekstiniu formatu (medį galima saugoti ir XML formatu, tačiau tekstinis formatas mažiau apkrautas ir lengviau skaitomas). Kadangi sukurtasis medis ne itin didelis, jis tikrinamas rankiniu būdu. Po to pagal tą pačią tvarką atliekama atvirkštinė transformacija. Naujai gautas konkretus AST tikrinimas tiek rankiniu, tiek automatiniu būdu. Automatinis sulyginimas vyksta vienu metu rekursiškai pereinant abiejų konkrečių AST elementus ir lyginant jų tipus bei tekstines reikšmes.

Antrajame etape naudojamas eksperimentui sukurtas Java kalbos kodo generatorius, kuris pagal įėjimo parametrus sukuria norimą kiekį kodo, sudaryto iš kelių dažniau naudojamų kalbos konstrukcijų. Iš kodo kuriamas konkretus AST, transformuojamas į bendrinį AST ir atgal. Abiejų transformacijų metu matuojama jų trukmė. Transformavimo procesas kartojamas daug kartų, su vis didėjančiu kodo kiekiu. Rezultatai kaskart išsaugomi į lentelę, pagal kurią braižomi grafikai, palengvinantys algoritmo spartos dėsningumų suvokimą. Beje, transformavimo procesui pateikiamas tas pats transformacijos aprašo failas, kuris naudojamas pirmajame etape.

5.3. Eksperimento etapų vykdymas

5.3.1. Transformavimo algoritmų korektiškumo tyrimas

AST transformavimo korektiškumui patikrinti buvo pasirinktas išeities tekstas, kurį sudaro dvi dažnai naudojamos Java kalbos konstrukcijos – klasės ir metodai (35 pav.). Nors konstrukcijos tik dvi, pilnai transformacijai reikalingos net penkios taisyklės (konkretaus AST transformacijos į bendrinį AST aprašas pateiktas 3-iajame priede):

- `SourceRootRule`. Ši taisyklė aprašo šakninių medžių elementų transformaciją. Pateiktajame medyje rastus klasių aprašo šaknynius elementus ji perduoda `ClassWithMethodsRule` taisyklei.
- `ClassWithMethodsRule`. Ši taisyklė aprašo klasės, turinčios bent vieną metodą, transformaciją. Rastus metodus ji perduoda apdoroti kitoms taisyklėms. Kadangi `void` ir `int` tipo metodų šakniniai elementai skiriasi, skiriasi ir juos apdorojančios taisyklės – `ProcedureRule` ir `IntFunctionRule`.
- `ClassWithoutMethodsRule`. Ši taisyklė panaši į `ClassWithMethodsRule`, tačiau neįtraukia metodus aprašančių elementų, todėl yra šiek tiek trumpesnė. Naudojant abi klases aprašančias taisykles, svarbu pirma nurodyti `ClassWithMethodsRule`, nes jos pradinė seka yra detalesnė, todėl paieškos kriterijus yra griežtesnis. `ClassWithoutMethodsRule` suveikia nepriklausomai nuo to, ar metodų yra, ar ne.
- `ProcedureRule`. Ši taisyklė aprašo nieko negrąžinančių `void` tipo metodų transformaciją.
- `IntFunctionRule`. Ši taisyklė aprašo `int` tipo reikšmę grąžinančių metodų transformaciją.

```
class MyFirstClass {
    void methodWithVoidReturnType() {
    }
    int methodWithIntReturnType() {
    }
}

class MySecondClass {
}
```

35 pav. Bandomasis Java kalbos išeities tekstas, skirtas AST transformavimo algoritmų korektiškumo tyrimui

Panaudojus šį trumpą išeities tekstą buvo sukurtas konkretus AST (priedas nr. 4), transformuotas į bendrinį (priedas nr. 5) ir, naudojant atvirkščią transformaciją (priedas nr. 6), vėl buvo sukurtas konkretus AST. Palyginus abu konkrečius AST nerasta jokių skirtumų. Abi transformacijos įvykdytos sėkmingai.

5.3.2. Transformavimo algoritmų greitaveikos tyrimas

Atliekant pirmąjį eksperimento etapą nebuvo aptikta akivaizdžių bibliotekos greitaveikos problemų, todėl pradėtas duomenų kiekio ir darbo trukmės priklausomybės tyrimas.

Nuspręsta panaudoti tuos pačius AST transformacijų aprašus, kurie naudoti pirmajame etape, nuolat didinant duomenų kiekius su specialiai šiam tyrimui sukurtu išeities tekstų generatoriumi. Generatorius priima tris parametrus, nuo kurių priklauso sugeneruoto teksto turinys: klasių kiekis, metodų kiekvienoje klasėje kiekis bei metodų rezultatų tipas (`void` arba `int`). Tyrimo metu dviem iš šių parametrų suteikiamos pastovios reikšmes, o trečiasis keičiamas.

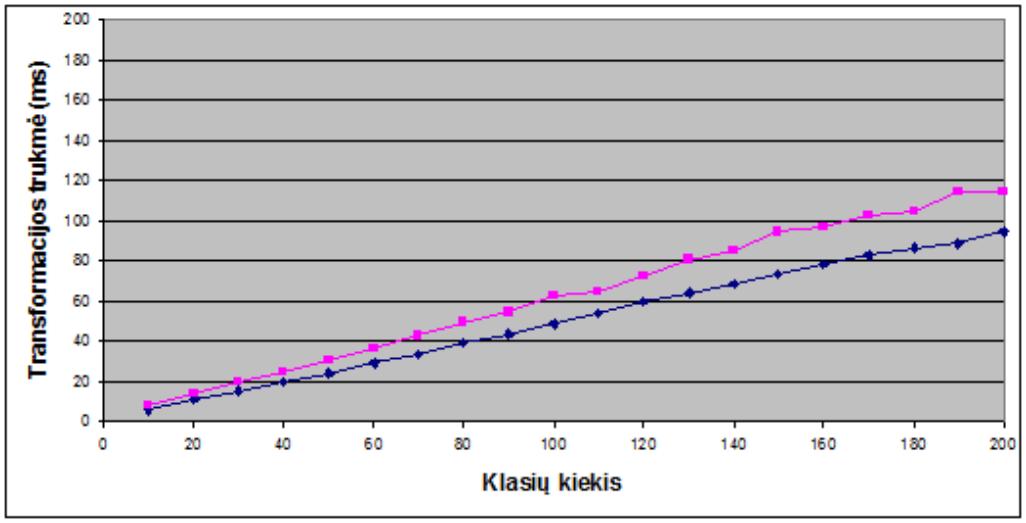
Tyrimo metu naudoti žingsniai aprašyti 3-ioje lentelėje. Joje pateiktas parametras n įgyja reikšmę iš intervalo tarp 10 ir 200, kiekvieno iteracijoje didinant reikšmę dešimtimi (10, 20, 30, ..., 190, 200). Su kiekvienu iš šių parametrų transformacijos atliekamos po 200 kartų. Matuojamas bendras laikas ir padalinamas iš atliktų kartų skaičiaus. Taip gaunamas vidutiniška kiekvieno vykdymo trukmė. Taip siekiama sumažinti trukmės skaičiavimo paklaidą, nes vienas transformacijos vykdymas įvyksta per greit, kad būtų galima pakankami tiksliai išmatuoti jos trukmę.

