UNIVERZA V MARIBORU

Fakulteta za elektrotehniko, računalništvo in informatiko

Damijan Rebernak

ASPEKTNO USMERJENE ATRIBUTNE GRAMATIKE

DOKTORSKA DISERTACIJA

Maribor, september 2008

Avtor: Damijan Rebernak, univ. dipl. inž. rač. in inf.

Naslov: Aspektno usmerjene atributne gramatike

UDK: 004.43:519.767(043.3)

Ključne besede: aspektno usmerjeno programiranje, atributne gramatike, domensko specifični aspektni jeziki, specifikacije jezika, generatorji prevajalnikov

Število izvodov: 8

Lektoriranje: Nina Ančič, prof. ang. in slov.

Obdelava besedila: Damijan Rebernak

Zahvala

Brez malih nedolžnih modrosti, ki sem jih bil v rani mladosti in vse do danes, deležen s strani mame, to delo nikoli ne bi prišlo v vaše roke. Mama, hvala za vse, kar si mi dala.

Za vso pomoč v času doktorskega študija, potrpljenje, vzpodbudne besede in za nepozabne izkušnje, ki me bodo gotovo spremljale celo življenje, se iskreno zahvaljujem svojemu mentorju izr. prof. dr. Marjanu Merniku.

Iskrena hvala tudi sodelavcem v Laboratoriju za programirne metodologije, ki so prenašali moje muhe, mi pomagali s številnimi nasveti in dolgimi debatami ob kavi.

Posebna zahvala velja dr. Mitji Leniču. Hvala za vse dolge pogovore in za potrpežljivo poslušanje mojih zmot.

Aspektno usmerjene atributne gramatike

UDK: 004.43:519.767(043.3)

Ključne besede:

aspektno usmerjeno programiranje, atributne gramatike, domensko specifični aspektni jeziki, specifikacije jezika, generatorji prevajalnikov

Povzetek

Razvoj programskih jezikov je, kljub uporabi naprednih orodij in metodologij, drag, odgovoren in zahteven postopek. Specifikacije so ponavadi zapletene, preobsežne, težko razumljive in le delno ponovno uporabne. Ravno to je razlog, da raziskovalci veliko pozornosti namenjamo izboljšanju modularnosti in ponovne uporabe specifikacij. Disertacija uvaja nov koncept razvoja programskih jezikov s čimer želimo izboljšati predvsem že opisane slabosti razvoja. S predlaganim pristopom k razvoju programskih jezikov, in predlaganim specifikacijskim jezikom, želimo razvijalcem ponuditi boljše orodje za načrtovanje in implementacijo programskih jezikov.

V prvem delu disertacije bralca popeljemo v svet programskih jezikov, od njihovih začetkov do uvoda v teorijo le–teh. V zadnjem času se precej uveljavlja aspektno usmerjeno programiranje, ki razvijalcem omogoča ločevanje dolžnosti v ponovno uporabne komponente. Opisu aspektno usmerjenega programiranja namenimo celotno poglavje (poglavje 3), saj je naše delo zasnovano na konceptih aspektno usmerjenega programiranja. V poglavju 4 opišemo formalne specifikacije programskih jezikov ter sorodna dela.

Osrednji del doktorske disertacije predstavlja poglavje 5, kjer zapišemo formalen model aspektno usmerjenih atributnih gramatik in večkratnega dedovanja le teh. V poglavju prikažemo motivacijo in teze za uvedbo aspektov, ki jih na koncu poglavja s primerom tudi potrdimo. Formalni model aspektno usmerjenih atributnih gramatik in večkratnega dedovanja le teh predstavlja izviren znanstveni prispevek disertacije.

V okviru disertacije smo razvili aspektno usmerjen domensko specifičen jezik za specificiranje programskih jezikov. Specifikacijski jezik orodja LISA smo nadgradili z aspektnimi lastnostmi. Za potrditev tez doktorske naloge

smo v orodju LISA implementirali prototipne jezike ter rezultate implementacije primerjali z že znanimi pristopi. Rezultati raziskave so opisani v poglavju 7. Rezultati so potrdili osnovno hipotezo, ki smo si jo zastavili na začetku disertacije. Dokazali smo, da je aspektno usmerjen pristop primeren za specificiranje programskih jezikov z atributnimi gramatikami. Predlagan pristop omogoča razvijalcem razvoj programskih jezikov na visokem abstraktnem nivoju ter zmanjša njihov trud pri samem razvoju in ponovni uporabi (razširjanju) programskega jezika.

Disertacijo zaključujemo s sklepnimi mislimi, s samokritičnim pogledom na naše delo ter možnostmi za nadaljne delo, ki jih disertacija nedvomno ponuja.

Aspect-Oriented Attribute Grammars

UDC: 004.43:519.767(043.3)

Keywords: Aspect-Oriented Programming, Attribute Grammars,

Domain-Specific Aspect Languages, Language specifications, Compiler Generators

Abstract

Despite various advanced tools and methodologies for the programming languages development, their programming is known as an expensive, responsible and a very complex task. Specifications of programming languages are usually very complex, extensive, hard to read and understand, and often only partially reusable. These are the main reasons why the research community puts tremendous efforts to improve modularity and reusability. This dissertation introduces the new concept of programming language development, which is going to improve already described drawbacks and simplify the development of programming languages.

The first part of the dissertation (section 2) introduces the reader the world of programming languages, their history and some theoretical background. Section 3 describes aspect-oriented programming (AOP), which is a programming technique for modularizing concerns that crosscut the basic functionality of programmes, and the core technology for improving programming language specifications used in this dissertation. Theoretical background of programming language specifications and related work are described in section 4.

The main part of the dissertation is presented in the section 5, which includes formal definitions of aspect-oriented attribute grammars (AOAGs) and multiple inheritance of AOAGs. The motivation and theses for introduction of AOAGs are presented in the beginning of the section, and are confirmed at the end of it. AOAGs and multiple inheritance of AOAGs are innovative scientific contributions of the dissertation.

The practical result of the dissertation is a new aspect-oriented domain specific specification language for specifications of programming languages. As the base language, the LISA specification language was used. The base

language was extended with aspect oriented features, which we presented in the previous section. In order to confirm theses stated at the beginning of the dissertation, some prototypical programming languages were developed in the LISA tool. The aspect-oriented implementations were compared with already known implementation approaches, using programming metrics for grammars. The results, presented in section 7 confirmed our thesises.

The concluding remarks with some self-critical thoughts about our work and thoughts about possible future work are gathered in the section 8.

KAZALO

Povzetek vii

Slike xi

Tabele xiii

Primeri xv

Algoritmi xvii

Kratice xix

1 Uvod 1

1.1 Motivacija in cilji . . . 3

1.1.1 Predpostavke in omejitve . . . 3

1.2 Originalni prispevki . . . 4

1.3 Struktura disertacije . . . 4

2 O programskih jezikih 7

2.1 Uvod . . . 7

2.2 Zgodovina . . . 10

2.3 Splošno namenski programski jeziki . . . 11

2.3.1 Objektno usmerjeni programski jeziki . . . 12

2.4 Domensko specifični programski jeziki . . . 13

2.4.1 Kdaj razviti domensko specifični jezik? . . . 14

3 Aspektno usmerjeno programiranje **19**
3.1 Uvod 20
3.2 Koncepti aspektno usmerjenega programiranja 21
3.3 Aspektno usmerjeni programski jeziki 25
3.3.1 Dimenzije načrtovanja aspektnih jezikov 28
3.3.2 Povzetek 30
3.4 AspectJ 30
3.4.1 Programski jezik 32
3.4.2 Prevajalnik in aspektno tkanje 35
3.4.3 Zaključek 35
3.5 Domensko specifični aspektno usmerjeni programski jeziki . . . 36
3.5.1 Model stičišč DSAL 38

4 Formalne specifikacije programskih jezikov **41**
4.1 Uvod 41
4.2 Leksikalne specifikacije 42
4.2.1 Regularni izrazi 43
4.3 Sintaksne specifikacije 46
4.3.1 Gramatike 47
4.4 Semantične specifikacije 51
4.4.1 Atributne gramatike 51
4.4.2 Denotacijska semantika 53
4.4.3 Algebrajska semantika 54
4.5 Specifikacijski jeziki 56
4.5.1 Orodja za avtomatski razvoj prevajalnikov 57
4.5.2 Ponovna uporabnost specifikacij programskih jezikov . 58
4.6 Sorodna dela doktorski disertaciji 70
4.6.1 Modularne atributne gramatike 70
4.6.2 Specifikacijski jezik APS 71
4.6.3 JastAdd 71
4.6.4 Montages 74

5 Aspektno usmerjene atributne gramatike **75**
5.1 Uvod in motivacija 75
5.2 Aspektno usmerjene atributne gramatike 76
5.2.1 Primer 80
5.3 Večkratno dedovanje aspektno usmerjenih atributnih gramatik 83
5.4 Prednosti AOAG 87

6 Orodje $LISA_{ver3.0}$ **89**
6.1 Uvod . . . 89
6.2 Specifikacijski jezik . . . 93
6.2.1 Komponentni del jezika . . . 94
6.2.2 Domensko specifični aspektni jezik . . . 96
6.2.3 Aspektno tkanje . . . 107
6.3 Vzporednice z ortogonalnimi koncepti aspektno usmerjenih jezikov . . . 110
6.4 Primer . . . 113

7 Rezultati in analiza **123**
7.1 Uvod . . . 123
7.2 Uporabljene metode ocenitve rezultatov . . . 124
7.3 Postavitev eksperimenta . . . 128
7.4 Meritve in analiza . . . 129

8 Zaključek **139**
8.1 Cilji . . . 140
8.2 Spoznanja in razprave . . . 142

Literatura **145**

Življenjepis **154**

Bibliografija **156**

SLIKE

3.1 Tkanje aspektov in komponent 25
3.2 Shema delovanja aspectJ prevajalnika 36

4.1 Delovanje pregledovalnika 42
4.2 Delovanje razpoznavalnika 47
4.3 Drevo izpeljav za aritmetični izraz `5 + 3 * 2` 49
4.4 Osnovna shema delovanja generatorjev prevajalnikov 57
4.5 Arhitektura orodja JastAdd 73

5.1 Drevo ovrednotenja osnovne gramatike za stavek *abad*. 83
5.2 Drevo ovrednotenja po tkanju za stavek *abad*. 84

6.1 Arhitektura orodja LISA 90
6.2 Grafični vmesnik orodja LISA 92

7.1 Hieararhija dedovanja jezikov za jezik `Calc`. 130
7.2 Hieararhija dedovanja jezikov za jezik `RobotCalculator`. . . . 132
7.3 Hieararhija dedovanja jezikov za jezik `PLM`. 134
7.4 Hieararhija dedovanja jezikov za jezik `Tiny`. 137

TABELE

2.1 Primeri domensko specifičnih jezikov 14

4.1 Lastnosti osnovnih leksikalnih simbolov 44
4.2 Orodja za razvoj prevajalnikov 59

7.1 Rezultati meritev metrik za primer `Calc`. 131
7.2 Rezultati meritev metrik za primer `RobotCalculator`. 133
7.3 Rezultati meritev metrik za jezik `PLM`. 135
7.4 Rezultati meritev metrik za jezik `Tiny`. 136
7.5 Sumarna predstavitev rezultatov 138

Primeri

4.1 Stavek v programskem jeziku java 43
4.2 Primeri regularnih izrazov 46
4.3 Primeri regularnih definicij 46
4.4 Gramatika za aritmetične izraze 48
4.5 Gramatika za aritmetične izraze zapisana v BNF notaciji . . . 50
5.1 Prikaz konceptov aspektno usmerjenih atributnih gramatik . . 81
6.1 Osnovna sintaksa komponente v orodju LISA 93
6.2 Specifikacije programskega jezika `Robot` 97
6.3 Specifikacije programskega jezika `RobotTime` 99
6.4 Specifikacije programskega jezika `RobotCalc` 106
6.5 Specifikacije programskega jezika `RobotTime`, brez aspektnih lastnosti LISA jezika 111
6.6 Specifikacije programskega jezika `PlusMinus` 114
6.7 Specifikacije programskega jezika `MulDiv` 115
6.8 Specifikacije programskega jezika `Core` (implementacija brez aspektnih lastnosti LISA jezika) 116
6.9 Specifikacije `abstraktnega` programskega jezika `Stmts` 117
6.10 Specifikacije programskega jezika `Core` 118
6.11 Specifikacije programskega jezika `CoreInOut` 119
6.12 Specifikacije `abstraktnega` programskega jezika `StmtCounter` 120
6.13 Specifikacije `abstraktnega` programskega jezika `Block` 121
6.14 Specifikacije programskega jezika `Tiny` 122
6.15 Program v programskem jeziku `Tiny` 122

Algoritmi

5.1 Algoritem tkanja AOAG 80
6.1 Glavni algoritem tkanja v orodju LISA 108
6.2 Algoritem za združevanje semantičnih funkcij 109

Uporabljene kratice

Kratica	**Razlaga**
AG	atributne gramatike (ang. *Attribute Grammars*)
ANTLR	Orodje za generiranje razpoznavalnikov in prevajalnikov (ang. *ANother Tool for Language Recognition*)
AOAG	aspektno usmerjene atributne gramatike (ang. *Aspect-Oriented Attribute Grammars*)
AOP	aspektno usmerjeno programiranje (ang. *Aspect-Oriented Programming*)
AOSD	aspektno usmerjen razvoj programske opreme (*Aspect-Oriented Software Development*)
API	aplikacijski vmesnik (angl. *Application Programming Interface*)
ASF+SDF	razvojno okolje za programske jezike (angl. *Algebraic Specification Formalism and Syntax Definition Formalism*)
ASM	abstraktni stroj stanj (ang. *Abstract State Machine*)
AST	abstraktno sintaksno drevo (ang. *Abstract Syntax Tree*)
BNF	meta opisni jezik za definicijo sintakse programskih jezikov (angl. *Backus-Naur Form*)
CFG	kontekstno neodvisna gramatika (angl. *Context-Free Grammar*)
DSAL	domensko specifični aspektni jezik (ang. *Domain-Specific Aspect Language*)
DSL	domensko specifični programski jezik (ang. *Domain-Specific Language*)
EBNF	razširjen meta opisni jezik za definicijo sintakse programskih jezikov (angl. *Extended Backus-Naur Form*)
eLOC	efektivne vrstice programskega koda (angl. *effective Lines Of Code*)
GPL	splošno namenski programski jezik (ang. *General-Purpose Language*)
HTML	označevalni jezik za spletne vsebine (ang. *Hypertext Markup Language*)
IDE	interaktivno razvojno orodje (ang. *Interactive Development Environment*)
JDK	razvojno okolje za razvoj v programskem jeziku java (*Java Development Kit*)
JPM	model stičišč aspektnega oz. aspektno usmerjenega jezika (*Join Point Model*)
JVM	java virtualni stroj (*Java Virtual Machine*)
LISA	orodje za generiranje razpoznavalnikov/prevajalnikov (ang. *Language Implementation System based on Attribute Grammars*)
MAG	modularne atributne gramatike (ang. *Modulat Attribute Grammars*)
MCS	Komponentni sistem za razvoj programskih jezikov *Montage Component System*
OOP	objektno usmerjeno programiranje (ang. *Object-Oriented Programming*)
ReRAGs	vrsta atributnih gramatik (ang. *Rewritable Circular Reference Attributed Grammars*)

Don't panic.

– The Hitchhiker's Guide to the Galaxy –

Poglavje 1

Uvod

Programski jeziki sodijo med osrednja področja računalništva. Uporabljajo se od samega začetka in se z evolucijo računalniške znanosti spreminjajo ter dopolnjujejo, nastajajo pa tudi novi programski jeziki. Področja uporabe se venomer širijo, tako da je potreba po učinkovitem razvoju programskih jezikov še kako prisotna. Načrtovanje programskih jezikov je zelo zahteven, odgovoren in drag postopek. Od načrtovanja programskega jezika je odvisna učinkovitost razvijalcev programske opreme. Primeren programski jezik lahko bistveno zmanjša stroške razvoja in vzdrževanja ter zmanjša možnost programskih napak. Trenutno obstoječi splošno namenski jeziki nudijo zadovoljivo izrazno moč ter visok nivo abstrakcije. Zakaj, ob poplavi splošno namenskih programskih jezikov, potrebujemo nove programske jezike? Odgovor je v raznolikosti področij in domen uporabe, stroških učenja razvijalcev ter ceni razvoja programskih rešitev. Kako načrtujemo povsem nov programski jezik? Na žalost še vedno ne obstaja enoten, formalen način za načrtovanje programskih jezikov. Ker želimo načrtovati jezik, ki bo enoumen in formalno definiran, se pri načrtovanju in implementaciji naslanjamo na formalne metode za opis sintakse (BNF - Backus Naur Form) in semantike (denotacijska semantika, operacijska semantika, atributne gramatike, aksiomatska semantika itd.) programskih jezikov. Iz formalnih specifikacij je mogoče prevajalnik tudi uspešno avtomatsko generirati, kar je eden izmed

naših ciljev.

V zadnjem času se veliko pozornosti raziskovalcev usmerja na domensko specifične jezike. Ti programski jeziki so "manjši" od splošno namenskih in so namenjeni za uporabo le v določeni domeni, za katero so načrtovani. Ti jeziki primarno niso namenjeni zgolj računalniškim strokovnjakom oz. programerjem, temveč predvsem domenskim strokovnjakom. So neprimerno lažji za učenje in uporabo kot splošno namenski programski jeziki ter nudijo razvijalcem zelo visok nivo abstrakcije.

Iz opisanega je razvidno, da je hiter razvoj učinkovitih programskih jezikov ključnega pomena. Pri tem imamo v mislih predvsem načrtovanje novih domensko specifičnih programskih jezikov in evolucijo že obstoječih. Procesi, npr. proizvodni proces, kjer lahko robote krmilimo z domensko specifičnim programskim jezikom, se spreminjajo in temu morajo uspešno slediti programski jeziki. Načrtovanje in implementacija domensko specifičnega programskega jezika vsekakor ni enostavna naloga in zahteva poznavanje domene, kakor tudi visok nivo poznavanja delovanja prevajalnikov ter veliko izkušenj na tem področju.

Trenutno obstaja veliko število raziskav, ki se ukvarjajo z različnimi pristopi implementacije domensko specifičnih programskih jezikov. V delu smo se pri načrtovanju programskih jezikov osredotočili predvsem na specifičen formalni model načrtovanja, model ponovno uporabnih modulov (komponent) programskega jezika ter avtomatsko generiranje interpreterja oz. prevajalnika za definiran programski jezik iz formalno podanih specifikacij. Za dosego zastavljenega cilja smo nadgradili že obstoječe delo [1], kjer je za ponovno uporabnost formalnih specifikacij programskega jezika uporabljen mehanizem večkratnega dedovanja atributnih gramatik. Obstoječe delo smo dopolnili s paradigmami poznanimi iz aspektno usmerjenega programiranja. Disertacija vključuje naslednje izboljšave:

- Ločevanje različnih lastnosti programskih jezikov v ponovno uporabne module.
- Poenostavitev integracije ponovno uporabnih modulov.
- Višji abstraktni nivo zapisa programskega jezika z atributnimi gramatikami.
- Poenostavljeno inkrementalno dodajanje semantičnih pravil. Ni potrebe po redefiniranju celotnega pravila oz. produkcije za dodajanje, spreminjanje in brisanje določenih semantičnih pravil.

1.1 Motivacija in cilji

Glavni cilj doktorske disertacije je raziskati možnosti uporabe aspektno usmerjenih paradigm pri specificiranju programskih jezikov z atributnimi gramatikami. V doktorski disertaciji potrjujemo naslednjo tezo.

Aspektno usmerjene atributne gramatike so primeren pristop za načrtovanje programskih jezikov, s katerim je mogoče doseči povečanje ponovne uporabnosti in zmanjšati obseg formalnih specifikacij, v primerjavi z osnovnimi atributnimi gramatikami.