Pagal lentelėje aprašytus žingsnius buvo atlikti AST transformacijų bandymai tiek iš konkrečių AST į bendrinius, tiek ir atvirkštine kryptimi. Kiekvieno iš šių žingsnių rezultatai pateikti 36-40 paveikslėliuose grafikų forma. Abscisėje pateikiami klasių ar metodų kiekiai, o ordinatėje – transformavimo algoritmų trukmė milisekundėmis. Tamsiai mėlyna spalva pažymėti grafikai rodo transformacijų trukmę iš konkretaus AST į bendrinį, o rožine – iš bendrinio AST į konkretų.

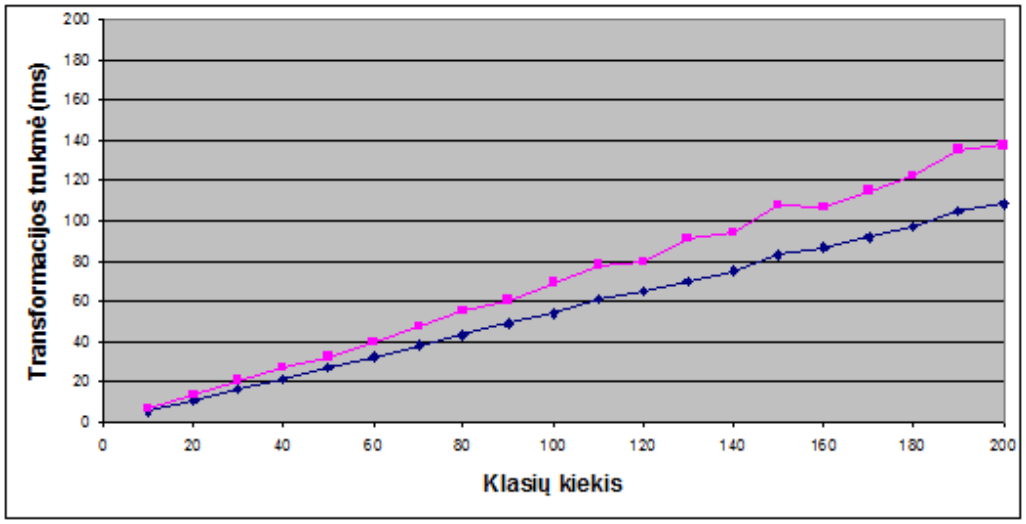
3 lentelė. Transformavimo algoritmų greitaveikos tyrimo žingsniai

Žingsnio nr.	**Klasių skaičius**	**Metodų, tenkančių vienai klasei, skaičius**	**Metodo rezultatų tipas**	**Aprašymas**
1	n	0	Nesvarbu	*Generuojamos klasės be metodų.* Transformavimo taisyklės turėtų būti lankomos tokia seka: 1) `SourceRootRule`; 2) `ClassWithMethodsRule`; 3) `ClassWithoutMethodsRule`. Klasė metodų neturi, todėl `ClassWithMethodsRule` taisyklės sekos sulyginimas su pateiktu AST nepavyktų, taip pereinant prie `ClassWithoutMethodsRule`.
2	1	n	`void`	*Generuojama viena klasė su daugybe `void` metodų.* Transformavimo taisyklės turėtų būti lankomos tokia seka: 1) `SourceRootRule`; 2) `ClassWithMethodsRule`; 3) `ProcedureRule`. Visų šių taisyklių aplankymas būtų sėkmingas, t. y. visų taisyklių sekos sutaptų su tuo transformavimo metu lankomais elementais.
3	1	n	`int`	*Generuojama viena klasė su daugybe `int` metodų.* Transformavimo taisyklės turėtų būti lankomos tokia seka: 1) `SourceRootRule`; 2) `ClassWithMethodsRule`; 3) `ProcedureRule`; 4) `IntFunctionRule`. Taisyklės `ProcudureRule` lankymas būtų nesėkmingas, todėl būtų pereita prie `IntFunctionRule`.
4	n	1	`void`	*Generuojama daug klasių, turinčių po vieną `void` metodą.* Transformavimo taisyklės turėtų būti lankomos tokia seka: 1) `SourceRootRule`; 2) `ClassWithMethodsRule`; 3) `ProcedureRule`. Visų šių taisyklių aplankymas būtų sėkmingas, t. y. visų taisyklių sekos sutaptų su tuo transformavimo metu lankomais elementais.

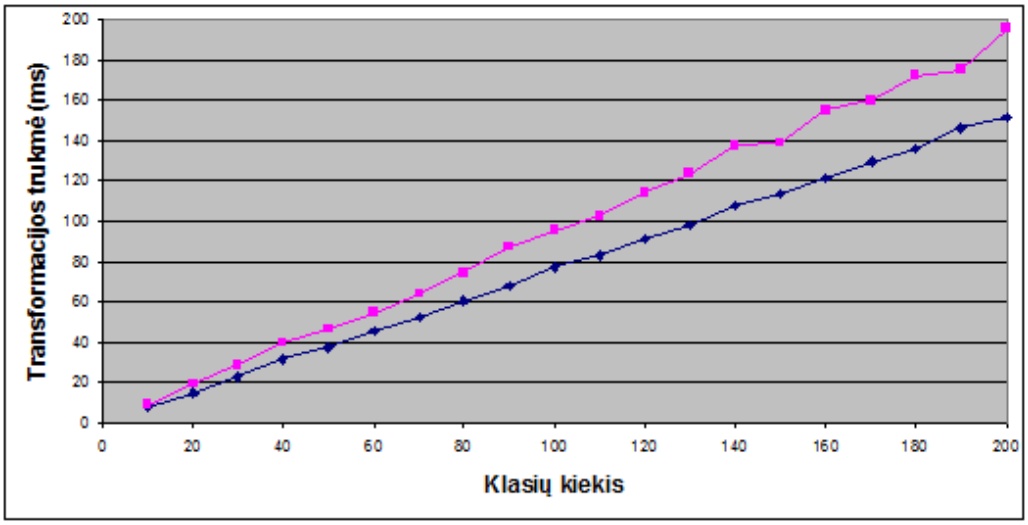
5	n	1	`int`	*Generuojama viena klasė su daugybe* `int` *metodų.* Transformavimo taisyklės turėtų būti lankomos tokia seka: 1) `SourceRootRule`; 2) `ClassWithMethodsRule`; 3) `ProcedureRule`; 4) `IntFunctionRule`. Taisyklės `ProcedureRule` lankymas būtų nesėkmingas, todėl būtų pereita prie `IntFunctionRule`.



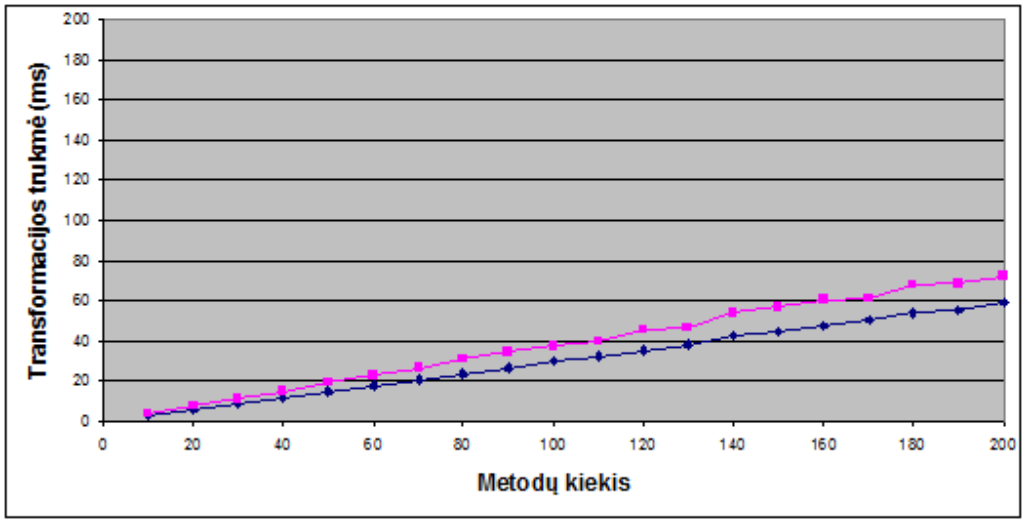
36 pav. AST transformavimo algoritmų greitaveikos grafikas (klasių kiekis – kintamas, metodų nėra)



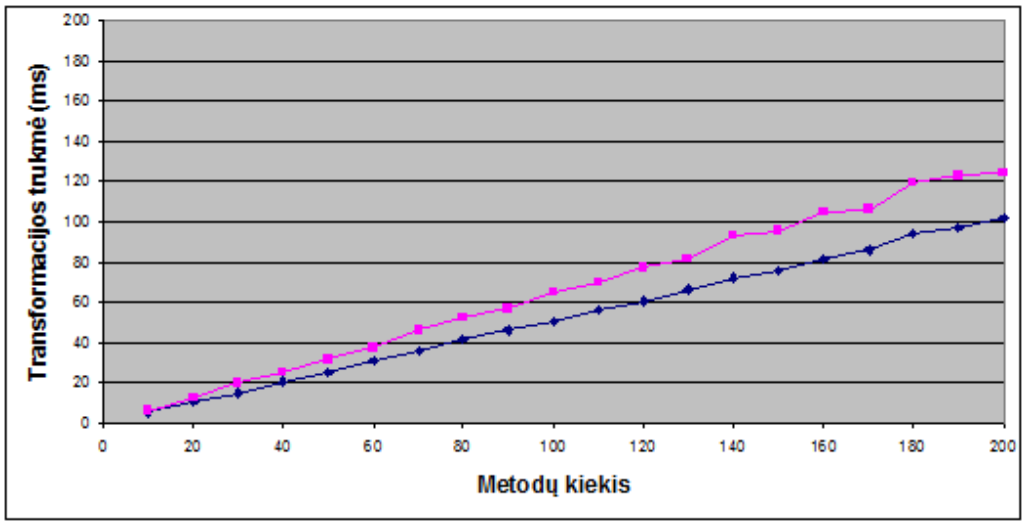
37 pav. AST transformavimo algoritmų greitaveikos grafikas (klasių kiekis – kintamas, kiekviena klasė turi po „void“ tipo metodą)



38 pav. AST transformavimo algoritmų greitaveikos grafikas (klasių kiekis – kintamas, kiekviena klasė turi po „int“ tipo metodą)



39 pav. AST transformavimo algoritmų greitaveikos grafikas (viena klasė su kintamu kiekiu „void“ tipo metodų)



40 pav. AST transformavimo algoritmų greitaveikos grafikas (viena klasė su kintamu kiekiu „int“ tipo metodų)