Teza razširja trenutni model atributnih gramatik. Tezo potrdimo z razvojem aspektno usmerjenega specifikacijskega jezika, temelječega na aspektno usmerjenih atributnih gramatikah, ki izboljšajo ločevanje posameznih dolžnosti programskih jezikov. Specifikacijski jezik smo integrirali v orodje, ki omogoča avtomatsko generiranje prevajalnikov programskih jezikov. Pri razvoju jezika smo izkoristiti vse prednosti atributnih gramatik in jih z opisanimi koncepti nadgradili ter jim tako povečali praktično uporabnost. Na ta način izboljšamo strukturiranost specifikacij, kar v praksi rezultira k boljši modularnosti, ponovni uporabnosti in inkrementalnemu razvoju programskih jezikov ter posledično lažji implementaciji le–teh. Tako se olajša razvoj predvsem domensko specifičnih jezikov, ki se v praksi vedno bolj uveljavljajo.

1.1.1 Predpostavke in omejitve

V raziskavi se omejimo na atributne gramatike navkljub temu, da se zavedamo možnosti apliciranja predstavljenih originalnih idej na ostalih formalizmih za specifikacijo programskih jezikov. Natančneje se osredotočimo na nadgradnjo večkratnega dedovanja atributnih gramatik, ki že ponuja mehanizme za modularni in inkrementalni razvoj programskih jezikov. Za atributne gramatike smo se odločili zaradi dobrih rezultatov predhodnih raziskav in zaradi njihove razumljivosti in razširjenosti uporabe pri načrtovanju programskih jezikov. Zaradi specifičnosti formalnih specifikacij jezikov z atributnimi gramatikami na deklarativnem nivoju vseh ortogonalnih konceptov aspektno usmerjenih programskih jezikov ni mogoče vključiti v aspektne atributne gramatike. Primernost predlaganega pristopa preverimo na izbranih prototipnih programskih jezikih. Pri ocenjevanju velikosti, kompleksnosti in modularnosti specifikacij jezika uporabimo že uveljavljene programske metrike.

1.2 Originalni prispevki

Disertacija uvaja nove koncepte na področju načrtovanja, formalnega zapisa specifikacij programskih jezikov ter avtomatskega generiranja interpreterjev/prevajalnikov za programski jezik. Koncepti so združitev in nadgradnja že znanih konceptov na področju načrtovanja in implementacije programskih jezikov, ob tem pa predlagana disertacija dodaja še naslednje novosti:

- Definiranje formalnega modela aspektno usmerjenih atributnih gramatik.
- Definicija in implementacija algoritma za tkanje v aspektno usmerjenih atributnih gramatikah.
- Definicija in implementacija algoritma za večkratno dedovanje aspektno usmerjenih atributnih gramatik.
- Načrtovanje in implementacija novega specifikacijskega jezika v orodju LISA (*Language Implementation System based on Attribute Grammars*).
- Napotki za načrtovanje programskih jezikov z uporabo vzorcev za načrtovanje in avtomatsko generiranje interpreterja/prevajalnika iz tako podanih formalnih specifikacij programskega jezika.

Praktičen del naloge je implementiran v programskem jeziku java. Rezultat dela je orodje, ki iz podanih aspektno usmerjenih specifikacij atributnih gramatik avtomatsko generira prevajalnik in druga orodja za razvoj in uporabo novo razvitega programskega jezika.

1.3 Struktura disertacije

Uvodnemu poglavju disertacije sledi poglavje o programskih jezikih, ki bralca uvede v svet programskih jezikov ter ga seznani s teorijo programskih jezikov ter formalnimi specifikacijami. Naslednje poglavje (poglavje 3) je namenjeno opisu aspektno usmerjenega programiranja, katerega lastnosti smo poskušali uporabiti pri specifikaciji programskih jezikov ter aplikaciji le–tega na atributne gramatike. V četrtem (4) poglavju se lotimo pregleda formalnih metod za specifikacijo programskih jezikov ter pregleda sorodnih del. Osrednje poglavje doktorske disertacije je peto (5) poglavje, kjer je predstavljen teoretični model aspektno usmerjenih atributnih gramatik. Na podlagi teoretičnih dognanj smo nadgradili domensko specifični jezik za specifikacijo programskih

jezikov orodja za avtomatsko generiranje prevajalnikov $LISA_{ver.2.0}$. Specifikacijski jezik je opisan v šestem (6) poglavju, ki mu sledi poglavje z doseženimi rezultati. Disertacija je zaključena s pregledom spoznanj in dosedanjih dosežkov ter nekaj mislimi o prihodnjih raziskavah.

I personally believe we developed language because of our deep inner need to complain.

– Jane Wagner –

Poglavje 2

O programskih jezikih

Celotna disertacija govori o programskih jezikih, zato je prav, da se v začetku dela na kratko, in na dokaj neformalen način, seznanimo z njimi. Poglavje bralca uvede v svet programskih jezikov in ga seznani z njihovo zgodovino.

2.1 Uvod

Medsebojna človeška komunikacija poteka na več načinov. Glavni vir sporazumevanja med ljudmi je naravni ali materni jezik, ki se ga naučimo v rani mladosti in nam v večini primerov služi kot osnovni vir medsebojne komunikacije prav do smrti. Ne smemo pa zanemariti tudi drugih oblik komuniciranja, kot so gibi rok, mimika obraza itd. Kakor hitro se znajdemo izven meja naše domovine, se soočimo s problemom sporazumevanja, saj različni narodi govore različne jezike. V daljni preteklosti je s stališča komunikacije vladalo idealno stanje. Vsi na planetu naj bi govorili isti jezik. Na ta način ni bilo težav v komuniciranju, ki jih srečujemo danes. Edina ovira pri komuniciranju je bila izrazna moč jezika, ki so ga uporabljali. O tem nam govori zgodba o Babilonskem stolpu [2], ki so ga gradili z namenom doseči velikost in slavo Boga. Projekt je potem, ko jim je Bog zmešal jezike, neslavno propadel,

ljudje pa so se razselili širom po planetu. Zgodba nam priča o pomembnosti (enoumne) komunikacije. Tudi danes žal ne poznamo univerzalnega prevajalnika, kot npr. v znanstveno fantastičnih delih[1], ali univerzalnega jezika (čeprav že obstajajo poskusi [4]), tako da je problem komunikacije še vedno prisoten. Naslednji problem naravnih jezikov je njihova dvoumnost, saj lahko imajo določene besede, besedne zveze ali celo celi stavki v različnih kontekstih povsem različen pomen. Vsem nam je znan stavek *"Gori na gori gori"*, kjer ima ista beseda v enem stavku kar tri pomene. Tudi na področju programskih jezikov imamo isti problem. Računalnik "razume" le en jezik, mi pa se želimo z njim pogovarjati – *programirati* – v več programskih jezikih. Napram naravnim jezikom morajo biti programski jeziki enoumni, saj je le na ta način mogoča ustrezna interpretacija zapisanih postopkov. Na žalost tudi v tem primeru nimamo univerzalnega prevajalnika, ki bi znal prevajati vse jezike, zato je razvoj prevajalnikov za različne programske jezike zelo pomembna disciplina računalniške znanosti.

Programski jezik je umetni jezik, namenjen kontroli izvajanja računalnika, služi pa za interakcijo človek–računalnik. Teorija programskih jezikov je ena izmed najbolje raziskanih področij računalništva, saj sodi na njegovo osrednjo področje. Trenutno obstaja v svetu več kot 3000 programskih jezikov in vsaj 40 različnih vrst programskih jezikov [5, 6], in v prihodnosti lahko pričakujemo še večji razmah. Nemogoče je pričakovati, da bi programerji poznali vse programske jezike oz. vedno uporabljali samo en programski jezik. Verjetno pa nikoli ne bo obstajal en sam univerzalen programski jezik. Nekateri programski jeziki so primerni za hitre rešitve in prototipiranje, spet drugi za večje projekte. Tako je sposobnost ocenjevanja jezika zelo pomembna pri določanju uporabnosti programskega jezika in izbiri le tega pri razvoju novih programskih produktov. Poznavanje principov programskih jezikov je tudi pogoj za uspešno načrtovanje novih programskih jezikov. Preden se lotimo formalnega pristopa k študiju programskih jezikov, si oglejmo nekaj preprostih in neformalnih definicij programskih jezikov:

- Programski jezik je jezik za zapis računalniških programov, ki vsebujejo izračune ali algoritme z možnostjo nadzora zunanjih naprav, kot so tiskalniki, roboti itd.

[1] *The Babel fish is small, yellow and leechlike, and probably the oddest thing in the Universe. It feeds on brainwave energy received not from its own carrier but from those around it. It absorbs all unconscious mental frequencies from this brainwave energy to nourish itself with. It then excretes into the mind of its carrier a telepathic matrix formed by combining the conscious thought frequencies with nerve signals picked up from the speech centres of the brain which has supplied them. The practical upshot of all this is that if you stick a Babel fish in your ear you can instantly understand anything said to you in any form of language [3].*

- Programski jezik omogoča podajanje navodil za izvajanje računskih postopkov, ki so enostavna tako za človeka kot za računalnik.
- Programski jezik predstavlja komunikacijo z računalniki.
- Programski jezik je programerjevo osnovno orodje.

Od programskega jezika se zahteva, da je univerzalen, tj. omogoča zapis vsake rešljive naloge (univerzalen je vsak jezik, v katerem lahko definiramo rekurzivne funkcije). Po drugi strani pa želimo programski jezik oklestiti vseh nepotrebnih konstruktov in njegovo notacijo približati domeni, v kateri se uporablja. Večna je torej dilema ali naj za razvoj določenega programskega produkta (aplikacije) uporabimo splošno namenski programski jezik ali pa po drugi strani že obstaja *"manjši"* domensko specifični jezik, ki ga lahko uporabimo. Morebiti pa se lotimo celo razvoja novega jezika, ki bo notacijsko bližje danemu problemu. Da bi se ob poplavi vseh jezikov lažje odločili za pravega, je smiselno jezike razvrstiti v razrede. V literaturi zasledimo več načinov klasifikacije programskih jezikov. Pa si poglejmo nekaj razredov programskih jezikov[2]:

- Glede na namen uporabe delimo programske jezike na:
  - splošno namenske jezike (***G**eneral-**P**urpose **L**anguage*) in
  - domensko specifične jezike (***D**omain-**S**pecific **L**anguage*).
- Glede na način opisa problema oz. rešitve delimo programske jezike na:
  - imperativne (stavki jezika so zapisani v logičnem zaporedju, kot bi se naj izvajali) in
  - deklarativne (program podaja opis podatkov in relacij med njimi ne pa tudi načina izvajanja programa, tega določa že sama semantika programskega jezika).
- Glede na zapis programov delimo programske jezike na:
  - linearne ali tekstovne (algoritem programa je podan v tekstovni obliki) in
  - vizualne (algoritem oz. celoten program je podan z vizualnimi konstrukti).
- Glede na vzorec zapisa algoritma delimo programske jezike na:

[2]Programski jezik določenega razreda lahko spada tudi v druge razrede.

  - proceduralne (program je zapisan kot seznam instrukcij; program vsebuje podprograme imenovane funkcije oz. procedure),
  - objektne,
  - objektno usmerjene,
  - aspektno usmerjene,
  - funkcijske,
  - logične (uporaba matematične logike za zapis programa),
  - paralelne itd.

- Glede na nivo oziroma na število strojnih instrukcij, ki so potrebne, da se izvede posamezni stavek jezika, delimo programske jezike na:
  - nizke oz. zbirne,
  - visoke in
  - zelo visoke.

- Glede na način implementacije delimo programske jezike na:
  - interpretirane in
  - prevedene.

Na tem mestu se ne bomo spuščali v podrobnejšo primerjavo posameznih razredov programskih jezikov, ker to ne spada v področje disertacije. Direktna primerjava vseh teh programskih jezikov tudi ni mogoča in je v literaturi v taki obliki tudi ni zaslediti. Se pa notacije (sintakse) posameznih programskih jezikov med sabo precej razlikujejo. Dobro načrtovan programski jezik, predvsem sintaksa, lahko bistveno pripomore k uspehu določenega programskega jezika.

2.2 Zgodovina

Zgodovina programskih jezikov se začne v t.i. predzgodovinski dobi (gledano s stališča programskih jezikov), dobi pred računalniki. V tem obdobju so obstajali modeli in ideje za zapis programov. Za prvo programerko štejemo Ado Byron, ki je leta 1830 zapisala program za reševanje Bernoulijevih enačb za analitični stroj (Babaggeov stroj). Naslednji pomembni mejniki zgodnjega obdobja zgodovine programskih jezikov so naslednji:

- kodiran zapis algoritma na luknjanih karticah (Herman Hollerith, 1890),

- račun labda (*Lambda Calculus* – 1933),
- Turingov stroj (1936) in
- Plankalkul (Konrad Zuse; prvi algoritemski programski jezik, ki pa ni nikoli bil implementiran – 1945).

V samem začetku razvoja programskih jezikov smo bili priča mnogim novostim (zbirniki, prevajalniki, prvi višjenivojski programski jeziki itd.). Pozameznih jezikov, ki so pomenili mejnike v razvoju le teh, posebej ne bi naštevali. Bi pa vseeno izpostavili nekaj mejnikov iz zgodnjega obdobja razvoja programskih jezikov:

- 1944 – poročilo EDVAC, kjer je prvič opisan strojni jezik (Von Neumann),
- 1951 – prvič je opisana ideja prevajanja uporabniških instrukcij v strojne instrukcije in
- 1954 - 1957 – prvi visoki programski jezik fortran (skupina pod vodstvom Backusa).

Več o zgodovini programskih jezikov najde bralec v [7].

Tem jezikom je kasneje sledila prava poplava različnih jezikov. Vsak jezik je prispeval nekaj novih konceptov in idej ter pripomogel k uspešnosti sodobnih programskih jezikov. Prvi programski jeziki niso bili načrtovani, ker se je razvijalcem zdel ta korak razvoja zelo preprost. Pomanjkanje načrtovanja je rezultiralo k slabi sintaksi jezika in napakam programerjev, ki so razvijali v tem jeziku. Danes velja načrtovanje za eno izmed pomembnejših faz razvoja programskih jezikov in programskih produktov nasploh. Verjamemo, da bodo spoznanja opisana v tej disertaciji pripomogla k boljšemu načrtovanju in učinkovitejši implementaciji programskih jezikov.

2.3 Splošno namenski programski jeziki

Večina programskih jezikov o katerih smo govorili v prejšnjem poglavju spada med t.i. splošno namenske programske jezike (GPL). Med splošno namenske programske jezike štejemo vse jezike, katerih izrazna moč presega zgolj določeno domeno. S takšnimi programskimi jeziki lahko razvijamo programe za poljubno domeno. Natančna definicija o tem, kdaj lahko določen jezik označimo z značko splošno namenski programski jezik, ne obstaja. Za

boljše razumevanje naštejmo nekaj trenutno najpogosteje uporabljenih splošno namenskih programskih jezikov [8]: java, C, C++, visual basic, pascal itd. Za kriterij ali je jezik splošno namenski ali domensko specifični ni pomembno, ali je jezik preveden/interpretiran, linearen/vizualen, proceduralen/objektni/aspektni itd., šteje zgolj aplikativnost programskega jezika.

2.3.1 Objektno usmerjeni programski jeziki

Večina danes najpogosteje uporabljenih programskih jezikov sledi konceptom objektnega programiranja in/ali objektno usmerjenega programiranja (***O**bject-**O**riented **P**rogramming* – OOP). Tudi mi se pri izvirnih prispevkih disertacije deloma naslonimo na te koncepte. Zato je prav, da jih malce pobližje spoznamo.

Bistvena entiteta v objektnih jezikih je objekt. Objekt je avtonomna entiteta s stanjem, ki sprejema sporočila oz. izvaja operacije nad stanjem. Bolj natančna definicija se glasi:

Definicija 2.3.1 *Objekt predstavlja množico operacij, ki si delijo stanje. Operacije nad stanji določajo sporočila, na katera se objekt odzove. Operacije si delijo stanje, ki je skrito in dostopno samo preko operacij. Operacijam pravimo metode. Množica metod pridruženih objektu določa njegov vmesnik in obnašanje.*

Za prvi objektno usmerjen programski jezik štejemo jezik simula-67 [9], ki je kot prvi uvedel koncepte, kot so: objekt, razred in dedovanje. Te koncepte so kasneje nadgradili s programskim jezikom smalltalk [10]. Pravi razcvet pa je objektno usmerjeno programiranje doživelo s predstavitvijo jezika C++ [11], ki je nadgradnja jezika C.

Pomembni koncepti objektno usmerjenega programiranja so:

- razred (*class*): klasificira objekte glede na njihovo strukturo in skupno obnašanje ter služi kot šablona za ustvarjanje objektov,
- objekt (*object*): kapsulira stanje in obnašanje,
- kapsulacija (*encapsulation*): definira skrivanje stanja objekta, stanje je dostopno samo preko operacij,
- dedovanje (*inheritance*): klasificira objekte/razrede v hierarhije, osnovna tehnika za inkrementalno spreminjanje programov,
- pošiljanje sporočil: mehanizem za sprožitev operacij,

- močno tipiziranje (*strong typing*): statične omejitve glede na uporabo operacij,
- vključitveni polimorfizem (*inclusion polymorphism* [12]): objekt lahko pripada mnogim nadrazredom, objekti določenega tipa se lahko uporabijo tudi tam, kjer pričakujemo objekte nadrazreda.

Osnovna ideja objekto usmerjenega programiranja je zadano nalogo opisati z množico objektov, ki si med seboj pošiljajo sporočila in na ta način izvajajo operacije. Objekti kapsulirajo stanje in obnašanje. Strukturo (stanje) objekta opišemo s spremenljivkami. Tem spremenljivkam pravimo tudi objektne oz. instančne spremenljivke (*instance variables*). Obnašanje objekta pa je določeno z metodami (*methods*), ki operirajo nad instančnimi spremenljivkami in vhodnimi podatki. Objekt ponavadi ustvarimo s pomočjo razredov, ki služijo kot šablona. Z razredom torej opišemo strukturo in obnašanje vseh tistih objektov, ki pripadajo temu razredu (so ustvarjeni iz določenega razreda). Kot smo dejali že na začetku tega poglavja, je eden izmed najpomembnejših konceptov objektno usmerjenega programiranja dedovanje. Dedovanje omogoča inkrementalen razvoj (spreminjanje) programov. Iz nadrazredov (lahko bi jim rekli tudi prototipi) podedujemo lastnosti in obnašanje določenega razreda. Le to lahko v izpeljanem (podedovanem) razredu spremenimo/brišemo, dodamo pa lahko tudi nove lastnosti ali metode. Moč hierarhičnega modeliranja je na ta način omogočena tudi pri razvoju programske opreme.

Natančnejši pregled objektno usmerjenega programiranja in njegovih konceptov je preobsežen, da bi se mu v disertaciji v celoti posvetili. Podrobnosti lahko bralec najde v [13].

2.4 Domensko specifični programski jeziki

Nasprotno od splošno namenskih programskih jezkov je uporaba domensko specifičnih programskih jezikov (DSL) vezana zgolj na določeno domeno. Razvoj programskih sistemov z domensko specifičnimi jeziki se že v osnovi razlikuje od razvoja v splošno namenskih programskih jezikih. Pri razvoju s splošno namenskimi programskimi jeziki se morajo razvijalci najprej seznaniti z domeno, za katero razvijajo produkt, definirati bodoče uporabnike, zapisati njihove zahteve ipd. Pri domensko specifičnih jezikih teh problemov ne zasledimo, saj so te stvari že znane ter vsebovane v jeziku samem. Domensko specifični jezik predstavlja pomemben preskok v programiranju od vprašanja *"Kaj bo sistem delal?"* h *"Kako bo sistem deloval?"* [14]. Domensko specifični jeziki so v svoji notaciji (sintaksi jezika) bližje domenskim

domensko specifičen jezik	aplikacijska domena
BNF	specifikacija sintakse programskega jezika
HTML	svetovni splet
LaTeX	oblikovanje besedil
Make	razvoj programske opreme
SQL	podatkovne baze

Tabela 2.1: Primeri domensko specifičnih jezikov

strokovnjakom, ki predstavljajo ciljne uporabnike teh jezikov. Pri razvoju domensko specifičnih programskih jezikov težimo k možnosti, da za uporabo (programiranje) ni potrebno poznavanje konceptov programskih jezikov in orodij za razvoj, temveč le domeno uporabe. Na ta način poenostavimo razvoj ter zmanjšamo stroške pri razvoju programskih sistemov. Seveda pa te prednosti niso zastonj. Davek plačamo v visoki ceni razvoja domensko specifičnega jezika, ki predstavlja velik problem.