5.4. Eksperimento išvados

1) Atliktas eksperimentas su bendrinio AST medžio, sudaryto iš nedidelės apimties pavyzdinio Java išeities teksto fragmento, transformavimu į bendrinį AST ir atvirkščiai. Abiejų transformacijų rezultatai sutapo su tais, kurių tikėtasi. Tai rodo, jog biblioteka veikia teisingai.
2) Atlikus eksperimentą su konkrečiais AST medžiais, sukurtais iš skirtingo dydžio generuoto Java kalbos išeities tekstų, pavyko įžvelgti transformavimų greitaveikos kitimo dėsnius. Pastebėta, jog kintant bet kuriam vienam iš parametrų – klasių skaičiui, metodų skaičiui kiekvienoje klasėje, metodų rezultatų tipui, – vykdymo trukmė tiesiogiai proporcinga parametro dydžiui. Algoritmas yra O(n) eilės.
3) Greitaveikos eksperimento meto išaiškėjo, jog algoritmų vykdymo greitis pakankamai didelis ir auga toleruotinu tempu, todėl biblioteką galima pritaikyti praktinėms reikmėms.
4) Pastebėta, jog transformavimo procesas iš konkretaus AST į bendrinį visuomet veikia greičiau. Atvirkštinis procesas užima maždaug 20 procentų daugiau laiko. Peržvelgus ir palyginus abi transformacijas atliekantį kodą, prieita išvados, jog transformacija lėtesnė dėl intensyviau naudojamos refleksijos programavimo sąsajos (angl. „reflection API“). Pastaroji suteikia galimybę dinamiškai iškviesti bendrinių AST medžių elementų klasių metodus (tai reikalinga dinamiškai bandant gauti bendrinio AST elementų atributų reikšmes kai sulyginamas pradinis medis bei pateikta elementų seka), tačiau tai šiek tiek užtrunka, nes atliekama iškviečiamų metodų paieška pagal jų pavadinimą ir parametrų rinkinį.

6. IŠVADOS

1) Buvo išanalizuota panaši į kuriamą sistemą programinė įranga ir priemonės: Stratego/XT, TXL, XSTL, Alchemist ir atributinės gramatikos. Analizė parodė, kokios jų esminės savybės, kokios pagrindinės sudedamosios dalys, kuo kuriamoji sistema yra kitokia ir kodėl naudinga, nepaisant konkuruojančių sprendimų.
2) Analizė taip pat atskleidė ir unikalias sistemos problemas: tinkamų AST medžių formatų, medžių elementų sekų radimo, užklausų vykdymo mechanizmo bei transformacijų aprašų formato parinkimo klausimus. Egzistuojančių sprendimų panaudojimas šioms problemoms spręsti sutaupė daug pastangų ir laiko. Ypač daug sutaupyta pasirinkus Object Management Group specifikaciją bendrinio AST kūrimui.
3) Analizė, atlikta prieš sistemos prototipo kūrimą, pasirodė nepakankamai gera, nes buvo orientuota į bendresnę rinkoje egzistuojančių panašių įrankių analizę. Trūko gilesnės įrankių veikimo principų ir algoritmų analizės. To pasėkoje sukurtas prototipas veikė korektiškai tik su labai ribotos apimties AST medžiais.
4) Paskutinysis sistemos kūrimo etapas buvo itin naudingas, nes įvyko sistemos perėjimas nuo prototipo iki pakankamai gerai veikiančios sistemos.
5) Sistemos atnaujinimas sukėlė daugybę architektūrinių bei algoritmų realizacijos pasikeitimų, kurie galėjo būti pražūtingi. Tačiau kadangi sistema nėra itin didelė, pakeitimus pavyko realizuoti per paskutiniam vystymo etapui paskirtą laiką.
6) Atlikus eksperimentą pavyko įrodyti, jog pasiekti du pagrindiniai tikslai – įrodyta, kad biblioteka veikia su teisingai suformuotais duomenimis ir kad transformacijos vykdomos pakankamai greitai. Transformacijų algoritmų greitis yra O(n) eilės.
7) Pastebėta, jog bendrinių AST medžių transformacijų į konkrečius AST trukmė yra maždaug penktadaliu ilgesnė, nei transformacijų trukmė atvirkštine kryptimi. Išanalizavus išeities tekstus padaryta išvada, jog mažesnį greitį lemia intensyvesnis refleksijos programavimo sąsajos (angl. „reflection API“) naudojimas ieškant bendrinių AST elementų sekų.

7. LITERATŪRA

[1] **Abstract Syntax Tree.** Žiūrėta: 2009 m. sausio 3 d., prieiga per internetą: http://en.wikipedia.org/wiki/Abstract_syntax_tree

[2] M. Fowler, K. Beck, J.Brants, W. Opdyke, D. Roberts, E. Gamma. **Refactoring: improving the design of existing code**. Addison-Wesley Object Technology Series.

[3] Sangeeta Sabharwal. **Software Engineering : Tools, Principles and Techniques.** Umesh Publications.

[4] K. Czarnecki, S. Helsen. **Feature-based survey of model transformation approaches.** IBM Systems Journal, vol. 45, no. 3, 2006.

[5] M. Feilkas. **How to represent Models, Languages and Transformations?** Technische Universität München.

[6] A. van Deursen, P. Klint. **Domain-Specific Language Design Requires Feature Descriptions.** Centrum voor Wiskunde en Informatica (CWI), Amsterdam, The Netherlands.

[7] Eiffel Software. **Eiffel programming language.** Žiūrėta 2009 m. vasario 1 d., prieiga per internetą: http://www.eiffel.com/

[8] Eelco Visser. **Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems.** Institute of Information and Computing Sciences, Utrecht University.

[9] James R. Cordy, Thomas R. Dean, Andrew J. Malton, Kevin A. Schneider. **Source Transformation in Software Engineering Using the TXL Transformation System.** Queen's University, Kingston, Canada.

[10] Jonne van Wijngaarden, Eelco Visser. **Program Transformation Mechanics: A Classification of Mechanisms for Program Transformation with a Survey of Existing Transformation Systems.** Institute of Information and Computing Sciences, Utrecht University.

[11] **XSLT.** Žiūrėta: 2009 m. gegužės 19 d., prieiga per internetą: http://en.wikipedia.org/wiki/XSLT

[12] Greger Linden, Henry Tirri, A. Inkeri Verkamo. **Alchemist: A General Purpose Transformation Generator.** Department Of Computer Science, University Of Helsinki, Finland.

[13] **Attribute Grammar.** Žiūrėta: 2009 m. gegužės 22 d., prieiga per internetą: http://en.wikipedia.org/wiki/Attribute_grammar

[14] M. van den Brand, P.-E. Moreau, J. Vinju. **Generator of efficient strongly typed abstract syntax trees in Java.** IEE Proc.-Softw, April 2005, vol. 152, no. 2.

[15] T. J. Parr, R. W. Quong. **ANTLR: A Predicated-LL(k) Parser Generator.** Software – Practice and Experience, vol. 25(7), July 1995.

[16] **Object Management Group oficiali svetainė.** Žiūrėta 2009 m. gegužės 9 d., prieiga per internetą: http://www.omg.org/

[17] **Abstrakčios Sintaksės Medžio Metamodelio specifikacija.** Žiūrėta 2009 m. gegužės 9 d., prieiga per internetą: http://www.omg.org/docs/ptc/08-11-05.pdf

[18] Joost Visser. **Matching Objects Without Language Extension.** Departamento de Informatica, Universidade do Minho, Portugal.

[19] S. Khimta, P. S. Sandhu, A. S. Brar. **A Complexity Measure for JavaBean based Software Components.** Proceedings of World Academy of Science, Engineering and Technology, vol. 32, 2008 m. rugpjūtis.

[20] **Cascading Style Sheets.** Žiūrėta: 2009 m. kovo 3 d., prieiga per internetą: http://en.wikipedia.org/wiki/CSS

[21] **MapForce – Graphical Data Mapping, Conversion, and Integration Tool.** Žiūrėta 2010 m. gegužės 27 d., prieiga per internetą: http://www.altova.com/products/mapforce/data_mapping.html

8. TERMINŲ IR SANTRUMPŲ ŽODYNAS

- **API (Application Programming Interface) –** aplikacijų programavimo sąsaja
- **AST (Abstract Syntax Tree)** – abstrakčios sintaksės medis
- **Bendrinis AST (Generic Abstract Syntax Tree)** – abstrakčios sintaksės medis, nesusietas su konkrečia programavimo kalba ir talpinantis įvairių programavimo kalbų struktūrą
- **BNF (Backus-Naur Form) –** Backus-Naur forma (nuo konteksto nepriklausančių gramatikų aprašo forma)
- **CASE (Computer-Aided Software Engineering) –** automatizuotas kompiuterinis programinės įrangos projektavimas
- **CSS (Cascading Style Sheets)** – pakopinių stilių lapai (interneto puslapių stilių aprašo kalba)
- **EBNF (Extended Backus-Naur Form)** – išplėstinė Backus-Naur forma
- **Konkretus AST (Specific Abstract Syntax Tree)** – abstrakčios sintaksės medis, talpinantis konkrečios programavimo kalbos (C#, C++, Java ar pan.) struktūrą
- **Parser** – gramatinės analizės generatorius
- **Refactoring –** pertvarkymas (tai toks programinės įrangos pakeitimo procesas, kuris nepakeičia išorinio programos kodo elgesio, tačiau pagerina jo vidinę struktūrą)
- **Reflection –** refleksija (tai procesas, kurio metu programa gali stebėti ir modifikuoti savo pačios struktūrą ir elgesį)

9. PRIEDAI

9.1. Priedas nr. 1. Klasės „SpecificAstPattern“ išeities tekstas

```
package asttransformationlibrary.astpatterns;

import ast.specific.*;
import java.util.*;
import org.antlr.runtime.tree.CommonTree;
import util.DataStructureUtility;
import util.reflection.ReflectionUtility;

public class SpecificAstPattern extends SourcePattern<CommonTree> {
    private final HashMap<String, Integer> nodeTypeNamesWithValues;
    private final HashMap<Integer, String> nodeTypeValuesWithNames;

    public SpecificAstPattern(SourceNode rootNode) {
        super(rootNode);

        this.nodeTypeNamesWithValues =
ReflectionUtility.getFinalIntFieldNamesWithValues(JavaParser.class);
        this.nodeTypeValuesWithNames =
DataStructureUtility.switchKeysWithValues(nodeTypeNamesWithValues);
    }

    @Override
    public PatternMatchResult match(CommonTree rootAstNode) {
        HashMap<String, Object> foundVariables = new HashMap<String, Object>();
        boolean success = match(this.getRootNode(), rootAstNode, foundVariables);

        return new PatternMatchResult(success, foundVariables);
    }

    private boolean match(SourceNode patternNode, CommonTree astNode, HashMap<String, Object>
foundVariables) {
        if (matchVariable(patternNode, astNode, foundVariables)) {
            LinkedList<SourceNode> childPatternNodes = patternNode.getChildren();
            List<CommonTree> childAstNodes = astNode.getChildren();

            if (childPatternNodes.size() > 0) {
                if (childAstNodes != null) {
                    Iterator childPatternNodeIterator = childPatternNodes.iterator();
                    Iterator childAstNodeIterator = childAstNodes.iterator();

                    boolean success = true;
                    while (success && childPatternNodeIterator.hasNext()) {
                        if (childAstNodeIterator.hasNext()) {
                            SourceNode childPatternNode = (SourceNode)
childPatternNodeIterator.next();
                            CommonTree childAstNode = (CommonTree) childAstNodeIterator.next();

                            if (childPatternNode instanceof SourceNodeList) {
                                // Cycle through all nodes that match list pattern.
                                while (matchListVariable(childPatternNode, childAstNode,
foundVariables) && childAstNodeIterator.hasNext()) {
                                    childAstNode = (CommonTree) childAstNodeIterator.next();
                                }
                            } else {
                                success = match(childPatternNode, childAstNode, foundVariables);
                            }
                        } else {
                            return false;
                        }
                    }
                    return success;
                }
                return false;
            }
            return true;
        }
        return false;
    }
```

```
    private boolean matchListVariable(SourceNode patternNode, CommonTree astNode, HashMap<String,
Object> foundVariables) {
        String astNodeTypeName = nodeTypeValuesWithNames.get(astNode.getType());
        if (patternNode.getType().equals(astNodeTypeName) || patternNode.getType().equals("*")) {
            String key = patternNode.getVariable();
            LinkedList listVariable;

            if (foundVariables.containsKey(key)) {
                listVariable = (LinkedList) foundVariables.get(key);
                listVariable.add(astNode);
            } else {
                listVariable = new LinkedList();
                listVariable.add(astNode);
                foundVariables.put(key, listVariable);
            }

            return true;
        }
        return false;
    }

    private boolean matchVariable(SourceNode patternNode, CommonTree astNode, HashMap<String,
Object> foundVariables) {
        String astNodeTypeName = nodeTypeValuesWithNames.get(astNode.getType());
        if (patternNode.getType().equals(astNodeTypeName)) {
            if (patternNode.isVariable()) {
                foundVariables.put(patternNode.getVariable(), astNode);
            }
            return true;
        }
        return false;
    }
}
```