V literaturi [15, 16] zasledimo domensko specifične jezike tudi pod drugimi imeni, kot so: majhni jeziki (*little languages*), makroji, aplikacijski jeziki (*application languagues*), problemsko orientirani jeziki (*problem-oriented languagues*) ipd. Tabela 2.1 prikazuje nekaj zelo znanih domensko specifičnih jezikov in domeno, za katero so bili razviti. Večino teh jezikov pozna že povprečen poznavalec računalništva. Kot je razvidno iz tabele, uporaba domensko specifičnih jezikov ni novost. Jezik BNF [17] za specifikacijo sintakse programskih jezikov je bil razvit leta 1959 in ga uporabljamo še danes.

Razvoj domensko specifičnega jezika poteka po naslednjih korakih [16]: odločitev (*decision*), analiza (*analysis*), načrtovanje (*design*), implementacija (*implementation*) in namestitev (*deployment*). Kot pri razvoju vsakega jezika oz., če pogledamo še širše, vsakega programskega produkta, si faze ne sledijo sekvenčno. Posamezne faze imajo precejšen vpliv na drugo fazo. Npr. načrtovanje je pogosto omejeno z implementacijskimi podrobnostmi. Prav tako pa lahko v fazi implementacije ugotovimo, da smo naredili napako oz. bili nedosledni v fazi načrtovanja ali v drugih fazah. V tem primeru je potrebno ponoviti postopke v določeni fazi in razvoj jezika nadaljevati tam. Celoten proces razvoja domensko specifičnega jezika velja za težko opravilo in sodi med njegove večje slabosti.

2.4.1 Kdaj razviti domensko specifični jezik?

Obstaja večna dilema, ali naj se lotimo razvoja programa v splošno namenskem programskem jeziku ter s tem tvegamo drag razvoj in verjetno še dražje

vzdrževanje programa, ali pa naj se lotimo razvoja domensko specifičnega jezika, ki bo domenskim strokovnjakom omogočil razvoj programov in s tem nadomestil morebitne nove verzije programa razvitega v splošno namenskem programskem jeziku. Predvsem je ta dilema prisotna v zadnjem času, ko je konkurenca na trgu programskih sistemov precej ostra, kar narekuje nenehno izboljševanje produktov. Težimo torej k čimbolj racionalnemu razvoju novih produktov. Domensko specifične jezike vidimo kot eden izmed mehanizmov za učinkovito zmanjševanje stroškov pri razvoju zelo specializirane programske opreme. Končnim uporabnikom v določeni domeni torej želimo omogočiti razvijanje lastnih programskih produktov [18].

Ključno je torej odgovoriti na naslednje vprašanje: *Kdaj razviti domensko specifičen jezik? [16].* Seveda je vprašanje zelo kompleksno in je odvisno od mnogih dejavnikov. Pred sprejetjem odločitve je potrebno poznati dobre in slabe lastnosti domensko specifičnih jezikov. Na to vprašanje odgovarjamo v fazi *odločitve* razvoja domensko specifičnega jezika. Natančnejša analiza tega vprašanja je preobširna, da bi jo poglobljeno obravnavali v tej disertaciji. Na vprašanje *"kdaj"* poskusimo odgovoriti s pregledom prednosti in morebitnih slabosti domensko specifičnih jezikov.

Prednosti domensko specifičnih jezikov so:

Lažje programiranje
: Primerna domensko specifična notacija je enostavnejša kot notacija splošno namenskih programskih jezikov. Domensko specifični jeziki omogočajo izražanje rešitve na nivoju abstrakcije, primerne domeni, za katero razvijamo končno aplikacijo. Na ta način je domenskim strokovnjakom omogočen razvoj programov v takšnem jeziku, kar vpliva na izboljšanje produktivnosti končnih uporabnikov. Uporaba domensko specifičnih jezikov, zato ne sme biti podcenjena. Zaradi zapisa programa v primerni domenski abstrakciji so programi tudi bolj berljivi in lažji za učenje, kar še dodatno skrajša čas razvoja in omogoča lažje vzdrževanje.

Ponovna uporaba
: Domensko specifični jeziki že zaradi prilagojene notacije spodbujajo uporabnike k ponovni uporabnosti. V kolikor je notacija lažja in berlivejša, je tudi ponovna uporaba tako zapisanega programskega koda lažja. Domensko specifične jezike lahko uspešno razvijemo iz aplikacijskih knjižnic splošno namenskih programskih jezikov [19]. Znanje domene je tako vsebovano implicitno s skrivanjem običajnih programskih vzorcev v implementaciji ali eksplicitno s parametrizacijo v do-

mensko specifični notaciji. Tako vsak uporabnik ponovno uporablja komponente knjižnic in znanje domene.

Izvedljivost kode
Domensko specifični programi niso nujno izvedljivi.

Lažja verifikacija
V domensko specifični jezik vgrajeno znanje domene omogoča uporabnikom le–tega preverjanje pravilnosti programov na nivoju domene, kar bistveno pripomore k hitrejšemu in učinkovitejšemu razvoju.

Uporabniško programiranje
Obstaja močna povezava med končnimi uporabniki in domensko specifičnimi jeziki, saj je uporabniku s primernim znanjem domene z minimalnim trudom učenja notacije omogočeno razvijati lastne programske rešitve [18]. Tipičen primer je programiranje v elektronski preglednici z uporabo Excel-ovega makro jezika.

Seveda nobena zgodba nima samo ene plati, tako tudi domensko specifični jeziki nimajo zgolj prednosti, ampak njihova uporaba prinese tudi določene nevšečnosti. Slabe plati domensko specifičnih jezikov so naslednje:

Stroški razvoja
Razvoj domensko specifičnih jezikov je zapleten, saj je potrebno poznavanje domene, kot tudi razvoja programskih jezikov. Le redki poznajo oboje. V nasprotnem primeru je potrebno najeti strokovnjake na obeh področjih.

Stroški učenja končnih uporabnikov
Stroški učenja domensko specifičnih jezikov s strani uporabnikov niso zanemarljivi. Seveda je potrebno uporabnike (razvijalce/programerje) splošno namenskih programskih jezikov tudi izobraziti. A za razliko od uporabnikov domensko specifičnih jezikov, so le–ti računalniški strokovnjaki.

Omejena uporabnost
Domensko specifični jeziki niso razviti za vse domene, zato je njihova uporabnost zelo omejena.

Slabša učinkovitost
V večini primerov se domensko specifični programi generirajo oz. se iz domensko specifičnih programov generira izvorni kod splošno namenskega programskega jezika. Pri generiranju lahko pride do izgube učinkovitosti v primerjavi z ročno napisanim kodom.

Omejena notacija
V primerjavi s splošno namenskimi jeziki je notacija domensko specifičnih jezikov zelo omejena, kar zmanjšuje fleksibilnost programiranja.

Omejena dostopnost
Velikost uporabniške skupine, podpora jezika, standardizacija in vzdrževanje lahko sčasoma postanejo časovno zahtevna opravila.

Vsekakor je prednosti, vsaj po našem trdnem prepričanju, domensko specifičnih jezikov več kot slabosti in predstavljajo pomembne programsko-inženirske pridobitve. Če smo uspešno pretehtali prednosti in slabosti ter aplikativnost za določeno domeno ter ugotovili, da je domensko specifični jezik prava rešitev našega problema, si s tem še vedno nismo odgovorili na vprašanje *"Kako razviti domensko specifični jezik?"*. Natančen pregled vzorcev in različnih implementacijskih pristopov pri razvoju domensko specifičnih jezikov je opisan v [16, 19, 20].

Advice is what we ask for when we already know the answer but wish we didn't.

– Erica Jong –

Poglavje 3

Aspektno usmerjeno programiranje

Aspektno usmerjeno programiranje je že ob svojem nastanku, pred dobrim desetletjem, pritegnilo široko pozornost raziskovalne skupnosti. Krog raziskovalcev, ki se ukvarjamo z aspektno usmerjenim programiranjem se še vedno veča, kakor tudi področja kamor se koncepti aspektno usmerjenega programiranja širijo. V doktorski disertaciji se ukvarjamo z aspektno usmerjenimi atributnimi gramatikami, ki so nadgradnja atributnih gramatik s koncepti, poznanimi iz aspektno usmerjenega programiranja. Poglavje je namenjeno pregledu osnovnih konceptov aspektno usmerjenega programiranja in motivaciji za uporabo le–teh na področju specificiranja programskih jezikov, ki je osnovna tema doktorske disertacije.

3.1 Uvod

Razdrobljen programski kod (zapis iste funkcionalnosti v več modulih) lahko pri načrtovanju in implementaciji povzroči nemalo nevšečnosti in seveda dodatnih stroškov. Tehniki za preprečitev prepletanja programskega koda pravimo ločevanje dolžnosti, kjer posamezno funkcionalnost poskušamo načrtovati in implementirati čimbolj neodvisno. Ločevanje dolžnosti je pri načrtovanju in implementaciji programske opreme zelo pomembno. Še bolj kot pri sami implementaciji programske opreme se prednosti ločevanja dolžnosti pokažejo ob ponovni uporabi in vzdrževanju programskih produktov. Princip ločevanja dolžnosti omogoča, da se s posameznimi komponentami sistema ukvarjamo ločeno. Še več, na ta način lahko posamezne funkcionalnosti sistema razvijamo v ločenih komponentah. To nam omogoče ustrezno abstrakcijo, kapsulacijo in rahlo sklopljenost. Med uspešne pristope za ločevanje dolžnosti, ki se v zadnjih letih vedno bolj uveljavlja, spada tudi aspektno usmerjeno programiranje (AOP) oz. aspektno usmerjen razvoj programske opreme (AOSD). Aspektno usmerjeno programiranje sledi podobnim ciljem kot ostali pristopi za ločevanje dolžnosti [21]; ti cilji so naslednji:

- zmanjšati kompleksnost programske opreme (izboljšati berljivost koda, razumljivost itd.),
- lokalizacija posameznih dolžnosti, kjer se programski kod prepleta ter ta kod organizirati v ločene module,
- omejiti spremembe v programski opremi,
- izboljšati evolucijo programske opreme in
- izboljšati integracijo komponent.

Aspektno usmerjeno programiranje oz. določen aspektno usmerjen programski jezik mora razvijalcu omogočati identifikacijo, kapsulacijo, modularizacijo in kompozicijo posameznih dolžnosti programskega produkta. Z drugimi besedami AOP mora razvijalcu nuditi mehanizme za ločevanje komponent in aspektov, ki predstavljajo modularen zapis določene funkcionalnosti. Zakaj je ločevanje dolžnosti tako pomembno? Vzemimo na primer objektno usmerjen jezik, kjer razvijalec opiše dolžnost sistema skozi koncept razreda. Vseh dolžnosti sistema ni mogoče načrtovati in razviti tako, da bi bile posamezne lastnosti sistema zapisane zgolj v enem razredu, kar vodi v pojav prepletanja koda.

3.2 Koncepti aspektno usmerjenega programiranja

V nadaljevanju disertacije in tudi tega poglavja se veliko srečujemo s termini in koncepti, ki se uporabljajo v aspektno usmerjenem programiranju. V tem poglavju razložimo terminologijo, ki se uporablja v AOP in določene koncepte le–tega. Posamezni avtorji uporabljajo različno terminologijo, a se je ta zdaj v skupnosti, ki se ukvarja z AOP, že dokaj ustalila. Opisana terminologija je povzeta po knjigi [22], kjer je na enem mestu zbranih več člankov priznanih avtorjev s področja AOP.

Osnovna terminologija AOP obsega naslednje termine:

Dolžnosti (*Concerns*)
Vsak inženirski proces se ukvarja z več stvarmi ter mora zadoščati več kriterijem. Te lastnosti sistema imenujemo dolžnosti, ker jih je sistem dolžan zagotavljati. Lahko so različne, od visoko-nivojskih (npr. sistem mora biti enostavno upravljiv) do zelo nizko-nivojskih (npr. oddaljeni predmeti naj bodo predpomnjeni). Poznamo različne dolžnosti: lokalne (npr. ob pritisku določene tipke naj se sproži določena akcija), globalne; imajo vpliv na celoten sistem (npr. odzivnost sistema mora biti na določenem nivoju), za določanje obnašanja sistema (npr. vsa povpraševanja v podatkovni bazi morajo biti zabeležena) itd. Vsak sistem ima različne dolžnosti, tako da se na tem mestu ni smiselno poglabljati v opis posameznih dolžnosti, temveč ostanimo zgolj na konceptualni ravni.

Prekrivanje dolžnosti (*Crosscutting Concerns*)
Razvoj programskih rešitev zajema dolžnosti podane na nivoju uporabnika, kakor tudi dolžnosti na implementacijskem nivoju. Predvsem pri implementaciji se pogosto dogaja, da posameznih dolžnosti ni mogoče zapisati na modularni način. Implementacijsko se torej določene dolžnosti prekrivajo z ostalimi, kar v praksi rezultira k razdrobljenemu kodu za določeno lastnost (funkcionalnost sistema). Odkrivanje morebitnih prekrivanj dolžnosti je zapleteno opravilo, saj ne obstaja splošno znan formalni postopek. Prav tako pa je mogoče, da določena dolžnost implicitno vsebuje neko drugo, čeprav bi jih v normalnih pogojih morali načrtovati in implementirati ločeno. Ločevanje dolžnosti komponent sistema je bistven korak pri načrtovanju programske opreme. Če je le–to uspešno, lahko bistveno pripomore h kakovosti programske opreme. Še posebej je ločevanje dolžnosti pomembno za skalabilnost,

inkrementalni razvoj, zmanjšanje kompleksnosti in ponovno uporabnost programske opreme. V literaturi zasledimo več tehnik in orodij, ki pomagajo razvijalcem pri učinkovitejšemu načrtovanju in implementaciji programskih sistemov s stališča ločevanja dolžnosti. Naj naštejemo le nekaj rešitev (vse rešitve žal niso več v aktivni fazi raziskav): adaptivno programiranje [23] (orodje DemeterJ [24]), kompozicijski filtri [25], hiperprostori (*Hyperspaces*) [26] (orodje Hyper/J [27]) itd.

Prepletanje koda (*Code Tangling*)

Z uporabo običajnih razvojnih postopkov implementacija prekrivajočih dolžnosti rezultira k prepletanju koda za implementacijo neke dolžnosti in ostalim programskim kodom (le–ta predstavlja implementacijo druge dolžnosti ipd.). Po drugi strani pa nas dobra praksa načrtovanja programskih produktov uči načrtovati posamezne komponente čimbolj ločeno, kjer vsaka komponenta predstavlja implementacijo posamezne dolžnosti. Pri tem naj bi bila komunikacija med komponentami čimbolj enostavna. Tako načrtovani sistem je lažje implementirati in vzdrževati. Prekrivanje dolžnosti in s tem prepletanje koda to delo precej oteži, saj ne zadostuje pogojem dobre načrtovalske in implementacijske prakse. Takšen kod je razpršen po mnogih komponentah in ob morebitni spremembi (nadgradnja, vzdrževanje itd.) zahteva poseg v vse komponente, kjer je takšen kod prisoten. Ob tem je potrebno poudariti, da je programski kod v večini komponent skoraj popolnoma identičen, saj implementira isto funkcionalnost. Zadnja opisana lastnost je še posebej pomembna, saj ob rutinskem opravilu razvijalci pogosto napravijo napake, ki pa jih je zelo težko odkriti (sistem v večini primerov dela, ob določeni situaciji pa ne). Zelo pomemben aspekt prepletanja programskega koda je tudi testiranje. Vsak programski kod, ki implementira neko prekrivajočo dolžnost, mora zadostiti specifikacijam in ne sme posegati v funkcionalnost same komponente. V praksi to pomeni, da je potrebno ob vsaki spremembi funkcionalnosti komponent, testirati tudi funkcionalnost prepletenega koda in obratno.

Aspekti (*Aspects*)

Aspekt je modularna enota, ki implementira posamezno dolžnost in se prekriva z eno ali več dolžnostmi sistema. Definicija samega aspekta lahko vsebuje določen programski kod, ki definira obnašanje aspekta in navodila za klic (kdaj/kje/kako), oz. instanciranje le–tega. Aspekti so lahko, odvisno od aspektnega jezika, urejeni v hierarhije. Ponavadi vsebujejo ločena dela za specifikacijo obnašanja in način interakcije s preostalim sistemom.

Stične točke – stičišča (*Join Points*)

Stične točke definirajo mesta v strukturi oz. izvajalnemu toku programa, kamor lahko pripnemo neko dodatno funkcionalnost/obnašanje. Model stičišč (JPM) določa dovoljena stičišča in omogoča skupni okvir za načrtovanje strukture aspektov. Najpogostejše stične točke v aspektnih jezikih so klici metod. Ostala stičišča so: deklaracija/dostop/spreminjanje instančne spremenljivke, izjeme, določena stanja v izvajanju ipd. Npr., če ima določen programski jezik klic metode v modelu stičišč, to pomeni, da lahko ob klicu metode izvedemo nek dodatni kod definiran z aspektom.

V osnovi poznamo dve vrsti stičnih točk, statične in dinamične. Termina se nanašata na elemente, ki so znani pred izvajanjem programa (statični elementi) in elemente, ki so izračunljivi šele ob izvajanju programa (dinamični elementi). Statične stične točke so tiste, kjer lahko aspekte v osnovne komponente (program) vključimo v času prevajanja. Dinamične pa tiste, kjer je to mogoče šele med izvajanjem, torej moramo izvajanje programa nekako prekiniti in na določenem mestu izvesti kod aspekta.

Nasvet (*Advice*)

V nasvetu definiramo kod, ki ga želimo izvesti na določenem stičišču. Način aplikacije dodatnega koda definiranega v nasvetu se v aspektnih jezikih razlikuje, možno pa ga je ob definiciji nasveta tudi spremeniti. Mnogi aspektni jeziki omogočajo izvedbo nasvetov `pred` (*before*), `po` (*after*), `namesto` (*instead*) ali `okrog` (*around*) neke stične točke. Poskusimo na preprostem primeru pokazati kaj te deklaracije pomenijo. V primeru, da uporabimo deklaracijo *before* na stično točko (*method call*), se bo definiran nasvet izvedel pred klici vseh metod definiranih z izbrano stično točko. Na tem mestu vidimo tudi glavno razliko med aspektno usmerjenim programiranjem in "klasičnim" programiranjem, kjer so podprogrami klicani eksplicitno.

Specifikator stičišč (*Pointcut Designator*)

Specifikator stičišč (*pointcut*) definira množico stičišč. Ta lastnost aspektnih jezikov je zelo pomembna in predstavlja kvantifikacijski mehanizem, saj lahko z enim stavkom opišemo dogajanje na mnogih mestih v programu. S specifikatorjem stičišč lahko npr. opišemo vse točke v programu, kamor moramo dodati kod za beleženje morebitnih napak (*logging concern*).

Kompozicija (*Composition*)

Idejo združevanja ločeno razvitih elementov programa v programskem

inženirstvu imenujemo kompozicija. Različni programski jeziki omogočajo različne načine kompozicije, kot so: podprogrami, dedovanje, generično instanciranje itd. Zelo pomembno pri kompoziciji komponent je zagotavljanje kompatibilnosti le–teh. Vse morebitne nepravilnosti je potrebno zaznati v fazi razvoja, kar je posebej pomembno pri aspektno usmerjenem programiranju, kjer je razvijalcu še težje zagotavljati kompatibilnost in predvideti vse možne scenarije. Osnovni mehanizmi za zagotavljanje kompatibilnosti so: preverjanje tipov, podpisi klicev podprogramov, vmesniki itd.