9.2. Priedas nr. 2. Klasės „SpecificToGenericAstTransformer“ išeities tekstas

```
package asttransformationlibrary.asttransformations;

import ast.generic.*;
import ast.specific.SpecificAst;
import asttransformationlibrary.astpatterns.*;
import java.util.*;
import org.antlr.runtime.tree.CommonTree;
import util.reflection.ReflectionUtility;
import util.text.TemplateProcessor;

public class SpecificToGenericAstTransformer extends AstTransformer<SpecificAst, GenericAst> {
    private final HashMap<String, Class> targetAstClassHierarchy;

    public SpecificToGenericAstTransformer(Transformation transformation) {
        super(transformation);

        this.targetAstClassHierarchy = GenericAst.getAstHierarchyClassessAsHashMap();
    }

    protected GenericAst transformSourceAstToTargetAst(SpecificAst sourceAst) throws Exception {
        Rule rule = this.getTransformation().getRootRule();
        GastmObject       targetAstRoot       =       this.transformSouceAstNodeToTargetAstNode(rule,
sourceAst.getRoot());
        return this.createTargetAst(targetAstRoot);
    }

    private GenericAst createTargetAst(GastmObject targetAstRoot) {
        GenericAst targetAst = new GenericAst();
        targetAst.setRoot(targetAstRoot);
        return targetAst;
    }

    protected GastmObject transformSouceAstNodeToTargetAstNode(Rule rule, CommonTree sourceAstNode)
throws Exception {
        SpecificAstPattern sourcePattern = new SpecificAstPattern(rule.getSourceRootNode());
        PatternMatchResult matchResult = sourcePattern.match(sourceAstNode);
```

```
        if (matchResult.isSuccess()) {
            return                                  this.createTargetAstNode(rule.getTargetRootNode(),
matchResult.getFoundVariables());
        } else {
            return null;
        }
    }

    private GastmObject createTargetAstNode(TargetNode targetPatternNode, HashMap<String, Object>
sourceVariables) throws Exception {
        GastmObject targetAstNode = this.createEmptyTargetAstNode(targetPatternNode);

        createChildGenericAstNodes(targetAstNode, targetPatternNode, sourceVariables);
        assignPropertyValues(targetAstNode, targetPatternNode,  sourceVariables);

        return targetAstNode;
    }

    private GastmObject createEmptyTargetAstNode(TargetNode targetPatternNode) {
        Class targetNodeClass = this.targetAstClassHierarchy.get(targetPatternNode.getType());
        return (GastmObject) ReflectionUtility.createInstanceOf(targetNodeClass);
    }

    private void createChildGenericAstNodes(GastmObject targetAstNode, TargetNode targetPatternNode,
HashMap<String, Object> sourceVariables) throws Exception {
        for (TargetNode childTargetPatternNode : targetPatternNode.getChildren()) {
            if (childTargetPatternNode instanceof TargetNodeList) {
                TargetNodeList childTargetPatternNodeList = (TargetNodeList) childTargetPatternNode;
                String targetNodeIdentifier = childTargetPatternNodeList.getNodeIndentifier();

                // Retrieve target node list that needs to be populated with values.
                LinkedList<GastmObject>      targetAstNodeList      =      (LinkedList<GastmObject>)
ReflectionUtility.invokeGetter(targetAstNode, targetNodeIdentifier);

                if  (childTargetPatternNodeList.isTransformationResult())  {  //  Add  target  list
elements using the result of some other transformation.
                    // Get transformation and variable.
                    LinkedList<String>                  simpleTransformationNames                  =
childTargetPatternNodeList.getRules();

                    String sourceVariableName = childTargetPatternNodeList.getSourceVariable();
                    LinkedList<CommonTree>      sourceAstNodes      =      (LinkedList<CommonTree>)
sourceVariables.get(sourceVariableName);

                    for (CommonTree sourceAstNode : sourceAstNodes) {
                        for (String simpleTransformationName : simpleTransformationNames) {
                            Rule                        simpleTransformation                        =
this.getTransformation().getRuleByName(simpleTransformationName);
                            GastmObject                     childTargetAstNode                     =
this.transformSouceAstNodeToTargetAstNode(simpleTransformation, sourceAstNode);

                            if (childTargetAstNode != null) {
                                targetAstNodeList.add(childTargetAstNode);
                                break; // Transformation applied successfully.
                            }
                        }
                    }
                } else { // Add target list elements as they are provided.
                    for             (TargetNode             listElementTargetPatternNode             :
childTargetPatternNode.getChildren()) {
                        GastmObject                     childTargetAstNode                     =
createTargetAstNode(listElementTargetPatternNode, sourceVariables);
                        targetAstNodeList.add(childTargetAstNode);
                    }
                }
            } else {
                String targetNodeIdentifier = childTargetPatternNode.getNodeIndentifier();
                GastmObject    childTargetAstNode    =    createTargetAstNode(childTargetPatternNode,
sourceVariables);
                ReflectionUtility.invokeSetter(targetAstNode,                   targetNodeIdentifier,
childTargetAstNode);
            }
        }
```

```
    }

    private    void    assignPropertyValues(Object    targetNode,    TargetNode    targetPatternNode,
HashMap<String, Object> sourceVariables) throws Exception {
        TemplateProcessor templateProcessor = new TemplateProcessor();

        for (PropertyAssignment propertyAssigment : targetPatternNode.getPropertyAssignments()) {
            // Get value expression and evaluate it.
            String valueExpression = propertyAssigment.getValueExpression();
            String value = templateProcessor.process(valueExpression, sourceVariables);

            // Assign value to appropriate property.
            String propertyName = propertyAssigment.getPropertyName();
            ReflectionUtility.invokeSetter(targetNode, propertyName, value);
        }
    }
}
```