Tkanje (*Weaving*)

Tkanje (*aspect weaving*) je proces kompozicije osnovnih modulov in funkcionalnosti v aspektih (osnovna ideja tkanja je prikazana na sliki 3.1). Pri tem procesu se tvori delujoč sistem. Modelov tkanja in samih implementacij je več in se lahko med posameznimi aspektnimi jeziki precej razlikujejo. Na splošno ločimo statično in dinamično tkanje. Oba pristopa sta primera metaprogramiranja, kjer program obravnavamo kot podatek. Običajno v določenem okolju uporabljamo samo en način tkanja, možno je uporabiti tudi oba.

Statično tkanje

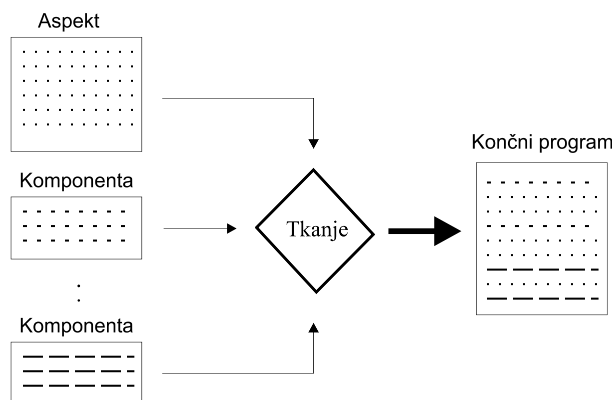
Pri statičnem tkanju spremenimo izvorni kod komponent tako, da na utrezna mesta (izbrane stične točke) vstavimo izvorni kod aspektov. Pristop je možno implementirati s preprocesiranjem, transformacijo ali s posebnim prevajalnikom. Slabost tega pristopa je močna sklopljenost komponent in aspektov (težko je ločiti med posameznimi komponentami in aspekti), oteženo pa je tudi ločeno prevajanje osnovnih komponent in aspektov. To povzroča veliko težav pri razvoju, predvsem razhroščevanju programov, kjer sledimo izvajanju prepletenega programa in ne programa, kot je bil razvit. Naslednji problem predstavlja javljanje napak, saj je izhod tkanja ponavadi izvorni kod celotne komponente, ki se prevede z običajnim prevajalnikom. Na ta način prevajalnik javi lokacijo napake v prepletenem programu. Čeprav ima ta pristop precej slabosti, je v praksi še vedno najbolj razširjen.

Dinamično tkanje

Statično tkanje na dinamične stične točke (izračunljive šele ob izvajanju programa) ni možno. V tem primeru uporabimo dinamično tkanje, ki poteka med izvajanjem programa. Dinamično tkanje ima precej prednosti pred statičnim, saj omogoča fleksibilnejšo rabo aspektov. Aspekti in komponente se v tem primeru v

sistemu pojavljajo v obliki ločenih entitet, kar bistveno pripomore k lažjemu sledenju izvajanja programa; lažje in razumljivejše pa je tudi javljanje napak. Med izvajanjem je mogoče določen aspekt uporabiti, ustaviti njegovo izvajanje, ali ga celo nadomestiti z drugim aspektom. Izvedba dinamičnega tkanja ni možna pri vseh programskih jezikih.

Ovijanje (*Wrapping*)
Ovijanje se nanaša na možnost uporabe nasvetov `pred`, `po`, `namesto` itd. na stične točke. Ovijanje nam torej predstavlja nek vsebnik, ki zagotavlja pravilno komunikacijo med aspekti in osnovno komponento. Na ta način je omogočen ločen razvoj določenih delov komponent, ki so zadolženi za implementacijo določene dolžnosti.



Slika 3.1: Tkanje aspektov in komponent

3.3 Aspektno usmerjeni programski jeziki

V prejšnjem poglavju smo se podrobneje seznanili s koncepti aspektno usmerjenega programiranja. Osnovna ideja le–tega je vpeljava aspektov na nivoju načrtovanja programske opreme. Na ravni (objektno usmerjenih, funkcijskih itd.) programskih jezikov jih ne moremo preslikati v eno izmed obstoječih abstrakcij, zato je potrebna ustrezna podpora na nivoju programskega jezika, s katero je mogoče uspešno predstaviti aspekte. Razvojno okolje za podporo aspektom je ponavadi sestavljeno iz več jezikov. Običajno imamo en komponentni (osnovni, bazni) jezik (*component language*) za opis komponent funkcionalne dekompozicije sistema in enega ali več aspektnih jezikov

(*aspect language*) za opis aspektov [28].

V zadnjem času se je razvoj programskih jezikov, katerih namen je izboljšati dekompozicijo in ločevanje dolžnosti v obstoječih programskih jezikih, kakor tudi "pravih" aspektno usmerjenih jezikov, ki že v osnovi ponujajo aspekte kot enega izmed mehanizmov za dekompozicijo sistema, precej pospešil. Danes najdemo aspektne jezike že za skoraj vse najpogosteje uporabljene programske jezike. Nekateri od teh so že prerasli okvire osnovnega jezika in le–temu dodajajo nove koncepte in za razvoj v tem jeziku ne potrebujejo več prevajalnika za osnovni programski jezik. Več o posameznih implementacijah aspektnih jezikov je dostopno na spletni strani aspektno usmerjenega programiranja [29, 30]. Na tem mestu se ne bi spuščali v obširnejšo razpravo o konceptih posameznih aspektnih programskih jezikov, našteli bomo le nekaj najvidnejših predstavnikov le–teh. Večina jezikov je nastala iz že obstoječega programskega jezika, zato tudi programske jezike klasificiramo po tem kriteriju.

- Aspektni programski jeziki, ki bazirajo na programskem jeziku java.
  - AspectJ (`http://www.eclipse.org/aspectj`),
  - CeasarJ (`http://www.caesarj.org`),
  - Logic AJ (`http://roots.iai.uni-bonn.de/research/logicaj`),
  - Javassist (`http://www.csg.is.titech.ac.jp/~javassist`),
  - InjectJ (`http://injectj.fzi.de/InjectJ/CMS/index_html`) ,
  - Steamloom (`http://www.st.informatik.tu-darmstadt.de/static/pages/projects/AORTA/Steamloom.jsp`),
  - ...
- Aspektni programski jeziki za platformo .NET.
  - Aspect.NET (`https://www.academicresourcecenter.net/curriculum/pfv.aspx?ID=6801`),
  - AspectC# (`http://www.castleproject.org/aspectsharp`),
  - Seasar.NET (`http://www.seasar.org/en/dotnet`),
  - AspectDNG (`http://sourceforge.net/projects/aspectdng`),
  - ...
- Programski jeziki, ki bazirajo na programskem jeziku C/C++.

  - AspectC++ (http://en.wikipedia.org/wiki/AspectC++),
  - FeatureC++ (http://wwwiti.cs.uni-magdeburg.de/iti_db/forschung/fop/featurec/),
  - AspectC (http://www.cs.ubc.ca/labs/spl/projects/aspectc.html),
  - ....

- Aspektni programski jeziki, ki temeljijo na programskem jeziku ruby.
  - AspectR (http://aspectr.sourceforge.net/),
  - Aquarium (http://aquarium.rubyforge.org/),
  - ...

Aspektno usmerjene različice obstajajo tudi za druge programske jezike, kot npr.: cobol, python, javascript, smalltalk, PHP, make, lisp itd.

Aspektno usmerjeno programiranje, in koncepti le–tega, se uspešno uveljavja tudi na drugih področjih, kot so:

- Razvoj novih aspektno usmerjenih jezikov.

  Ti jeziki ne temeljijo na enem izmed že obstoječih jezikov.

- Domensko specifični aspektno usmerjeni programski jeziki.

  Vedno več je razvijalcev, ki se za uspešnejši razvoj domensko specifičnih jezikov zateka h konceptom aspektno usmerjenega programiranja. V skupnosti razvijalcev domensko specifičnih aspektno usmerjenih jezikov aktivno sodelujemo tudi v naši raziskovalni skupini, in sicer kot soorganizatorji in člani programskih odborov raznih znanstvenih simpozijev [31, 32, 33].

- Vgrajeni sistemi.
- Programska ogrodja in aplikacijski strežniki.

  Koncepte aspektno usmerjenega programiranja zasledimo za raznih programskih ogrodjih [34], in aplikacijskih strežnikih [35].

3.3.1 Dimenzije načrtovanja aspektnih jezikov

Pri načrtovanju novega aspektnega jezika in pri izbiri pravega programskega jezika za razvoj moramo preučiti določene dimenzije oz. modele, katerih podporo želimo v izbranem programskem jeziku. Dimenzije, na katere moramo biti pozorni pri načrtovanju novega ali izbiri željenega aspektnega programskega jezika, so naslednje:

- Simetrija/asimetrija (*Symmetry vs. Asymmetry*). *Ali programski jezik razlikuje med "osnovnim" in "aspektnim" delom jezika?*
  Določeni aspektni programski jeziki so sestavljeni iz več jezikov, iz osnovnega ali komponentnega dela jezika in enega ali več aspektnih jezikov za dekompozicijo sistema.

- Model stičišč (*Join Point Model*). *Na katerih točkah v programu lahko vključimo obnašanje definirano z aspekti?*
  Predvsem je pomembno, da pri načrtovanju jezika preučimo, ali naj bodo stične točke statične oz. dinamične.

- Model kompozicije (*Composition mechanism*). *Kakšne jezikovne konstrukte ponuja jezik za opis načina in mesta aplikacije aspektov?*
  Nekateri jeziki ponujajo ločen jezik za opis aplikacije aspektov.

- Kvantifikacija (*Quantification*). *Kakšne mehanizme ima na voljo razvijalec za sistematično aplikacijo aspektov?*

- Kapsulacija (*Encapsulation*). *Kakšna je jezikovna podpora za omejevanje vidnosti in učinkov aspektov in njihove interakcije z ostalim programom?*

- Statična varnost (*Type safety*). *Kakšne mehanizme ima na voljo jezik za preverjanje kompatibilnosti komponent, ki jih združujemo?*

- Zavedanje o aspektih (*Obliviousness*). *Ali mora biti programski kod, na katerega želimo aplicirati aspekte, zapisan v posebni obliki, da je primeren za aplikacijo aspektov? Programerju, ki piše osnovno komponento, ni potrebno razmišljati o možnostih apliciranja aspektov na razviti programski kod.*

- Domenska specifičnost (*Domain specificity*). *Ali jezik omogoča razvoj programov za poljubno domeno?*

- Ponovna uporabnost in parametrizacija aspektov (*Reuse and aspect parametrization*). *Kakšna je možnost ponovne uporabe aspektov v različnih kontekstih?*

- Reševanje konfliktov (*Conflict resolution*). *Ali jezik pozna mehanizme za opis in razreševanje konfliktov, ki lahko nastanejo pri aplikaciji aspektov?*

- Povezava z obstoječimi jeziki (*Legacy relationships*). *Je programski jezik namenjen razširitvam obstoječih jezikov in okolij razvitih v teh jezikih?*

- Kontrola aspektov v fazi izvajanja (*Run-time aspect dynamics*). *Ali jezik nudi podporo za spremembo aspektov v času izvajanja?*

- Možnost analiz (*Analyzability*). *Ali jezik ponuja mehanizme za statično preverjanje izvornega koda?*

- Razhročevalnost (*Debugability*). *Kakšen nivo razhroščevanja, in orodja za to, nam omogoča jezik?*

- Možnost testiranja (*Testability*). *Ali je možno aspekte testirati ločeno od ostalega programa?*

- Implementacija (*Implementation mechanism*). *Kako je jezik implementiran? Kako je jezik mogoče implementirati?*

Opisane dimenzije so ločene, a ne povsem ortogonalne. Vsak razvijalec, bodisi uporabnik aspektno usmerjenega jezika ali razvijalec novega aspektno usmerjenega jezika, bi pred začetkom dela moral odgovoriti na ta vprašanja.

3.3.2 Povzetek

Kot smo že omenili na začetku poglavja, so raziskave na področju AOP zelo obširne.

- Objektno usmerjeno programiranje
  Na področju aspektno usmerjenega programiranja

- Domensko specifični programski jeziki

- *Early aspects*
  To je ena sama bedarija.

-

3.4 AspectJ

Orodje oz. programski jezik aspectJ [36, 37] izvira iz raziskav, ki jih je opravila Cristina Videira Lopes v času doktorskega študija (njena doktorska disertacija) [38]. Na začetku so aspectJ sestavljali komponentni jezik JCore in aspektna jezika COOL in RIDL. Sam programski jezik je od svojih začetkov doživel precej sprememb in nadgradenj.

AspectJ je splošno namenski aspektno usmerjen jezik, temelječ na programskem jeziku java. Temelječ na programskem jeziku java pomeni, da jezik v bistvu ni samostojen, ampak na nek način razširja ta programski jezik java s koncepti, ki omogočajo ločevanje dolžnosti. AspectJ prevajalnik proizvede popolnoma java kompatibilen izvorni ali vmesni (*bytecode*) kod, ki je izvedljiv z običajnim java virtualnim strojem (JVM).

Pri razvoju programskega jezika in podpornih (razvijalskih) orodjih so načrtovalci poskušali razviti programski jezik, ki bo:

- Splošno namenski programski jezik.

  Programski jezik ni načrtovan za specifično domeno (npr. porazdeljene aplikacije, bančne aplikacije (*transaction management*), poslovne aplikacije itd.). Podobno, kot je java splošno namenski objektno usmerjen programski jezik, je aspectJ splošno namenski aspektno usmerjen programski jezik. Prekrivajoče dolžnosti v aspectJ-u opišemo v aspektih, ki so glavna kapsulacijska enota. Le–ti omogočajo razvijalcu dovolj širok spekter funkcionalnosti za opis ločevanja dolžnosti v vseh vrstah sistemov.

- Programsko kompatibilen s programskim jezikom java.

  Vsak program zapisan v programskem jeziku java je tudi program v programskem jeziku aspectJ (aspectJ prevajalnik ga uspešno prevede).

- Omogočal podporo aspektom na nivoju programskega jezika.

  Aspekti so v programskem jeziku aspectJ prvo-razredne vrednosti, kar je bila ena izmed ključnih odločitev pri načrtovanju jezika. Razvijalcem je na ta način omogočena podobna manipulacija z aspekti, kot so vajeni z razredi. V času izvajanja se tako ustvarjajo primerki (instance) aspektov, ki hranijo stanje in obnašanje posameznega aspekta. Vsi koncepti aspektne usmerjenosti so vgrajeni v sam jezik. Ta lastnost je zelo pomembna s stališča razvijalcev, saj jim omogoča modeliranje rešitve na nivoju aspektov z uporabo aspektno usmerjenega meta-modela (*"aspect-oriented thinking"*). Ta način razmišljanja in posledično načrtovanje sistema je težji v primeru, ko imamo na voljo objektno usmerjen programski jezik (in seveda objektno usmerjen meta-model), ki mu z dodatnimi konstrukti dodajamo aspektne lastnosti.

- Enostaven za učenje.

  K zagotovitvi tega cilja veliko pripomore dejstvo, da je programski jezik aspectJ razširitev popularnega programskega jezika java. Pri razvoju se lahko tako uporabijo vsi znani konstrukti tega jezika, razvite komponente, knjižice itd.

- Inkrementalno prilagodljiv.

  Pod terminom "inkrementalno prilagodljiv" razumemo zmožnost postopnega učenja programskega jezika. Razvijalcem ni potrebno obvladati vseh konceptov za razvoj relativno zapletenih in uporabnih aplikacij.

- Enostavno integrabilen v obstoječa orodja za razvoj programske opreme.

  Sama implementacija jezika in razvijalskih orodij je načrtovana tako, da omogoča enostavno integracijo v številna razvojna okolja. Pomembno pri tej integraciji je dejstvo, da je preveden kod za programski jezik aspectJ enak prevedenemu kodu za programski jezik java in tako izvedljiv na vseh implementacijah JVM. Distribucija programskega jezika aspectJ (podobno kot JDK) vsebuje `ant` [39] opravila, ki omogočajo enostaven prehod iz prevajanja v programskem jeziku java v okolje aspectJ. Prav tako je distribuciji dodano orodje `ajdoc`, ki po vzoru `javadoc` orodja generira API dokumentacijo v formatu HTML.

Podpora za razvoj aplikacij v programskem jeziku aspectJ je že vključena v številna znana orodja, kot so: Eclipse, JBuilder, NetBeans in Emacs.

- Primeren za komercialno uporabo.

  Programski jezik aspectJ je namenjen za uporabo v realnih industrijskih projektih in ne zgolj v raziskovalne namene (npr. za preverjanje konceptov). Takšen prevajalnik mora ustrezati določenim zahtevam, kot so: robustnost, ustrezno javljanje sporočil o morebitnih napakah med prevajanjem, zadovoljiva hitrost prevajanja, ustrezna kvaliteta prevedenega vmesnega koda itd.

3.4.1 Programski jezik

Kot smo že omenili, je programski jezik aspectJ nadgradnja jave, ki ji dodaja koncept stičnih točk. V bistvu tega koncepta ne dodaja jeziku, ampak zgolj definira točke, kjer je možna aplikacija aspektov. Ostali koncepti, kot so: opis stičišč, nasveti, med-tipske deklaracije (*inter-type declarations*), so vezani na jezik aspectJ, kjer so kapsulirani z modularno enoto imenovano aspekt.

Kot je v navadi pri spoznavanju (učenju) novega jezika, si najprej poglejmo vsem poznani program *"Hello World"*, zapisan v programskem jeziku aspectJ. Najprej zapišimo programček v programskem jeziku java.

```
package test;

public class HelloWorld {

        public static void main(String[] args) {
                new HelloWorld().sayHello();
        }

        private void sayHello() {
                System.out.println("Hello World.");
        }
}
```

Z uvedbo aspekta ta programček nadgradimo, tako da se bo pred in po klicu metode *sayHello* izpisalo sporočilo o klicu.

```
package test;

public aspect Tracer {
   pointcut sayHelloMethod() :
             within(HelloWorld) && call(* sayHello(..));

   before (): sayHelloMethod() {
        System.out.println("Before ...");
   }

   after (): sayHelloMethod() {
        System.out.println("...After!");
   }
}
```

Rezultat izvajanja tega programa po uvedbi aspektov je:

```
Before ...
Hello World!
...After!
```

Stičišča

Model stičišč je v vsakem aspektnem jeziku zelo pomemben. O tem priča dejstvo, da je model stičišč koncept, ki je od nastanka programskega jezika doživel največ sprememb. Stičišča so točke (dogodki) v izvajalnem toku programa. Seveda niso zanimive vse točke. Npr. klic ali izvajanje metode je v programskem jeziku aspectJ stična točka, posamezna vrstica (npr. četrta vrstica v določeni metodi) pa ne, čeprav jo je mogoče identificirati. Stične točke so definirane v naslednjih točkah v izvajalnem toku programa:

- izvajanje metode ali konstruktorja,
- izvajanje nasveta,
- klic metode ali konstruktorja,
- branje/pisanje instančne spremenljivke,
- izvajanje upravljanja izjem (`catch` blok),
- statična inicializacija razreda in
- inicializacija objekta ali aspekta.

Čeprav se na prvi pogled zdi, da so nekatere izmed teh točk statični sintaktični konstrukti izvornega koda, je pomembno razumeti, da so te

točke dinamične in v večini primerov izračunljive šele v času izvajanja programa.

Opis stičišč

Ponavadi v aspektnih jezikih opisujemo več stičnih točk hkrati. Temu konceptu, ki opisuje množico stičišč, rečemo specifikator stičišč (*pointcut*). Definicija specifikatorja stičišč je v programskem jeziku aspectJ kompozicija treh primitivnih definicij. Prva označuje vrsto stične točke (klic metode, dostop do instančne spremenljivke itd.). Rezervirane besede za definicijo stičišč so: `call`, `execution`, `get`, `set`, `handler`, `adviceexecution`, `staticinitialization`, `preinitialization` in `initialization`. Druga išče ujemanje s stičnimi točkami na nivoju izvajalnega konteksta (Ali je določen objekt instanca razreda *HelloWorld*?). Te stične točke označimo z rezerviranimi besedami `this`, `target` in `args`. Tretja definicija pa označuje stične točke glede na obseg, le–te definiramo z rezerviranimi besedami `within`, `withincode`, `cflow` in `cflowbelow`.

Pomembna lastnost definicije specifikatorja stičišč je možnost abstrakcije in kompozicije. Stičišča opišemo z rezervirano besedo **pointcut**, kar omogoča poimenovanje le–teh. Kompozicija je zagotovljena z možnostjo kombiniranja posameznih primitivnih in poimenovanih specifikatorjev stičišč. Pri tem lahko uporabimo operatorje `&&`, `||` in `!`.

Nasvet

Pojem nasveta se v programskem jeziku aspectJ ne razlikuje od ostalih aspektnih jezikov. Z nasveti definiramo obnašanje, ki se izvede na ustreznih točkah v izvajalnem toku programa. Vsakemu nasvetu tako pridružimo še ime specifikatorja stičišč, ki te točke definira. AspectJ pozna tri vrste nasvetov: `before` (nasvet se izvede preden dosežemo stično točko), `after` (nasvet se izvede potem, ko se kod definiran s stično točko že izvede) in `around` (najmočnejši mehanizem, saj lahko kod v stični točki nadomestimo s kodom nasveta oz. kod definiran s stično točko izvedemo na poljubnem mestu v nasvetu). Ker je s specifikatorjem stičišč mogoče definirati tudi stične točke za upravljanje izvajanja izjem, nasvet `after` razširimo še z naslednjima definicijama: `after returning` (nasvet se izvede zgolj ob uspešnem izhodu iz stične točke), `after throwing` (definira izvajanje v primeru, ko se v stični točki proži izjema). Posebej pozorni moramo biti pri uporabi nasveta `after`, saj se le–ta izvede v obeh primerih.

Med-tipske deklaracije

Včasih je potrebno ločevanje dolžnosti definirati tudi na statičnem nivoju. V ta namen lahko v aspektu definiramo vpeljavo novih spremenljivk, metod, dodajanja staršev itd. v že obstoječe razrede. Na ta način spremenimo statično strukturo razreda.

Aspekti

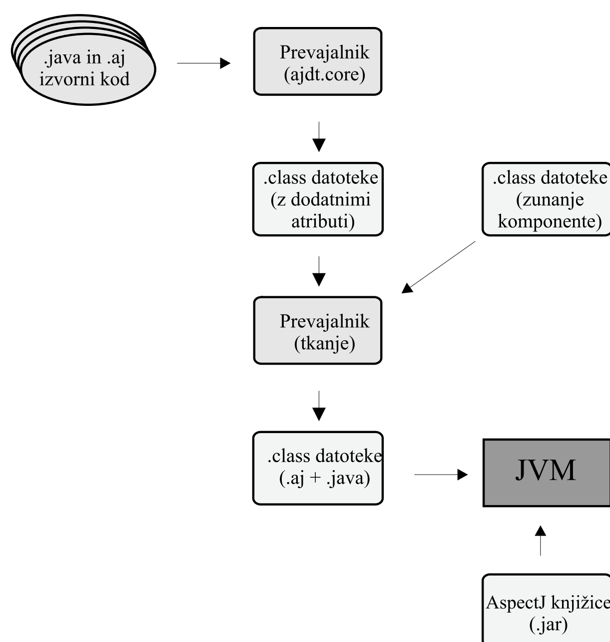
Aspekti kapsulirajo že opisane lastnosti in predstavljajo osnovno enoto za definiranje prekrivanja dolžnosti v ločenih komponentah. Podobno kot razredi, imajo tudi aspekti svoje stanje in obnašanje. Razlika je v tem, da se instanca aspekta (in posledično tudi uporaba nasvetov) ustvari implicitno. Privzeto v času izvajanja obstaja ena instanca vsakega aspekta, lahko pa eksplicitno predefiniramo življenski cikel določenega aspekta. To storimo z rezerviranimi besedami `pertarget`, `perthis`, `percflow` in `percflowbelow`, ki označujejo situacije za ustvarjanje instance aspekta.

3.4.2 Prevajalnik in aspektno tkanje

Naloga aspektnega prevajalnika za programski jezik aspectJ je prevesti java in aspectJ kod v vmesni kod, ki ga je mogoče izvesti na poljubnem JVM. Aspekti so pri tem prevedeni v razrede, ki vsebujejo metode (z nadzorovanimi imeni) za nasvete in med-tipske deklaracije. Specifikatorji stičišč in nasveti ter ostale interne informacije so v razred zakodirane z atributi v bazenu konstant. Drugi del prevajalnika predstavlja tkanje, ki je implementirano z nizom transformacij vmesnega koda. Proces prevajanja (na dokaj visokem abstraktnem nivoju) je prikazan na sliki 3.2.

3.4.3 Zaključek

AspectJ ponuja razvijalcem splošno namenski aspektno usmerjen jezik, ki je enostaven za učenje, integracijo v različna razvijalska orodja ter dovolj izrazno močan za razvoj resnih poslovnih aplikacij. Razvoj orodja se nadaljuje predvsem v smeri še boljše podpore ločevanju večjega spektra dolžnosti na nivoju jezika, grafičnih razvijalskih orodij (IDE) ter dokumentacije. Enak pristop je pričakovan seveda tudi pri ostalih aspektno usmerjenih jezikih. Ker je programski jezik aspectJ nadgradnja programskega jezika java, je podvržen nenehnim spremembam in dopolnitvam, kar je zagotovilo za napredek jezika. Število razvijalcev, objav v obliki člankov in knjig o programskem jeziku aspectJ-u se nenehno povečuje, kar zagotavlja jeziku svetlo prihodnost. Programski jezik aspectJ še vedno obstaja vodilni (referenčni) aspektno usmerjen



Slika 3.2: Shema delovanja aspectJ prevajalnika

programski jezik.

3.5 Domensko specifični aspektno usmerjeni programski jeziki

Čeprav se je razvoj aspektno usmerjenih programskih jezikov začel z domensko specifičnim jezikom [38, 40], je večina raziskovalnih in razvojnih aktivnosti usmerjena v splošno namenske aspektne jezike.

Podobne vzporednice kot med domensko specifičnimi programskimi jeziki (podrobneje smo o teh jezikih govorili v poglavju 2.4) in splošno namenskimi programskimi jeziki, je mogoče potegniti tudi med aspektnimi jeziki. Splošno namenski aspektni jeziki nudijo visok nivo abstrakcije in so namenjeni razvoju aplikacij za širok spekter domen. Kljub temu se je v praksi izkazalo, da je za opis prekrivanja dolžnosti v določenih domenah smotrneje uporabiti domensko specifični aspektni jezik [41, 42]. Domensko analizo in verifikacijo je mogoče opisati z domensko specifičnim aspektnim jezikom, s čimer preprečimo marsikatero napako [43]. V drugih primerih lahko dodajanje aspektov rezultira k neučinkovitemu kodu, tako da so potrebne domenske optimiza-

cije [44].

Iz opisanega je razvidno, da je potrebno uporabiti domensko specifične rešitve, v kolikor želimo v celoti zagotoviti ločevanje dolžnosti. To tezo potrjujejo tudi številni drugi raziskovalci. Gray je pokazal, da v različnih domenah potrebujemo različne modele dekompozicije, torej imamo opravka z različnimi načini ločevanja dolžnosti, kar privede do potrebe po domensko specifičnih aspektnih programskih jezikih [45]. V skladu s tem razmišljanjem potrebujemo tudi druge načine aspektnega tkanja (*weaving*), celo na različnih abstraktnih nivojih (npr. specifikacije, modeli, gramatike itd.). Hugunin je v svoji študiji o aspektnih jezikih definiral štiri ključna področja nadaljnega razvoja aspektnih jezikov (splošno namenskih in domensko specifičnih) [46]:

1. izboljšanje ločenega prevajanja in statičnega preverjanja,

2. povečanje izrazne moči za definicijo stičišč,

3. poenostavitev uporabe aspektov v specializiranih domenah in

4. izboljšava uporabnosti in nadgradnja orodij za razvoj z aspektno usmerjenim pristopom.

Trenutno vodilni aspektno usmerjen programski jezik aspectJ nudi širok spekter podpore ločevanju dolžnosti v obliki abstraktnih aspektnih knjižnic, a to v celoti ne zadošča za razvoj aplikacij v vseh domenah. Ravno zato ne preseneča, da so domensko specifični aspektni jeziki (DSAL) eden izmed ključnih raziskovalnih opcij na področju aspektno usmerjenega razvoja programske opreme. Nekatere izzive s katerimi se srečujejo razvijalci v splošno namenskih aspektnih jezikih, lahko uspešno nadomestimo z domensko specifičnimi aspektnimi jeziki.
Kljub vsem prednostim le–te niso zastonj. Največje slabosti domensko specifičnih aspektnih jezikov so naslednje:

- visoki stroški razvoja (pomembna je tudi odločitev, kdaj/kako se lotiti razvoja; poglavje 2.4.1),

- težavna integracija z ostalimi orodji,

- učenje končnih uporabnikov (ponavadi niso strokovnjaki s področja razvoja programov) in

- zagotavljanje podpore za ločevanje vseh dolžnosti v domeni (potreba po razvoju več jezikov za isto domeno).

Izmed teh je največji izziv pri razvoju domensko specifičnih aspektnih jezikov prav zmanjšanje stroškov razvoja. Če se razvoja jezika ne lotimo z ustreznimi orodji in metodologijo, lahko stroški razvoja presežejo prihranke pri uporabi jezika. Po našem prepričanju je večina orodij za razvoj domensko specifičnih jezikov primernih tudi za razvoj aspektnih jezikov.
V določenih domenah se za ustrezno podporo dolžnostim jezika in ločevanje le–teh pokaže potreba po razvoju več domensko specifičnih jezikov. Problem pri tem pristopu je interakcija med posameznimi jeziki in komponentnim jezikom ter javljanje napak in ustrezna podpora s strani razvijalskih orodij. Razvoj takšnega jezika (jezikov) morda ni ustrezen, saj so stroški razvoja visoki, prav tako pa končni uporabniki potrebujejo veliko napora, da osvojijo jezik.

Kljub slabostim smo mnenja, da z domensko specifičnim pristopom bistveno izboljšamo abstraktni nivo razvoja programov in na ta način končnim uporabnikom omogočimo razvoj programov v svoji domeni.

3.5.1 Model stičišč DSAL

Model stičišč aspektnih jezikov je eden izmed njihovih najpomembnejših lastnosti. Pri načrtovanju in implementaciji modela stičišč za domensko specifične aspektne jezike lahko naletimo na povsem nove pristope kot pri splošno namenskih jezikih (npr. aspectJ). Pri načrtovanju modela stičišč pri domenskem pristopu moramo odgovoriti na naslednja vprašanja:

1. Kaj so stične točke domensko specifičnega aspektnega jezika?
2. So stične točke statične ali dinamične?
3. Kako natančen/zapleten mehanizem je potreben za definicijo stičišč?
4. Kakšen jezik je potreben za opis stičišč?
5. Kaj so nasveti v izbrani domeni?
6. Kaj opisujejo nasveti? Je to zgolj razširitev obnašanja ali tudi strukture programa?
7. Kako poteka izmenjava podatkov med stičišči in nasveti (izmenjava konteksta)? Ali je potrebna parametrizacija nasvetov?

Načrtovalec domensko specifičnega aspektnega jezika mora biti pozoren na ustrezno razreševanje odvisnosti med posameznimi aspekti oz. aspektnimi deli jezika. V določenih primerih je potrebno v jezik vgraditi tudi možnost

dinamičnega dodajanja/brisanja/spreminjanja aspektov v času izvajanja programa. Za domensko specifične rešitve se ponavadi odločimo, ker želimo za določeno domeno zagotoviti programski jezik, ki bo omogočal višji nivo abstrakcije, ponovne uporabnosti in razširljivosti. Te lastnosti se morajo upoštevati pri načrtovanju domensko specifičnega aspektnega jezika, kakor tudi pri načrtovanju domensko specifičnih dolžnosti v tem jeziku. Princip abstrakcije pravi, da je možno skonstruirati abstrakcijo nad vsakim sintaksnim razredom, katerih posamezne fraze specificirajo nekakšen izračun (npr. funkcijska abstrakcija, abstrakcija podprograma, generična abstrakcija). Splošno namenski programski jeziki omogočajo velik izbor abstrakcijskih mehanizmov, medtem ko domensko specifični jeziki skušajo razvijalcem ponuditi ozek nabor specifičnih, predefiniranih abstrakcij. Razlika je logična, saj splošno namenski jeziki nikakor ne morejo nuditi abstakcijskih mehanizmov za razvoj aplikacij v vseh domenah. Ravno zato, ker so domensko specifični jeziki (ponavadi) namenjeni zgolj eni domeni, je mogoče v določeni domeni zagotoviti skoraj vse potrebne abstrakcije. Večina domensko specifičnih jezikov ne podpira splošno namenskih abstrakcijskih mehanizmov, saj so predefinirane abstrakcije dovolj za definicijo vseh aplikacij v domeni. Če izhajamo iz tega razmišljanja, lahko pričakujemo, da bodo domensko specifični aspektni jezki nudili predefinirane (fiksne) mehanizme za definiranje stičišč, nasvetov in morebitnih drugih aspektnih konstruktov domenskega jezika z omejeno možnostjo specificiranja splošnih abstrakcij. Pri domensko specifičnih aspektnih jezikih, ki so aplicirani na splošno namenski komponentni jezik, lahko pričakujemo zapletenejše konstrukte/mehanizme za definicijo stičišč, nasvetov in ostalih aspektnih lastnosti jezika. Model stičišč naj bi torej bil generičen, ponovno uporaben, razumljiv in čimbolj rahlo sklopljen s strukturo aplikacije.

If builders built buildings the way programmers wrote programs, then the first woodpecker that came along would destroy civilization.

– Murphy's Law of Technology –

Poglavje 4

Formalne specifikacije programskih jezikov

V tem poglavju si poglejmo formalni opis programskih jezikov ter faze razvoja le–teh. Formalne specifikacije so za razvoj programskih jezikov zelo pomembne, saj omogočajo enoumen zapis sintakse, in kar je še pomembnejše semantike programskega jezika. Druga prednost formalnega zapisa lastnosti programskega jezika je v možnosti avtomatskega generiranja prevajalnikov za programske jezike.

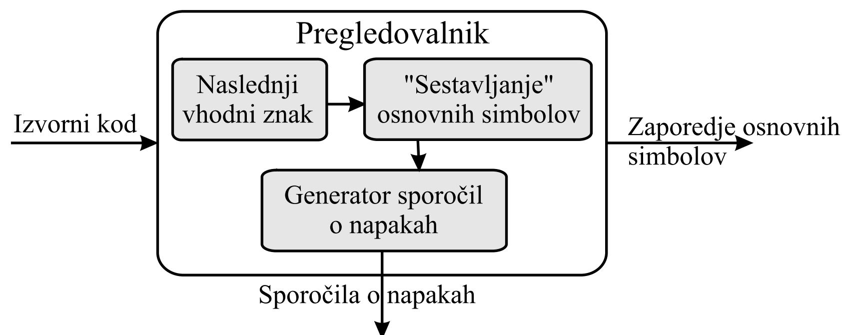
4.1 Uvod

Vsak jezik, tako naravni kot programski, formalno opišemo s sintakso, semantiko in pragmatiko. S pomočjo sintakse ugotavljamo pravilnost strukture stavkov, njihov pomen nato določimo s semantiko. Z uporabo jezikov pa se ukvarja pragmatika. S programskimi jeziki se srečujejo načrtovalci jezika, njihovi razvijalci ter končni uporabniki (programerji oz. razvijalci program-

ske opreme). Zelo pomembno je, da je jezik enoumen. Vsi uporabniki jezika morajo imeti enako predstavo o pomenu posameznih stavkov programskega jezika. Pravila, s katerimi določimo sintakso in semantiko ,so (morajo biti) natančno in enoumno določena.

4.2 Leksikalne specifikacije

Leksikalna analiza je prvi del ugotavljanja pravilnosti programa. Pri tem koraku se celotni program razbije na najmanjše leksikalne enote, pravimo jim **osnovni simboli**. To so rezervirane besede (*if, for, break, ...*), identifikatorji (imena spremenljivk, metod, razredov, ...), operatorji (+, -, %, ...) itd. Leksikalni analizator iz vhodnega programa izloči tudi znake, ki nimajo vpliva na pomen programa, kot so presledki, skoki v novo vrstico in komentarji. Program, ki opravlja leksikalno analizo, imenujemo leksikalni analizator ali pregledovalnik. Delovanje pregledovalnika je predstavljeno na sliki 4.1. V primeru, da najden osnovni simbol ni v naboru dovoljenih, vrne pregledovalnik sporočilo o tipu napake in mesto (vrstica in stolpec), kjer se napaka v izvornem kodu pojavi. Prav tako lahko pregledovalnik polni tabele, v katerih se shranjujejo podatki, ki so potrebni pri nadaljni obdelavi. Tem tabelam pravimo simbolne tabele.



Slika 4.1: Delovanje pregledovalnika

Delovanje pregledovalnika si poglejmo na primeru 4.1. Omenjeni primer pregledovalnik “razbije” na šest različnih osnovnih simbolov (rezervirana beseda, število, ločilni znak, operator, identifikator in komentar) ter bele presledke (*white spaces*), ki jih v fazi leksikalne analize odstrani in jih ne posreduje razpoznavalniku. Za bele presledke se smatrajo presledek, tabulator, skok v novo vrstico itd. Posamezne osnovne simbole pregledovalnik posreduje

razpoznavalniku v istem vrstnem redu, kot se pojavijo v izvornem kodu. Kot je razvidno iz primera, obstaja več vrst operatorjev (aritmetični `*`, relacijski `>=` in prireditveni `=`). Samo osnovni simbol ne pove razpoznavalniku dovolj informacij, zato se le–temu posreduje tudi leksikalna vrednost osnovnega simbola (*lexem*). *Lexem* je niz znakov izvornega koda, ki se ujema z vzorcem osnovnega simbola. Vsi osnovni simboli (npr. rezervirane besede) namreč nimajo enakega pomena pri razpoznavanju oz. v kasnejših fazah prevajanja. Informacije, ki se posredujejo razpoznavalniku si natančneje poglejmo v tabeli 4.1. Tabela prikazuje zaporedje osnovnih simbolov in leksemov, ki se posredujejo razpoznavalniku ob pregledovanju primera 4.1.

Primer 4.1 Stavek v programskem jeziku java

```
if (2*(radij+stevilo) >= 3.14159)
  stevilo = max_stevilo;
else
  stevilo = min_stevilo;
```

Vsak osnovni leksikalni simbol je opisan z vzorcem (*pattern*). Vzorec je pravilo, ki definira množico veljavnih leksemov za nek osnovni simbol. Očitno je, da potrebujemo formalno metodo za opis vzorcev na nedvoumen način. Najpogostejši metodi za opis osnovnih leksikalnih simbolov sta končni avtomati in regularni izrazi. Oboji imajo svoje prednosti in slabosti. Končni avtomati so človeku razumljivejši, a zato malce zahtevnejši definiranje s strani končnih uporabnikov.

4.2.1 Regularni izrazi

Osnovne simbole jezika običajno opišemo z regularnimi izrazi. Z regularnimi izrazi je mogoče definirati celotne jezike. Takšnim jezikom pravimo regularni jeziki (*regular languages*). Čeprav lahko z regularnimi izrazi definiramo le manjšo množico jezikov, so dovolj močni za opis osnovnih simbolov. Z regularnimi izrazi torej definiramo vzorec leksemov, ki bodo razpoznani za določen osnovni simbol. Vzorec je torej pravilo, ki pove, katera množica leksemov predstavlja osnovni simbol.