9.3. Priedas nr. 3. Transformacijos aprašo XML failas (iš konkretaus AST į bendrinį)

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Transformation root="SourceRootRule">
       <Rule name="SourceRootRule">
              <Source>
                     <Node type="JAVA_SOURCE">
                            <Node type="ANNOTATION_LIST"/>
                            <NodeList type="CLASS" var="var_class_definition_list"/>
                     </Node>
              </Source>
              <Target>
                     <Node type="CompilationUnit">
                            <NodeList    name="Fragments"    sourceVar="var_class_definition_list"
useRule="ClassWithMethodsRule|ClassWithoutMethodsRule" type="DefinitionObject"/>
                            <Property name="Language" value="Java"/>
                     </Node>
              </Target>
       </Rule>
       <Rule name="ClassWithMethodsRule">
        <Source>
            <Node type="CLASS">
                <Node type="MODIFIER_LIST"/>
                <Node type="IDENT" var="var_class_name"/>
                <Node type="CLASS_TOP_LEVEL_SCOPE">
                                    <NodeList type="*" var="var_procedure_or_function_definition"/>
                            </Node>
            </Node>
        </Source>
        <Target>
            <Node type="AggregateTypeDefinition">
                <Node type="Name">
                    <Property name="NameString" value="${var_class_name.Text}"/>
                </Node>
                <Node name="AggregateType" type="ClassType">
                                    <NodeList         type="DefinitionObject"         name="Members"
useRule="ProcedureRule|IntFunctionRule" sourceVar="var_procedure_or_function_definition"/>
                            </Node>
            </Node>
        </Target>
    </Rule>
       <Rule name="ClassWithoutMethodsRule">
        <Source>
            <Node type="CLASS">
                <Node type="MODIFIER_LIST"/>
                <Node type="IDENT" var="var_class_name"/>
                <Node type="CLASS_TOP_LEVEL_SCOPE"/>
            </Node>
        </Source>
        <Target>
            <Node type="AggregateTypeDefinition">
                <Node type="Name">
                    <Property name="NameString" value="${var_class_name.Text}"/>
                </Node>
                <Node name="AggregateType" type="ClassType"/>
```

```
            </Node>
        </Target>
    </Rule>
        <Rule name="ProcedureRule">
                <Source>
                        <Node type="VOID_METHOD_DECL">
                                <Node type="MODIFIER_LIST"/>
                 <Node type="IDENT" var="var_procedure_name"/>
                 <Node type="FORMAL_PARAM_LIST"/>
                 <Node type="BLOCK_SCOPE"/>
                        </Node>
                </Source>
                <Target>
                        <Node type="FunctionDefinition">
                                <Node type="Public" name="AccessKind"/>
                                <Node type="Name" name="IdentifierName">
                                        <Property                                                name="NameString"
value="${var_procedure_name.Text}"/>
                                </Node>
                                <NodeList name="ReturnTypes">
                                        <Node type="UnnamedTypeReference">
                                                <Node type="GastmVoid" name="Type"/>
                                        </Node>
                                </NodeList>
                        </Node>
                </Target>
        </Rule>
        <Rule name="IntFunctionRule">
                <Source>
                        <Node type="FUNCTION_METHOD_DECL">
                                <Node type="MODIFIER_LIST"/>
                                <Node type="TYPE">
                            <Node type="INT"/>
                                </Node>
                 <Node type="IDENT" var="var_function_name"/>
                 <Node type="FORMAL_PARAM_LIST"/>
                 <Node type="BLOCK_SCOPE"/>
                        </Node>
                </Source>
                <Target>
                        <Node type="FunctionDefinition">
                                <Node type="Public" name="AccessKind"/>
                                <Node type="Name" name="IdentifierName">
                                        <Property name="NameString" value="${var_function_name.Text}"/>
                                </Node>
                                <NodeList name="ReturnTypes">
                                        <Node type="UnnamedTypeReference">
                                                <Node type="GastmInteger" name="Type"/>
                                        </Node>
                                </NodeList>
                        </Node>
                </Target>
        </Rule>
</Transformation>
```

9.4. Priedas nr. 4. Konkretaus AST turinys tekstiniu formatu

```
JAVA_SOURCE : JAVA_SOURCE
    ANNOTATION_LIST : ANNOTATION_LIST
    CLASS : class
        MODIFIER_LIST : MODIFIER_LIST
        IDENT : MyFirstClass
        CLASS_TOP_LEVEL_SCOPE : CLASS_TOP_LEVEL_SCOPE
            VOID_METHOD_DECL : VOID_METHOD_DECL
                MODIFIER_LIST : MODIFIER_LIST
                IDENT : methodWithVoidReturnType
                FORMAL_PARAM_LIST : FORMAL_PARAM_LIST
                BLOCK_SCOPE : BLOCK_SCOPE
            FUNCTION_METHOD_DECL : FUNCTION_METHOD_DECL
                MODIFIER_LIST : MODIFIER_LIST
                TYPE : TYPE
                    INT : int
                IDENT : methodWithIntReturnType
```

```
                FORMAL_PARAM_LIST : FORMAL_PARAM_LIST
                BLOCK_SCOPE : BLOCK_SCOPE
    CLASS : class
        MODIFIER_LIST : MODIFIER_LIST
        IDENT : MySecondClass
        CLASS_TOP_LEVEL_SCOPE : CLASS_TOP_LEVEL_SCOPE
```

9.5. Priedas nr. 5. Bendrinio AST turinys tekstiniu formatu

```
CompilationUnit
    PreprocessorElement(*)@{PreProcessorElements}
    AnnotationExpression(*)@{Annotations}
    ProgramScope@{OpensScope}
    DefinitionObject(*)@{Fragments}
        AggregateTypeDefinition
            PreprocessorElement(*)@{PreProcessorElements}
            AnnotationExpression(*)@{Annotations}
            Name@{Name}
                PreprocessorElement(*)@{PreProcessorElements}
                AnnotationExpression(*)@{Annotations}
                NameString : "MyFirstClass"
            ClassType@{AggregateType}
                PreprocessorElement(*)@{PreProcessorElements}
                AnnotationExpression(*)@{Annotations}
                IsConst : false
                IsVolatile : false
                DefinitionObject(*)@{Members}
                    FunctionDefinition
                        PreprocessorElement(*)@{PreProcessorElements}
                        AnnotationExpression(*)@{Annotations}
                        IsRegister : false
                        StorageSpecification@{StorageSpecifier}
                        Public@{AccessKind}
                            PreprocessorElement(*)@{PreProcessorElements}
                            AnnotationExpression(*)@{Annotations}
                        LinkageSpecifier : null
                        Name@{IdentifierName}
                            PreprocessorElement(*)@{PreProcessorElements}
                            AnnotationExpression(*)@{Annotations}
                            NameString : "methodWithVoidReturnType"
                        TypeReference@{DefinitionType}
                        FunctionScope@{OpensScope}
                        Statement(*)@{Body}
                        FormalParameterDefinition(*)@{FormalParameters}
                        FunctionMemberAttributes@{FunctionMemberAttributes}
                        TypeReference(*)@{ReturnTypes}
                            UnnamedTypeReference
                                PreprocessorElement(*)@{PreProcessorElements}
                                AnnotationExpression(*)@{Annotations}
                                IsConst : false
                                IsVolatile : false
                                GastmVoid@{Type}
                                    PreprocessorElement(*)@{PreProcessorElements}
                                    AnnotationExpression(*)@{Annotations}
                                    IsConst : false
                                    IsVolatile : false
                                    IsSigned : false
                    FunctionDefinition
                        PreprocessorElement(*)@{PreProcessorElements}
                        AnnotationExpression(*)@{Annotations}
                        IsRegister : false
                        StorageSpecification@{StorageSpecifier}
                        Public@{AccessKind}
                            PreprocessorElement(*)@{PreProcessorElements}
                            AnnotationExpression(*)@{Annotations}
                        LinkageSpecifier : null
                        Name@{IdentifierName}
                            PreprocessorElement(*)@{PreProcessorElements}
                            AnnotationExpression(*)@{Annotations}
                            NameString : "methodWithIntReturnType"
                        TypeReference@{DefinitionType}
                        FunctionScope@{OpensScope}
                        Statement(*)@{Body}
```