Vsak jezik temelji na določenem slovarju besed, le–te pa so sestavljene iz končnega nabora znakov. Znaki so elementi neke končne množice, ki ji pravimo abeceda jezika in jo označimo s Σ. Beseda w dolžine `n` je element kartezičnega produkta $w \in \Sigma^n = \Sigma \times \Sigma \times \ldots \Sigma$. Beseda w je torej sestavljena takole: $w = w_1 w_2 \ \ldots \ w_n$, kjer velja $w_i \in \Sigma$. Dolžino besede označimo z $|w|$

Osnovni simbol (naziv)	*Lexem*	Tip	Vrstica	Stolpec
reserved	`if`	1	1	1
separator	`(`	2	1	4
number	`2`	3	1	5
operator	`*`	4	1	6
separator	`(`	2	1	7
id	`radij`	5	1	8
operator	`+`	4	1	13
id	`stevilo`	5	1	14
separator	`)`	2	1	21
operator	`>=`	4	1	23
number	`3.14159`	3	1	26
separator	`)`	2	1	33
comment	`// if stavek`	6	1	35
id	`stevilo`	5	2	3
operator	`=`	4	2	11
id	`max_stevilo`	5	2	13
separator	`;`	2	2	24
reserved	`else`	1	3	1
id	`stevilo`	5	4	3
operator	`=`	4	4	11
id	`min_stevilo`	5	4	13
separator	`;`	2	4	24

Tabela 4.1: Lastnosti osnovnih leksikalnih simbolov

in velja $|w| = n$. Besedo dolžine 0 imenujemo prazna beseda in jo označimo z ε (epsilon). Znake abecede in besede združujemo z operacijo združevanja (stik). Operator stika je $\circ$, ki pa ga v praksi pogosto izpustimo. Stik je definiran na naslednji način:

$$
\begin{aligned}
w &= w_1 w_2 ... w_n \\
b &= b_1 b_2 ... b_m \\
w \circ b &= w_1 w_2 ... w_n b_1 b_2 ... b_m
\end{aligned}
$$

Če na stik gledamo kot na produkt, lahko na naslednji način definiramo potenciranje besede:

$$w^0 = \varepsilon, w^1 = w, ..., w^i = w^{i-1} w$$

Nad jeziki lahko na podoben način definiramo še naslednje operacije: unija $L \cup M$ (označuje množico besed, ki so v jeziku L in v jeziku M), stik LM (označuje množico besed, ki so stik besed jezika L z besedami jezika M) in operaciji zaprtja, iteracijo (označuje nič ali več stikov jezika L: L^*) ter pozitivno iteracijo (označuje eno ali več stikov jezika L: L^+).

$$
\begin{array}{lcl}
L \cup M & = & \{s, s \in L \vee s \in M\} \\
LM & = & \{sl, s \in L \wedge l \in M\} \\
L^* & = & \cup_{i=0}^{\infty} L^i \\
L^+ & = & \cup_{i=1}^{\infty} L^i = LL^*
\end{array}
$$

Vsak regularni izraz r označuje jezik $L(r)$. Pravila, ki definirajo regularne izraze so naslednja:

1. prazna množica je regularni izraz, ki označuje jezik $\{\}$,
2. ε je regularni izraz, ki označuje jezik $\{\varepsilon\}$,
3. a je regularni izraz, ki označuje jezik $\{a\}$,
4. če sta s in l regularna izraza, ki označujeta jezika $L(s)$ in $L(l)$ potem velja:
   (a) regularni izraz $(s)|(l)$ označuje jezik $L(s) \cup L(l)$,
   (b) regularni izraz $(s)(l)$ označuje jezik $L(s)L(l)$,
   (c) regularni izraz $(s)^*$ označuje jezik $L(s)^*$.

Odvečnih oklepajev v regularnih izrazih se lahko znebimo, če se držimo naslednjih pravil:

- operacija zaprtja ima prednost pred opercaijo stik,
- stik ima prednost pred unijo in
- vsi operatorji so levo asociativni.

Določeni regularni izrazi se pojavljajo zelo pogosto, zato zanje vpeljemo določene operacije. Nekatere od teh smo že spoznali. Poglejmo si še operacijo razredi znakov, ki je definirana na naslednji način:

razredi znakov : z znakoma [ ] zapišemo razred gramatik in velja: $[abc]$ predstavlja regularni izraz $a|b|c$. V razredih znakov lahko poleg posameznih znakov uporabimo skrajšan zapis za zaporedje znakov (npr. $[a-z] = a|b|c|...|z$).

Primer 4.2 Primeri regularnih izrazov

`a|b`	regularni izraz označuje jezik $\{a, b\}$
`a|bc*`	regularni izraz označuje jezik, ki vsebuje en znak a ali en znak b ter nič ali več znakov c (npr. $\{a, b, bc, bcc, ...\}$)
`a*`	regularni izraz označuje jezik $\{\varepsilon, a, aa, aaa, ...\}$
`(a|b)*`	regularni izraz označuje množico znakov, ki vsebuje nič ali več znakov a ali b, torej jezik $\{\varepsilon, a, b, aa, bb, abaaabb, ...\}$

Za boljše razumevanje regularnih izrazov si poglejmo nekaj osnovnih pa tudi malce zahtevnejših primerov, prikazani so na primeru 4.2.

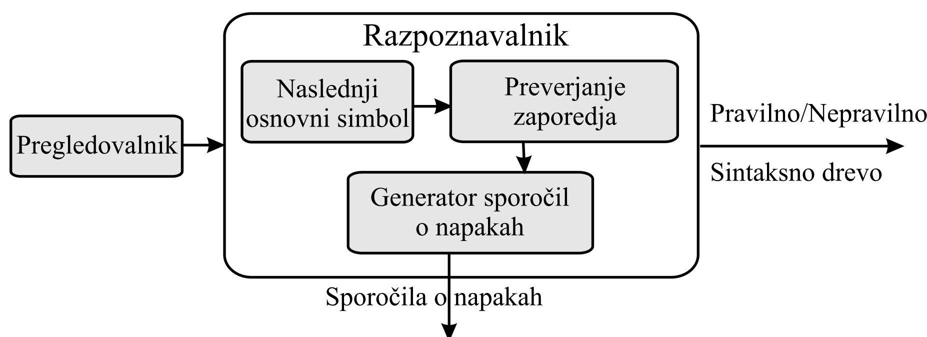
Če posamezne regularne izraze poimenujemo, govorimo o regularnih definicijah. Regularna definicija je zaporedje poimenovanih regularnih izrazov. Identifikatorje (imena) regularnih izrazov lahko uporabimo pri definiranju naslednjih regularnih definicij (primer 4.3).

Primer 4.3 Primeri regularnih definicij

znak	$\rightarrow$	`[A-Za-z]`
cifra	$\rightarrow$	`[0-9]`
identifikator	$\rightarrow$	`znak(znak|cifra)*`
int	$\rightarrow$	`cifra+`

4.3 Sintaksne specifikacije

Sintaksna analiza je naslednja faza prevajanja in sledi leksikalni analizi. Naloga faze je preverjanje sintaksne (slovnične) pravilnosti programa in jo opravlja razpoznavalnik. Preverjamo pravilnost zaporedja osnovnih leksikalnih simbolov, ki jih vrne leksikalni analizator. Izhod sintaksnega analizatorja je informacija o tem ali je bilo razpoznavanje uspešno (v primeru, ko razvijamo prevajalnik vrne tudi interno podatkovno strukturo, ki predstavlja sintaksno drevo vhodnega programa, le–ta se uporabi pri semantični analizi). Ob neuspešnem razpoznavanju pa vrne tudi seznam napak. Informacija o sintaksnih napakah je za razvijalce programov še posebno pomembna, saj je od primernosti te informacije odvisna hitrost odkrivanja storjenih napak pri pogramiranju. V nekaterih primerih razvijalci prevajalnikov razpoznavalnik "obogatijo" s semantičnimi akcijami, tako da se hkratno z razpoznavanjem opravlja tudi semantična analiza. Delovanje razpoznavalnika je predstavljeno na sliki 4.2.



Slika 4.2: Delovanje razpoznavalnika

Ta faza razvoja prevajalnika je še posebej pomembna, saj zraven implementacije razpoznavalnika obsega tudi načrtovanje sintakse programskega jezika. "Oblika" sintakse programskega jezika je zelo pomembna za njegov uspeh. Prav tako je zelo pomembno, da je sintaksa enostavna za učenje, berljiva, ne vsebuje preveč konstruktov itd. Prav sintaksa programskega jezika je pogosto vzrok za uporabo določenega programskega jezika. Programerji se raje odločajo za programske jezike s preprostejšo in berljivejšo sintakso. Zelo pomembna pa je tudi enoumnost sintakse. V preteklosti je prav pomankljiva sintaksa botrovala številnim napakam v programih.

4.3.1 Gramatike

Glavni vir teoretičnega ozadja pri razvoju razpoznavalnika predstavlja teorija naravnih jezikov. Teorija jezikov je bila že skoraj v celoti razvita, ko so jo povzeli znanstveniki s področja računalništva in jo aplicirali na razvoj prevajalnikov za programske jezike. Pri razvoju prevajalnikov je ta teorija uporabljena za predstavitev gramatike (slovnice) programskega jezika, kasneje pa je ta gramatika uporabljena pri implementaciji razpoznavalnika. Z gramatiko je torej predstavljena struktura oz. sintaksa programskega jezika. V tem poglavju spoznamo gramatike ter se seznanimo s formalnim zapisom gramatik za programske jezike.

Neformalna definicija gramatike bi se glasila takole: *gramatiko sestavlja končno mnogo pravil, s katerimi lahko generiramo neskončno mnogo stavkov.* Pri naravnih jezikih so stavki sestavljeni iz besed. V programskih jezikih pa so programi oz. programski stavki sestavljeni iz osnovih leksikalnih simbolov, ki smo jih spoznali v prejšnjem poglavju. Vsi stavki, tako v naravnem kot v programskem jeziku, so del nekega jezika, če so v skladu s podano gramatiko.

Definicija 4.3.1 *Gramatika je četverka:* $G =\lt N, T, P, Z\gt$*, kjer je :*

N	:	*končna množica neterminalnih ali nekončnih simbolov,*
T	:	*končna množica terminalnih ali končnih simbolov,* $T \subseteq \Sigma^*$ *(terminalni simbol je sinonim za osnovni leksikalni simbol),*
P	:	*množica produkcij oz. končna neprazna podmnožica relacije* $(T \cup N)^* N (T \cup N)^* \rightarrow (T \cup N)^*$ *, produkcija je oblike:* $\alpha \rightarrow \beta$*, kjer sta* α *in* β *poljubno zaporedje terminalnih ali neterminalnih simbolov,*
Z	:	*začetni simbol,* $Z \in N$*.*

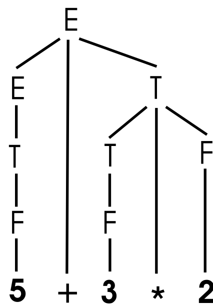
Terminalne simbole (terminale) pišemo z malimi in neterminalne simbole (neterminale) z veliki črkami. Elemente $(T \cup N)^*$ (simboli gramatike) označujemo z malimi grškimi črkami. Zaporedje terminalnih simbolov zapišemo s T^* in zaporedje neterminalnih simbolov z N^*. Kaj pa v bistvu sploh so neterminali, terminali in produkcije? Kot smo že omenili, je terminal sinonim za osnovni leksikalni simbol, z neterminali pa si pomagamo pri definiranju in poimenovanju produkcij. Produkcija pomeni, da lahko vsak niz na levi strani zamenjamo z nizom na desni strani produkcije. Vsak stavek jezika je sestavljen tako, da začnemo pri začetnem simbolu (neterminalu) in razvijamo produkcije, dokler ne ostanejo zgolj terminali. Množica vseh stavkov, ki so lahko generirani iz gramatike G imenujemo jezik izpeljan iz gramatike G.

Gramatiki *G1* in *G2* sta ekvivalentni, če velja $L(G1) = L(G2)$ (definirata isti jezik). Gramatika je levo rekurzivna, če vsebuje vsaj eno produkcijo oblike: $A \Rightarrow^*_{lm} A\alpha$. Desno rekurzivna pa je gramatika takrat, ko vsebuje vsaj eno produkcijo oblike: $A \Rightarrow^*_{lm} \alpha A$.

Kot primer si poglejmo gramatiko preprostega jezika aritmetičnih izrazov (primer 4.4) in drevo izpeljav te gramatike za primer `5 + 3 * 2` (slika 4.3).

Primer 4.4 Gramatika za aritmetične izraze

```
E  →  E + T
E  →  E - T
E  →  T
T  →  T * F
T  →  T / F
T  →  F
F  →  ( E )
F  →  #int
```



Slika 4.3: Drevo izpeljav za aritmetični izraz `5 + 3 * 2`

Klasifikacija gramatik

Teorija jezikov ima kot veja računalništva svoje začetke v sredini 50 let, ko je Noam Chomsky podal matematični model gramatik za naravne jezike [47]. Chomsky je klasificiral gramatike v štiri razrede glede na omejitve pri obliki produkcij. Razredi gramatik so naslednji:

- G0 – gramatike brez omejitev (*Unrestricted Grammars*).
- G1 – kontekstno odvisne gramatike (*Context-Sensitive Grammars*).

  Pri tej gramatiki lahko v določeni izpeljavi simbola zahtevamo odvisnost od njegove okolice. Prav tako te gramatike nimajo praznih produkcij ($\mathtt{A} \rightarrow \varepsilon$) ter generirajo kontekstno odvisne jezike.

- G2 – kontekstno neodvisne gramatike (*Context-Free Grammars*).

  Produkcije konteksno neodvisnih gramatik so oblike: $\mathtt{A} \rightarrow \beta \wedge \mathtt{A} \in \mathtt{N}$. Levo stran predstavlja en sam neterminal, izpeljava le tega pa ni odvisna od okolice. Gramatika lahko vsebuje prazne produkcije ($\mathtt{A} \rightarrow \varepsilon$).

- G3 – regularne gramatike (*Regular Grammars*).

  Regularne gramatike lahko imajo na desni strani le en sam terminal ali neterminal, ki mu sledi natanko en terminal. Produkcije regularne gramatike imajo obliko (prikazan je primer desno regularne (linearne) gramatike):

  $\mathtt{T} \rightarrow \mathtt{t\ V}$
  $\mathtt{T} \rightarrow \mathtt{t}$
  $\mathtt{T} \rightarrow \varepsilon$

  kjer je $\mathtt{t} \in \Sigma$ in $\mathtt{T}, \mathtt{V} \in \mathtt{N}$.

Pri razvoju programskih jezikov se uporabljata predvsem zadnja dva razreda gramatik. Čeprav se s kontekstno neodvisnimi gramatikam ne da opisati celotne sintakse programskega jezika, se najpogosteje uporablja prav ta razred gramatik. Učinkovit algoritem za razpoznavanje konteksno odvisnih gramatik namreč ne obstaja.

Metajezik BNF

Metajezik[1] s katerim opisujemo sintakso programskih jezikov se imenuje BNF (*Backus-Naur Form*) [17]. Notacija se imenuje po avtorjih J. W. Backus-u (sodeloval je pri razvoju programskega jezika FORTRAN) in P. Naur-u (sodeloval je pri razvoju programskega jezika Algol-60). V notaciji BNF se produkcije z enakim neterminalom na levi strani združijo, njihove desne strani pa se ločijo z metasimbolom `|`, ki predstavlja alternacijo. Prav tako je metasimbol $\rightarrow$ zamenjan z metasimbolom `::=`. Produkcije kontekstno neodvisne gramatike imajo sedaj naslednjo obliko:

$$\texttt{A ::= } \alpha_1 \texttt{ | } \alpha_2 \texttt{ | .. | } \alpha_n, \text{ kjer velja } \texttt{A} \in \texttt{N} \text{ in } \alpha_i \in \texttt{(T} \cup \texttt{N)}^*$$

Na primeru 4.5 je prikazana gramatika aritmetičnih izrazov (primer 4.4) zapisana v BNF notaciji.

Primer 4.5 Gramatika za aritmetične izraze zapisana v BNF notaciji

```
E  ::=  E + T | E - T | T
T  ::=  T * F | T / F | F
F  ::=   ( E ) | #int
```

Za zapis sintakse programskih jezikov se uporablja še notacija EBNF (*extended BNF*), ki vpelje dodatna metasimbola `{}` in `[]`, s katerima opišemo iteracijo in opcijo.

Produkciji oblike: $\texttt{A ::= } \alpha\texttt{A | } \varepsilon$ krajše zapišemo s produkcijo $\texttt{A ::= \{}\alpha\texttt{\}}$.

Produkciji oblike: $\texttt{A ::= } \alpha \texttt{ | } \varepsilon$ krajše zapišemo s produkcijo $\texttt{A ::= [}\alpha\texttt{]}$.

[1] Metajezik je jezik za opisovanje drugih jezikov. Simbole metajezika imenujemo metasimboli.

4.4 Semantične specifikacije

Semantika določa pomen programu. S semantično analizo specificiramo pomen stavkov programskega jezika, ki jih pravilno razpoznamo v fazi sintaksne analize. Programu, ki opravlja semantično analizo, pravimo evaluator, postopku pa evaluacija. Semantiko lahko podajamo na različne načine, najlažji je naravni jezik. Slabost podajanja semantičnega opisa v naravnem jeziku je njegova dvoumnost. S takim opisom obstaja velika nevarnost dvoumnega pomena programskega jezika, kar lahko predstavlja veliko oviro pri implementaciji jezika ter vodi v številne programske napake. Na žalost še ne obstaja enoten standarden zapis semantike programskih jezikov, poznamo pa nekaj formalnih metod za opis semantike (atributne gramatike [48], denotacijska semantika [49, 50], operacijska semantika [51], aksiomatska semantika [52], algebrajska semantika [53] itd.). V nadaljevanju spoznamo le nekaj od teh.

4.4.1 Atributne gramatike

Atributne gramatike (AG) je v poznih 60-ih predstavil D. E. Knuth [48]. Od predstavitve so bile uporabljene v mnogih vejah računalništva [54, 55, 56] ter se izkazale za zelo praktično uporabne. Njihova uporabnost se izkaže predvsem pri specificiranju semantike programskih jezikov ter avtomatskem generiranju prevajalnikov/interpreterjev iz tako podanih semantičnih specifikacij, uporabljene pa so tudi na področjih kot so: programsko inženirstvo, porazdeljeno programiranje, logično programiranje, podatkovne baze, vmesniki naravnega jezika, vizualno programiranje, razpoznavanje vzorcev itd. AG so nadgradnja kontekstno neodvisnih gramatik, kjer vsakemu simbolu dodamo množico atributov, ki predstavljajo informacijo o semantiki. Vrednost atributov je zapisana s semantičnimi pravili (semantične funkcije), ki so dodana vsem produkcijam gramatike. Semantične funkcije so lokalne v okviru ene produkcije in določajo pravila za izračun atributov. AG je torej sestavljena iz treh komponent: kontekstno neodvisne gramatike G, množice atributov A in množice semantičnih funkcij R: $AG = (G, A, R)$.

- $G = (N, T, P, Z)$, kjer sta N in T množici neterminalnih in terminalnih simbolov; $Z \in N$ je začetni simbol, ki se pojavi samo na levi strani prve produkcije; P je množica vseh produkcij, v kateri elementi (rečemo jim tudi simboli gramatike) $V \in N \cup T$ nastopajo v obliki parov $X \to \alpha$, kjer $X \in N$ in $\alpha \in V^*$. Prazni simbol ne levi strani produkcije označimo s simbolom ε.

- Vsakemu simbolu $X \in V$ je pridružena množica atributov $A(X)$, ki je dalje deljena na dve ločeni podmnožici $I(X)$ (podedovani atributi) in

$S(X)$ (pridobljeni atributi). Množico vseh atributov gramatike zapišemo z A in velja $A = \bigcup A(X)$.

- Množica semantičnih funkcij R je definirana v dosegu ene produkcije. Produkcija $p \in P, p : X_0 \rightarrow X_1 ... X_n (n \geq 0)$ ima primerek atributa $X_i.a$, če velja $a \in A(X_i), 0 \leq i \leq n$. Končna množica semantičnih funkcij R_p je pridružena produkciji p z natanko eno semantično fukcijo za vsak primerek pridobljenega atributa $X_0.a$ in natanko eno semantično funkcijo za vsak primerek podedovanega atributa $X_i.a, 1 \leq i \leq n$. R_p je torej množica semantičnih funkcij oblike $X_i.a = f(y_1, ..., y_k), k \geq 0$, kjer je $y_j, 1 \leq j \leq k$ primerek atributa v p in f semantična funkcija. V semantični funkciji $X_i.a = f(y_1, ..., y_k)$ je primerek atributa $X_i.a$ odvisen od vseh primerkov atributov $y_j, 1 \leq j \leq k$. Množico vseh semantičnih funkcij zapišemo z R in velja $R = \bigcup R_p$.

Pomen programa (vrednost pridobljenih atributov začetnega simbola) je definirana s postopkom, ki ga imenujemo evaluacija (*attribute evaluation*). V postopku se izračunajo vrednosti za vse primerke atributov v vseh vozliščih semantičnega drevesa izpeljav (*attributed semantic tree*) za posamezen program, zapisan v definiranem programskem jeziku.

Za boljše razumevanje konceptov si poglejmo semantiko jezika aritmetičnih izrazov (primer 4.5, stran 50)[2] zapisano z atributnimi gramatikami. Atributna gramatika aritmetičnih izrazov je sestavljena iz:

- kontekstno neodvisne gramatike $G = (\{\#int, +, -, *, /, (,)\}, \{Z, E, T, F\}, P, Z)$, kjer je $\#int$ regularna definicija $int = [0-9]+$,

- množice atributov $A = \{Z.vrednost, E.vrednost, T.vrednost, F.vrednost\}$ in

- množice semantičnih pravil R.

Množico produkcij P bomo zapisali skupaj s pripadajočimi semantičnimi funkcijami iz množice R. Vsakemu neterminalnemu simbolu iz množice N smo dodali pridobljen atribut $vrednost \in A(X)$. Produkcije s semantičnimi funkcijami so opisane v naslednji tabeli:

[2] Gramatiki smo dodali začetni neterminalni simbol Z.

produkcije	semantične funkcije
$Z ::= E$	$Z.vrednost = E.vrednost$
$E ::= E + T$	$E_0.vrednost = E_1.vrednost + T.vrednost$
$E ::= E - T$	$E_0.vrednost = E_1.vrednost - T.vrednost$
$E ::= T$	$E.vrednost = T.vrednost$
$T ::= T * F$	$T_0.vrednost = T_1.vrednost * F.vrednost$
$T ::= T/F$	$T_0.vrednost = T_1.vrednost/F.vrednost$
$T ::= F$	$T.vrednost = F.vrednost$
$F ::= (E)$	$F.vrednost = E.vrednost$
$F ::= \#int$	$F.vrednost = \#int.lexem$

4.4.2 Denotacijska semantika

Denotacijska semantika temelji na matematičnih osnovah. V začetku sedemdesetih sta jo predstavila Christopher Strachey in Dana Scott, korenine pa najdemo v Churchovem lambda računu [50] iz leta 1932. S pomočjo denotacijske semantike je mogoče napovedati obnašanje programa brez njegovega dejanskega izvajanja. Denotacijska semantika ne priredi pomena zgolj celemu programu, ampak vsaki frazi programskega jezika (vsakemu izrazu, ukazu, deklaraciji itd.). Pomen vsake fraze (imenujemo jo denotacija; od tod tudi ime) je določen s pomeni podfraz.

Uporabnost denotacijske semantike ni zgolj nedvoumnost zapisa semantike programskega jezika, temveč jo uporabljamo tudi pri sklepanju o lastnostih posameznega jezika in pri implementaciji le teh. Zapis semantike je na moč podoben funkcijskim programskim jezikom. Na semantične enačbe lahko gledamo kot na algoritem, ki interpretira določen konstrukt programskega jezika, na celoten denotacijski zapis pa kot na interpreter celotnega programskega jezika. Zaradi te lastnosti je zelo primerna za načrtovanje in implementacijo prototipnih jezikov, saj že v zgodnji fazi dobimo povratno informacijo.

Za prikaz zapisa semantičnih funkcij v denotacijski semantiki si poglejmo semantiko jezika aritmetičnih izrazov s primera 4.5. Gramatiki dodajmo naslednje frazi:

$$\begin{array}{lll} \text{E} & \in & \text{Izraz} \\ \text{c} & \in & \text{Številka} \end{array}$$

Pri definiranju semantike za podani jezik potrebujemo domeno celih števil: Integer = {..., –2, -1, 0, 1, 2, ...} in naslednje pomožne funkcije:

vsota	:	Integer x Integer	$\rightarrow$	Integer
razlika	:	Integer x Integer	$\rightarrow$	Integer
produkt	:	Integer x Integer	$\rightarrow$	Integer
deljenje	:	Integer x Integer	$\rightarrow$	Integer

Za vsako frazo jezika zapišemo semantično funkcijo, s katero določimo njen pomen. Pomen izraza je celo število, številka pa označuje naravno število. Formalno te stavke zapišemo z naslednjimi semantičnimi funkcijami:

ovrednoti : Izraz $\rightarrow$ Integer
vrednost : Številka $\rightarrow$ Integer

Gramatika vsebuje osem (8) različnih vrst izrazov. Za vsakega je potrebno definirati semantično funkcijo **ovrednoti**.

$$\textbf{ovrednoti}\ [\![E_1 + E_2]\!] = \text{vsota}(\textbf{ovrednoti}\ [\![E_1]\!], \textbf{ovrednoti}\ [\![E_2]\!])$$
$$\textbf{ovrednoti}\ [\![E_1 - E_2]\!] = \text{razlika}(\textbf{ovrednoti}\ [\![E_1]\!], \textbf{ovrednoti}\ [\![E_2]\!])$$
$$\textbf{ovrednoti}\ [\![E_1 * E_2]\!] = \text{produkt}(\textbf{ovrednoti}\ [\![E_1]\!], \textbf{ovrednoti}\ [\![E_2]\!])$$
$$\textbf{ovrednoti}\ [\![E_1 / E_2]\!] = \text{deljenje}(\textbf{ovrednoti}\ [\![E_1]\!], \textbf{ovrednoti}\ [\![E_2]\!])$$
$$\textbf{ovrednoti}\ [\![c]\!] = \text{vrednost}\ [\![c]\!]$$

Kot smo že omenili, je pomen celotnega programa sestavljen iz pomena določenih definiranih fraz. Pomen fraz je definiran s pomenom podfraz itd. Na ta način lahko s pomočjo definiranih denotacij določimo pomen programu $5 + 3 * 2$. To storimo na naslednji način:

$$\textbf{ovrednoti}\ [\![5 + 3 * 2]\!]$$
$$= \text{vsota}(\textbf{ovrednoti}\ [\![5]\!], \textbf{ovrednoti}\ [\![3 * 2]\!])$$
$$= \text{vsota}(\textbf{ovrednoti}\ [\![5]\!], \text{produkt}(\textbf{ovrednoti}\ [\![3]\!], \textbf{ovrednoti}\ [\![2]\!]))$$
$$= \text{vsota}(\textbf{vrednost}\ [\![5]\!], \text{produkt}(\textbf{vrednost}\ [\![3]\!], \textbf{vrednost}\ [\![2]\!])$$
$$= \text{vsota}(5, \text{produkt}(3, 2))$$
$$= \text{vsota}(5, 6)$$
$$= 11$$

4.4.3 Algebrajska semantika

Temelje algebrajske semantike (algebrajskih specifikacij) [57, 58] najdemo v abstraktni algebri. Programski jezik opišemo v obliki končnih struktur, operacij nad strukturami in relacijami med operacijami. Vsak izraz, stavek itd. programskega jezika je tako objekt v algebri. Glavna ideja algebrajske semantike je v poimenovanju različnih objektov in operacij nad objekti

programskega jezika ter nato uporaba algebrajskih aksiomov za opis karakteristik teh lastnosti. Na posamezne objekte lahko gledamo kot na module. Prednost takšnega zapisa je dekompozicija celotnega semantičnega zapisa v relativno majhne enote.

Na primeru si poglejmo algebrajsko specifikacijo semantike aritmetičnih izrazov naravnih števil.

```
module Izrazi
  exports
    sorts naturals
    functions
      succ : natural → natural;
      pred : natural → natural;
      add : natural x natural → natural;
      sub : natural x natural → natural;
      mul : natural x natural → natural;
      div : natural x natural → natural;
    end functions
    operations
      succ(_, _) : natural → natural
      add(_, _) : natural, natural → natural
      sub(_, _) : natural, natural → natural
      mul(_, _) : natural, natural → natural
      div(_, _) : natural, natural → natural
    end operations
  end exports
  variables
    m, n: natural;
  equations
    [N1] add(m, 0) = 0
    [N2] add(m, succ(n)) = succ(add(m, n))
    [N3] sub(m, succ(n)) = pred(sub(m, n))
    [N4] sub(m, 0) = m
    [N5] sub(succ(m), succ(n)) = sub(m, n)
    [N6] mul(m, 0) = 0
    [N7] mul(m, 1) = m
    [N8] mul(m, succ(n)) = add(m, mul(m, n))
    [N9] div(m, 0) = napaka
    [N10] div(0, succ(n)) = 0
    // predpostavimo, da velja m > succ(n)
```

[N11] div(m, succ(n)) = succ(div(sub(m, succ(n)),succ(n)))
end equations
end Izrazi

4.5 Specifikacijski jeziki

Za specifikacijo realnih programskih jezikov in učinkovito implementacijo programskih jezikov (ročno ali z generatorjem prevajalnikov) matematičen formalen zapis ni zadosten. Potrebujemo poseben zapis in strukture, ki primerno dvignejo abstraktni nivo zapisa same gramatike, pri tem pa ohranijo formalnost osnovnega koncepta. Glavni argumenti za uvedbo specifikacijskih jezikov so naslednji:

- specifikacije so preobsežne (lahko celo obsežnejše kot sama implementacija jezika),
- velika redundanca zapisa (npr. pri atributnih gramatikah je problem v lokalnosti semantičnih funkcij) ter slaba razumljivost specifikacij in
- pomanjkanje strukturiranosti zapisa.

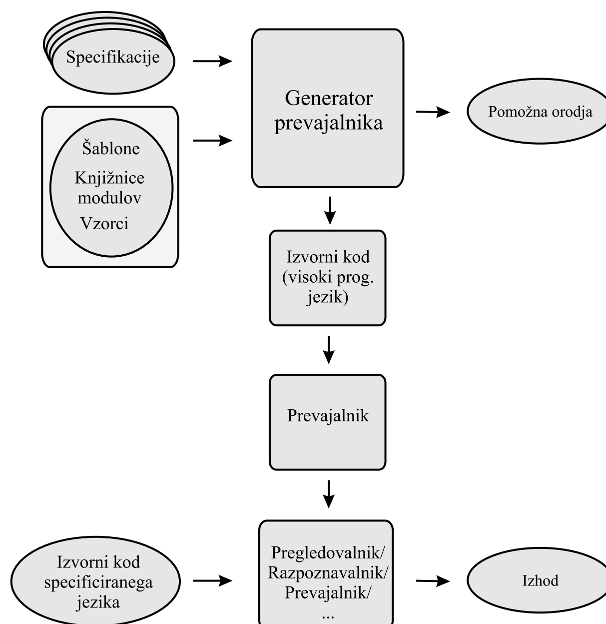
V ta namen so se razvili različni specifikacijski jeziki, ki so v osnovi implementacija določenega koncepta, nadgrajeni s konstrukti, ki odpravljajo slabosti njihove slabosti in ponavadi omogočajo avtomatsko generiranje prevajalnika iz takšnih specifikacij. Napram teoretičnim konceptom za specifikacijo programskih jezikov, pričakujemo od specifikacijskih jezikov naslednje prednosti:

- možnost avtomatskega generiranja razpoznavalnika/prevajalnika in drugih orodij (urejevalnik, razhroščevalnik itd.),
- višji abstraktni nivo zapisa programskega jezika,
- možnost modularnega zapisa,
- ponovna uporabnost specifikacij,
- možnost inkrementalnega razvoja itd.

V nadaljevanju si bomo pogledali nekaj primerov takšnih specifikacijskih jezikov in orodij za avtomatsko generiranje razpoznavalnikov/prevajalnikov, ki te jezike uporabljajo.

4.5.1 Orodja za avtomatski razvoj prevajalnikov

Razvoj in sama implementacija prevajalnika sodi med težje naloge, zato je težnja k avtomatizaciji generiranja prevajalnikov razumljiva. Med generatorje prevajalnikov štejemo orodja, ki popolnoma avtomatizirajo implementacijo prevajalnika. Avtomatiziran je lahko celoten postopek ali samo določena faza implementacije (leksikalna analiza, sintaksna analiza itd.). Generator dobi na vhodu formalni opis programskega jezika oz. le določenega dela (npr. generator pregledovalnikov dobi na vhodu formalni opis osnovnih leksikalnih simbolov). Izhod pa je ponavadi izvorni kod (lahko že tudi preveden) pregledovalnika, razpoznavalnika itd. Okvirna shema delovanja generatorjev prevajalnikov je prikazana na sliki 4.4. Najprej so se raziskovalci osredotočili



Slika 4.4: Osnovna shema delovanja generatorjev prevajalnikov

na razvoj generatorjev pregledovalnikov in razpoznavalnikov. Najbolj znana sta vsekakor Lex in Yacc [59], ki sta del standardnega okolja UNIX. Sledila so orodja za generiranje programskega koda iz ponavadi ločenih sintaksnih in semantičnih specifikacij (Centaur [60], LDL[61], SMoLCS [62] itd.). Nekaj od teh projektov je že opuščenih, a intenzivne raziskave na tem področju še potekajo. Nastajajo nova orodja z novimi idejami, koncepti za učinkovitejši in kakovostnejši razvoj programskih jezikov. Omenimo le nekaj teh orodij: ANTLR [63], ASF+SDF [64, 65, 66], SmartTools [67], JastAdd [68], LISA

[69, 70]. V tabeli 4.2 so prikazana nekatera od naštetih orodij in formalna metode, ki jih uporabljajo pri generiranju prevajalnika.

Na spletni strani [75] je predstavljen seznam generatorjev razpoznavalnikov. Nekateri med njimi omogočajo tudi podajanje semantike programskega jezika in tako tudi generiranje prevajalnikov.

Večina orodij za avtomatsko generiranje prevajalnikov podpira zgolj formalno načrtovanje in definicijo pregledovalnika in razpoznavalnika, medtem ko puščajo razvijalcem možnosti razširitve generiranega razpoznavalnika s semantiko. Semantiko v tem primeru opišemo na neformalen način. Seveda obstaja tudi nekaj orodij, ki podpirajo formalen zapis semantike programskega jezika. Nekatere od teh smo omenili že v tabeli 4.2, o drugih pa bomo v nadaljevanju še spregovorili.

Posebej zanimiva veja generiranja razpoznavalnikov predstavlja t.i. sklepanje o gramatikah (*Grammar Inference*) [76], ki sodi na področje strojnega učenja. Področje se ukvarja z možnostjo generiranja razpoznavalnika iz podanih primerov programov. Na podlagi podanih pravilnih in nepravilnih programov je mogoče sklepati o gramatiki jezika in zgenerirati razpoznavalnik za najdeno gramatiko. Področje ni novo, a so raziskovalci šele nedavno prišli do obetajočih rezultatov (uspešno najdena gramatika za večje primere) [77, 78, 79].

4.5.2 Ponovna uporabnost specifikacij programskih jezikov

V računalništvu in programskem inženirstvu pomeni pojem "ponovna uporabnost" zmožnost uporabe že razvitih komponent oz. delov programskega koda v novih programskih produktih. Ponovna uporaba lahko bistveno zmanjša čas ter stroške razvoja. Bistveno pa lahko pripomore tudi h kvaliteti, saj so večkrat uporabljene komponente temeljiteje testirane in zato zanesljivejše. S tem se bistveno zmanjšajo stroški vzdrževanja, ki predstavljajo glavnino stroška razvoja programske opreme. Seveda pa prednosti, ki jih prinaša ponovna uporaba, niso zastonj (*There Is No Such Thing As A Free Lunch [80]*). Veliko napora in znanja je potrebnega, da se programska oprema razvije po načelih ponovne uporabe. Ker spada razvoj programskih jezikov med težje discipline računalniške znanosti, je ponovna uporaba še toliko bolj pomembna. S primernim pristopom in podpornimi orodji si lahko obetamo bistveno lažji in hitrejši proces razvoja programskih jezikov ter "čistejše" koncepte le–teh. Pri ponovni uporabi specifikacij programskih jezikov si želimo zagotoviti predvsem naslednje:

- enostavno razširjanje obstoječih specifikacij z novimi koncepti in

Orodje	Sintaksa/semantika	Opis
ANTRL [63]	EBNF/sintaksno usmerjene preslikave	Semantika je opisana s poljubnim jezikom (java, C++, C#).
ASF+SDF [64]	SDF [71]/pogojna prepisovalna pravila (*rewrite rules*) – ASF [65]	Generira izvorni kod C. ASF je zasnovan na algebrajski semantiki.
LDL [61]	BNF/posebna vrsta atributnih gramatik	Orodje je zapisano v Prologu, zato so specifikacije na moč podobne logičnemu programiranju.
Centaur [60]	BNF (jezik METAL)/naravna semantika (operacijska semantika) ali algebrajska semantika	Zgenerira generično interaktivno okolje (strukturni urejevalnik, interpreter, očiščevalnik itd).
LISA [69]	BNF/atributne gramatike	Razvojno okolje. Zgenerira tudi druga orodja (strukturni urejevalnik, animator procesa evaluacije, grafična predstavitev končnega avtomata, sintaksnega dreva itd.)
LPS [72]	denotacijska semantika	Orodje je implementirano kot vgrajen domensko specifični jezik v programskem jeziku haskell.
ELI [73]	BNF/atributne gramatike	Orodje je v zadnjih letih preraslo naziv *generator prevajalnikov* in se uporablja za generiranje različnih programskih orodij [74].

Tabela 4.2: Orodja za razvoj prevajalnikov

- razvoj novega programskega jezika iz posameznih delov specifikacij drugih programskih jezikov.

Pri tem pa seveda želimo ponovno uporabiti vse lastnosti jezika: leksikalne specifikacije, sintaksne specifikacije, semantične specifikacije in opis domene programskega jezika. V nadaljevanju si natančneje pogledamo ponovno uporabo specifikacij, temelječih na atributnih gramatikah, s katerimi se v disertaciji tudi ukvarjamo.

Ponovna uporabnost specifikacij v AG

Slaba lastnost specifikacij zapisanih v atributnih gramatikah je vsekakor obsežnost in nestrukturiranost. Omenjeni slabosti prispevata tudi k večji neberljivosti specifikacij in težjemu razvoju ter vzdrževanju. Same specifikacije so lahko celo obsežnejše kot sama implementacija. Načrtovanje in implementacija programskih jezikov bi morala biti modularna z možnostjo inkrementalnega spreminjanja programskega jezika. Na tem področju je bilo v preteklosti že precej raziskav, ki pa niso v popolnosti zadovoljile potreb vsakdanje prakse. Nekatere od teh raziskav so:

- Dostop do oddaljenih atributov [81].

  Glavni prispevek je dostop do oddaljenih atributov, ki delno rešuje problem preprostega kopiranja atributov semantičnega drevesa in lokalnost atributnih gramatik.

- Modularne gramatike [82].

  Semantiko programskega jezika specificira avtor z več moduli in predlogami, ki jih preplete v končne specifikacije s pomočjo ujemanja vzorcev s produkcijami začetne kontekstno proste gramatike.