```
                        FormalParameterDefinition(*)@{FormalParameters}
                        FunctionMemberAttributes@{FunctionMemberAttributes}
                        TypeReference(*)@{ReturnTypes}
                            UnnamedTypeReference
                                PreprocessorElement(*)@{PreProcessorElements}
                                AnnotationExpression(*)@{Annotations}
                                IsConst : false
                                IsVolatile : false
                                GastmInteger@{Type}
                                    PreprocessorElement(*)@{PreProcessorElements}
                                    AnnotationExpression(*)@{Annotations}
                                    IsConst : false
                                    IsVolatile : false
                                    IsSigned : false
                AggregateScope@{OpensScope}
                DerivesFrom@{DerivesFrom}
        AggregateTypeDefinition
            PreprocessorElement(*)@{PreProcessorElements}
            AnnotationExpression(*)@{Annotations}
            Name@{Name}
                PreprocessorElement(*)@{PreProcessorElements}
                AnnotationExpression(*)@{Annotations}
                NameString : "MySecondClass"
            ClassType@{AggregateType}
                PreprocessorElement(*)@{PreProcessorElements}
                AnnotationExpression(*)@{Annotations}
                IsConst : false
                IsVolatile : false
                DefinitionObject(*)@{Members}
                AggregateScope@{OpensScope}
                DerivesFrom@{DerivesFrom}
    Language : "Java"
```

9.6. Priedas nr. 6. Transformacijos aprašo XML failas (iš bendrinio AST į konkretų)

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Transformation root="SourceRootRule">
       <Rule name="SourceRootRule">
              <Source>
                     <Node type="CompilationUnit">
                            <NodeList       name="Fragments"       var="var_class_definition_list"
type="DefinitionObject"/>
                     </Node>
              </Source>
              <Target>
                     <Node type="JAVA_SOURCE">
                            <Node type="ANNOTATION_LIST"/>
                            <NodeList      type="CLASS"      sourceVar="var_class_definition_list"
useRule="ClassWithMethodsRule|ClassWithoutMethodsRule"/>
                     </Node>
              </Target>
       </Rule>
       <Rule name="ClassWithMethodsRule">
        <Source>
            <Node type="AggregateTypeDefinition">
                <Node type="Name" var="var_class_name"/>
                <Node name="AggregateType" type="ClassType" >
                                    <NodeList         type="DefinitionObject"         name="Members"
var="var_procedure_or_function_definition"/>
                                </Node>
            </Node>
        </Source>
        <Target>
                     <Node type="CLASS">
              <Node type="MODIFIER_LIST"/>
              <Node type="IDENT" value="${var_class_name.NameString}"//>
              <Node type="CLASS_TOP_LEVEL_SCOPE">
                                    <NodeList    type="*"    useRule="ProcedureRule|IntFunctionRule"
sourceVar="var_procedure_or_function_definition"/>
                                </Node>
            </Node>
        </Target>
       </Rule>
```

```
        <Rule name="ClassWithoutMethodsRule">
         <Source>
             <Node type="AggregateTypeDefinition">
                 <Node type="Name" var="var_class_name"/>
                 <Node name="AggregateType" type="ClassType"/>
             </Node>
         </Source>
         <Target>
                         <Node type="CLASS">
                 <Node type="MODIFIER_LIST"/>
                 <Node type="IDENT" value="${var_class_name.NameString}"//>
                 <Node type="CLASS_TOP_LEVEL_SCOPE"/>
             </Node>
         </Target>
        </Rule>
        <Rule name="ProcedureRule">
                <Source>
                        <Node type="FunctionDefinition">
                                <Node type="Name" name="IdentifierName" var="var_procedure_name"/>
                                <NodeList name="ReturnTypes" type="TypeReference">
                                        <Node type="UnnamedTypeReference">
                                                <Node type="GastmVoid" name="Type"/>
                                        </Node>
                                </NodeList>
                        </Node>
                </Source>
                <Target>
                        <Node type="VOID_METHOD_DECL">
                                <Node type="MODIFIER_LIST"/>
                 <Node type="IDENT">
                                        <Property                                                name="Text"
value="${var_procedure_name.NameString}"/>
                                </Node>
                 <Node type="FORMAL_PARAM_LIST"/>
                 <Node type="BLOCK_SCOPE"/>
                        </Node>
                </Target>
        </Rule>
        <Rule name="IntFunctionRule">
                <Source>
                        <Node type="FunctionDefinition">
                                <Node type="Name" name="IdentifierName" var="var_function_name"/>
                                <NodeList name="ReturnTypes" type="TypeReference">
                                        <Node type="UnnamedTypeReference">
                                                <Node type="GastmInteger" name="Type"/>
                                        </Node>
                                </NodeList>
                        </Node>
                </Source>
                <Target>
                        <Node type="FUNCTION_METHOD_DECL">
                                <Node type="MODIFIER_LIST"/>
                                <Node type="TYPE">
                            <Node type="INT"/>
                                </Node>
                 <Node type="IDENT">
                                        <Property name="Text" value="${var_function_name.NameString}"/>
                                </Node>
                 <Node type="FORMAL_PARAM_LIST"/>
                 <Node type="BLOCK_SCOPE"/>
                        </Node>
                </Target>
        </Rule>
</Transformation>
```