- Objektne atributne gramatike [83, 84, 85].

  Z vpeljavo objektno usmerjenega programiranja v atributne gramatike, v te vključimo nov koncept neterminal = razred. Neterminale gramatike obravnavamo kot razrede. Odvisnosti med neterminali rešujemo z znanimi koncepti, kot je npr. dedovanje, objektno usmerjenega programiranja. Vsak neterminal oz. vozlišče v drevesu izpeljave ustreza pojmu objekta.

- Referenčne atributne gramatike [86].

  Atributi so lahko reference na oddaljena vozlišča.

- Dedovanje gramatik [87].

  Gramatika je opisana v razredu, ki ga lahko dedujemo in specializiramo.

Z možnostjo inkrementalnega in modularnega razvoja programskih jezikov omogočimo ponovno uporabnost specifikacij in pohitrimo sam razvoj programskega jezika. V specifikacije programskih jezikov[3] želimo torej vpeljati čimveč konceptov, ki so se kot zelo uporabni izkazali v splošno namenskih programskih jezikih. V atributnih gramatikah imajo razvijalci in načrtovalci programskih jezikov velike probleme pri dodajanju novih konceptov in njihove integracije v obstoječe specifikacije programskega jezika. Princip dedovanja nudi naravno rešitev danega problema. Specifikacije novega programskega jezika tako temeljijo na že obstoječih specifikacijah. Podedujejo se vse lastnosti jezika, ki se lahko v izpeljanem (dedovanem) jeziku[4] dopolnijo, spremenijo ali uničijo. Na ta način načrtovalec specificira le razlike med jezikoma. S tem omogočimo inkrementalno in modularno načrtovanje programskih jezikov z atributnimi gramatikami. Da bi v popolnosti izkoristili nek koncept, je potrebna formalna podlaga. V nadaljevanju si poglejmo formalni opis večkratnega dedovanja atributnih gramatik [1, 88], ki v atributne gramatike vpelje že znane koncepte objektno usmerjenega programiranja. Znane tehnike nadgradi z večkratnim dedovanjem atributnih gramatik in z uvedbo šablon v atributne gramatike. S tem pristopom je bilo razvito veliko prototipnih domensko specifičnih jezikov, kakor tudi realnih splošno namenskih objektno in aspektno usmerjenih programskih jezikov [89].

Večkratno dedovanje atributnih gramatik

Dedovanje se je v objektno usmerjenih programskih jezikih izkazalo za bistven mehanizem za modularen in inkrementalen razvoj programske opreme. Dedovanje omogoča ponovno uporabo že razvitih komponent, torej definiranje novih komponent, ki temeljijo na že obstoječih, od katerih podedujejo željene lastnosti, ostale pa prepišejo oz. definirajo dodatne lastnosti. Podobne koncepte, so avtorji [88] vpeljali v atributne gramatike. Pristop se zaradi vpeljanega večkratnega dedovanja v atributne gramatike imenuje *"večkratno dedovanje atributnih gramatik"*. Na ta način so lahko specifikacije novega

[3] Specifikacije programskih jezikov predstavljajo domensko specifični jezik. Domena le-tega je razvoj in avtomatsko generiranje programskih jezikov.

[4] Ko v specifikacije programskih jezikov vpeljemo koncepte objektno usmerjenega programiranja, na specifikacije posameznega programskega jezika (komponento) gledamo kot na razred v objektno usmerjenem programiranju. Mi bomo ta razred na kratko imenovali kar *jezik*.

jezika zasnovane na že obstoječem jeziku, od katerega podedujejo posamezne lastnosti (regularne definicije, sintaksa, semantika), ki pa jih lahko v novem jeziku nadgradijo/brišejo ter jim dodajajo nove lastnosti. Formalna metoda, ki so jo vpeljali že omenjeni avtorji, je zelo pomembna, saj omogoča razvijalcem formalen pristop pri ponovni uporabi in inkrementalnemu razvoju programskih jezikov.

V objektno usmerjenih programskih jezikih so posamezne lastnosti razreda, kot so npr. instančne spremenljivke in metode, predmet dedovanja in s tem ponovne uporabe oz. sprememb v izpeljanem (pododedovanem) razredu. V atributnih gramatikah je sklopljenost posameznih lastnosti specifikacij programskega jezika dokaj močna, zato je formalen model dedovanja še toliko bolj pomemben za razvoj novih jezikov. Lastnosti, ki so predmet dedovanja v atributnih gramatikah so: regularne definicije, definicije atributov in produkcijska pravila (kapsulacija BNF sintaksnih pravil in pripadajočih semantičnih funkcij).

Prednosti večkratnega dedovanja atributnih gramatik so naslednje:

- specifikacije so razširljive, saj razvijalec novega jezika specificira le dodatne koncepte novega jezika oz. specializacijo že obstoječih,
- specifikacije so ponovno uporabne (specifikacije novega jezika podedujejo lastnosti že obstoječih jezikov) in
- razvijalec jezika lahko načrtuje in implementira jezik temelječ na več specifikacijah.

Formalno dedovanje zapišemo kot:

$$R = P \oplus \triangle R$$

kjer R označuje novo definirano entiteto, P označuje podedovane lastnosti iz obstoječe entitete, $\triangle R$ označuje inkrementalno dodane nove lastnosti, ki ločujejo entiteto P od entitete R in $\oplus$ označuje operacijo združevanja lastnosti entitet P in $\triangle R$. Rezultat operacije združevanja (R) vsebuje vse lastnosti iz P razen lastnosti, ki so inkrementalno dodane v $\triangle R$ in morebiti niso kompatibilne z osnovno entiteto. Entiteta $\triangle R$ lahko definira nove lastnosti, lahko pa tudi redefinira oz. uniči že obstoječe lastnosti iz P. Dedovanje vnaša tranzitivno relacijo med entitete (objekte, razrede, gramatike, AG itd.). Entiteto P ponavadi imenujemo *starš*. V primeru razreda nadrazred, v našem primeru nadjezik itd. Podobno entiteto R imenujemo otrok oz. izpeljani razred, izpeljani jezik itd. Tip dedovanja, ki smo ga ravnokar opisali, se imenuje enkratno dedovanje, saj lahko izpeljana entiteta podeduje lastnosti zgolj enega

starša. Nasprotje enkratnemu dedovanju je večkratno dedovanje, kjer lahko izpeljana entiteta podeduje lastnosti več staršev hkrati. Formalno večkratno dedovanje zapišemo kot

$$R = P_1 \oplus P_2 \oplus ... \oplus P_n \oplus \triangle R$$

Večkratno dedovanje omogoča večje možnosti za inkrementalno spreminjanje kot enkratno dedovanje. Kljub prednostim večkratnega dedovanja (večja fleksibilnost in izrazna moč) prinaša le–to precej zapletov na konceptualni in tehnični (implementacijski) fazi. Zaradi teh zapletov nekateri moderni objektno usmerjeni programski jeziki (npr. java) večkratnega dedovanja ne podpirajo. Na področju atributnih gramatik so raziskovalci pokazali, da je večkratno dedovanje primernejše od enkratnega, saj omogoča večjo fleksibilnost in izrazno moč specifikacij [1].

Za postavitev formalnega modela večkratnega dedovanja atributnih gramatik je potrebno identificirati lastnosti specifikacij z atributnimi gramatikami, ki so predmet dedovanja. Te lastnosti so: leksikalne regularne definicije, definicije atributov (podedovani, pridobljeni) in sintaksna pravila (kapsulacija sintaksnih (BNF) produkcij in semantičnih funkcij za posamezno produkcijo). Vse entitete, ki so predmet dedovanja, morajo biti primerno poimenovane, saj jih samo na ta način lahko primerno ponovno uporabimo v izpeljanih jezikih. Končne specifikacije atributnih gramatik imajo tako naslednjo obliko:

$$Jezik = RegDefIme + AtributiIme + PraviloIme$$

Za vsak jezik l, na naslednji način definiramo množico nadjezikov $Predniki(l)$, jezika l:

$$Predniki : Jezik \rightarrow \{Jezik\}$$
$$Predniki(l) = \{l_1, l_2, \ldots, l_n\}$$

Brisanje lastnosti jezika

Posamezna lastnost starševskega jezika je lahko v izpeljanih razredih odstranjena in tako nedosegljiva iz vseh izpeljanih jezikov. Množico lastnosti jezika l_2, ki niso dosegljive iz jezika l_1 označimo z $RazveljavljenId(l_1, l_2)$ ter definiramo kot:

$$RazveljavljenId : (Jezik \times Jezik) \rightarrow \{Jezik\}$$
$$RazveljavljenId(l_1, l_2) = \{pr_1, pr_2, \ldots, pr_n\},$$

kjer je pr_i posamezna lastnost jezika.

Regularne definicije

Za vsak jezik l je definirana množica $LeksSpec(l)$, ki definira množico preslikav iz regularnih definicij v regularne izraze jezika l. Kot smo že omenili, regularne definicije predstavljajo poimenovane regularne izraze.

$$LeksSpec : Jezik \rightarrow RegDefIme \rightarrow RegIzraz$$
$$LeksSpec(l) = \{d_1 \mapsto rizr_1, d_2 \mapsto rizr_2, \ldots, d_n \mapsto rizr_n\}$$

Z $E_1, E_2, ..., E_m$ označimo množico preslikav iz regularnih definicij v regularne izraze jezikov $l_1, l_2, \ldots l_m$, formalno definirane kot:

$$\begin{aligned} E_1 &= \{d_{11} \mapsto e_{11}, d_{12} \mapsto e_{12}, \ldots, d_{1k} \mapsto e_{1k}\} \\ E_2 &= \{d_{21} \mapsto e_{21}, d_{22} \mapsto e_{22}, \ldots, d_{2l} \mapsto e_{2l}\} \\ &\vdots \\ E_m &= \{d_{m1} \mapsto e_{m1}, \ldots, d_{mn} \mapsto e_{mn}\}, \end{aligned}$$

kjer je d_{ij} regularna definicija in e_{ij} regularni izraz. Entiteto $E = E_2 \oplus \ldots \oplus E_m \oplus \triangle E_1$, kjer E_1 deduje od $E_2, \ldots, E_m$, formalno definiramo kot:

$$E = E_1 \cup \ldots \cup E_m.$$

Podedovana regularna definicija je torej unija vseh starševskih regularnih definicij in regularnih definicij novo definiranega jezika.

Definicije atributov

Za vsak jezik l definiramo $Attributi(l)$ kot množico preslikav atributov v pripadajoče tipe v jeziku l.

$$Atributi : Jezik \rightarrow AtributiIme \rightarrow Tip$$
$$Atributi(l) = \{a_1 \mapsto tip_1, a_2 \mapsto tip_2, \ldots, a_n \mapsto tip_n\}$$

Ker ima vsak atribut pripadajoči tip, je množica atributov A_i definirana kot:

$$A_i = \{a_{i1} \mapsto tip_{i1}, a_{i2} \mapsto tip_{i2}, \ldots, a_{in} \mapsto tip_{in}\}.$$

Množico podedovanih atributov[5] A formalno zapišemo kot $A = A_1 \ominus \ldots \ominus A_m$. Zaradi dejstva, da ima vsak atribut pridružen tip, ki je lahko različen v različnih jezikih, dedovanja atributov ne moremo definirati kot unijo atributov. Zaradi horizontalnega in vertikalnega prekrivanja smo za združevanje atributov uporabili operator $\ominus$. Horizontalno prekrivanje obravnavamo kot napako, medtem ko vertikalno prekrivanje razrešujemo z asimetričnim nasledniškim iskanjem (*lookup*) [90]. Množica podedovanih atributov $A = A_1 \ominus \ldots \ominus A_m$ je formalno definirana kot:

$$\begin{aligned} A = & A_1 \cup (A_2 \setminus \{a_{1p} \mapsto tip_{1p} | a_{1p} \in fst(A_1)\}) \\ & \cup \ldots \cup (A_m \setminus \{a_{1p} \mapsto tip_{1p} | a_{1p} \in fst(A_1)\}) \wedge \\ & (\neg \exists a_{ji}, j = 2..m, i = 1..n, k \neq l : (a_{ji} \mapsto tip_{jk}) \wedge \\ & (a_{ji} \mapsto tip_{jl}) \wedge (tip_{jk} \neq tip_{jl})). \end{aligned}$$

Sintaksna pravila

Sintaksna pravila kapsulirajo eno ali več sintaksnih (BNF) produkcij in semantične funkcije, ki pripadajo določeni produkciji. Za vsako pravilo r v jeziku l, definira $Pravila(l)(r)$, končno množico parov (p, R_p), kjer je p BNF produkcija in R_p končna množica semantičnih funkcij pripadajočih produkciji p.

$$\begin{aligned} & Pravila : Jezik \rightarrow PraviloIme \rightarrow ProdukcijaSemantika \\ & Pravila(l)(r) = \{(p, R_p) | p \in P, \\ & p : X_0 \rightarrow X_1 X_2 \ldots X_n, \\ & R_p = \{X_i.a = f(X_{0.b}, \ldots, X_{j.c}) | X_i.a \in DefAtr(p)\}\} \end{aligned}$$

Najprej si poglejmo dedovanje konteksno neodvisne gramatike. Naj G_1, G_2, ..., G_m označujejo kontekstno neodvisne gramatike formalno definirane kot:

[5] Termin "podedovanih" se nanaša na dedovanje atributnih gramatik in ne vrsto (podedovani, pridobljeni) atributa.

$G_1 = (N_1, T_1, P_1, Z_1,),$
$G_2 = (N_2, T_2, P_2, Z_2),$
$\vdots$
$G_m = (N_m, T_m, P_m, Z_m)$, potem je gramatika

$G = G_2 \oplus \ldots \oplus G_m \oplus \triangle G_1$, kjer G_1, deduje od $G_2, \ldots, G_m$, definirana na naslednji način:

$G = (N, T, P, Z_1)$, kjer
$N = N_1 \oslash \ldots \oslash N_m,$
$T = T_1 \oslash \ldots \oslash T_m,$
$P = P_1 \odot \ldots \odot P_m.$

Začetni neterminalni simbol končne gramatike G je začetni neterminalni simbol gramatike G_1. Ker lahko inkrementalno dodane produkcije P_1 razveljavijo določene produkcije, z definiranimi terminalnimi in neterminalnimi simboli, končna množica terminalnih simbolov T in neterminalnih simbolov N, ne more biti definirana zgolj kot unija podedovanih terminalnih in neterminalnih simbolov. Med podedovanimi terminalnimi in neterminalnimi simboli definiramo operacijo $\oslash$, ki je definirana na naslednji način:

$$
\begin{aligned}
& V_1 \oslash V_2 \oslash \ldots \oslash V_m = \\
& \quad V_1 \cup (V_2 \setminus \{x | x \in RazveljavljenId(l_1, l_2)\}) \\
& \quad \cup \ldots \cup \\
& \quad (V_m \setminus \{x | x \in RazveljavljenId(l_1, l_m)\}).
\end{aligned}
$$

Iz istega razloga ne moremo končno množico podedovanih produkcij P definirati zgolj kot unijo. Operacija $\odot$ je definirana kot:

$$
\begin{aligned}
& P = P_1 \odot \ldots \odot P_m = P_1 \cup \\
& \quad (P_2 \setminus \{p | p \in fst(Pravila(l_2)(r)) \wedge \\
& \quad r \in RazveljavljenId(l_1, l_2)\}) \cup \ldots \cup \\
& \quad (P_m \setminus \{p | p \in fst(Pravila(l_m)(r)) \wedge \\
& \quad r \in RazveljavljenId(l_1, l_m)\}) \wedge \\
& \quad dom(Pravila(l_i)) \cap dom(Pravila(l_j)) = \emptyset, \\
& \quad i = 2..m, j = 2..m \wedge i \neq j.
\end{aligned}
$$

Preden zapišemo dokončno definicijo dedovanja sintaksnih pravil, je potrebno definirati še operacijo $Zdruzi(R_{pC}, R_{pI})$, ki definira končno množico združenih semantičnih funkcij R_p (združitev semantičnih funkcij R_{pC} in R_{pI}). Če produkcija p obstaja v starševskem in izpeljanem razredu, je potrebno semantične funkcije združiti na način, ki zagotavlja konsistenten zapis atributne gramatike. V nasprotnem primeru semantične funkcije preprosto prepišemo v izpeljani jezik.

$$
\begin{aligned}
&Zdruzi : ProdukcijaSemantika \times ProdukcijaSemantika \rightarrow \\
&\qquad ProdukcijaSemantika \\
&Zdruzi(TrenutnaProd, PodedovanaProd) = \\
&\quad \{(p, R_p) | ((p, R_{pI}) \in PodedovanaProd \wedge \\
&\quad (p, R_{pC}) \in TrenutnaProd \wedge \\
&\quad R_p = zdruzi(R_{pC}, R_{pI})) \vee \\
&\quad ((p, R_p) \in PodedovanaProd \wedge \\
&\quad (p, R_{pC}) \notin TrenutnaProd) \vee \\
&\quad ((p, R_p) \in TrenutnaProd \wedge \\
&\quad (p, R_{pI}) \notin PodedovanaProd))\}
\end{aligned}
$$

Operacija $zdruzi(R_{pC}, R_{pI})$ definira množico semantičnih funkcij pridruženih produkciji p, kjer semantične funkcije za isti atribut redefinirajo podedovanega.

$$
\begin{aligned}
&zdruzi(R_{pC}, R_{pI}) = \\
&\quad \{X_i.a = f(X_{0.b}, \ldots, X_{j.c}) \\
&\qquad | X_i.a \in DefAtr(p_C) \\
&\qquad \vee (X_i.a \in DefAtr(p_I) \wedge \\
&\qquad X_i.a \notin DefAtr(p_C))\}
\end{aligned}
$$

Za funkcijo $f : A \rightarrow B$, definiramo funkcijo $f[a/b]$, ki se obnaša na enak način kot f, le da preslika vrednost $a \in A$ v $b \in B$. Formalno je funkcija definirana kot:

$$
\begin{aligned}
&(f[a/b])(a) = b \\
&(f[a/b])(a_0) = f(a_0); \forall a_0 \in A \wedge a_0 \neq a
\end{aligned}
$$

Sedaj, ko imamo definirano vse potrebno, lahko definiramo še dedovanje sintaksnih pravil. Iz že znanih razlogov (horizontalno in vertikalno prekrivanje in zagotavljanje konsistentnosti atributne gramatike) množica semantičnih pravil $R = R_1 \otimes \ldots \otimes R_m$, ne more biti definirana kot preprosta unija.

$$
\begin{aligned}
R = & R_1 \otimes \ldots \otimes R_m = \\
& R_1 \cup snd(Zdruzi((P_1, R_1), (P_2, R_2 \setminus \\
& \{R_p | R_p \in snd(Pravila(l_2)(r)) \wedge \\
& r \in RazveljavljenId(l_1, l_2)\}))) \\
& \cup \ldots \cup snd(Zdruzi((P_1, R_1), \\
& (P_m, R_m \setminus \{R_p | R_p \in snd(Pravila(l_m)(r)) \wedge \\
& r \in RazveljavljenId(l_1, l_m,)\}))) \\
& \wedge dom(Pravila(l_i)) \cap dom(Pravila(l_j)) = \emptyset, \\
& i = 2..m, j = 2..m, i \neq j.
\end{aligned}
$$

Večkratno dedovanje AG

Sedaj pa si poglejmo še formalni zapis *večkratnega dedovanja atributnih gramatik* v celoti. Naj bo AG_1, AG_2, $\ldots$, AG_m, atributna gramatika formalno definirana kot:

$$
\begin{aligned}
& AG_1 = (G_1, A_1, R_1), \\
& AG_2 = (G_2, A_2, R_2), \\
& \qquad \vdots \\
& AG_m = (G_m, A_m, R_m), \text{ potem}
\end{aligned}
$$

$$
\begin{aligned}
& AG = AG_2 \oplus \ldots \oplus AG_m \oplus \triangle AG_1 \ , \\
& \qquad \text{kjer je } AG_1, \text{ podedovana od } AG_2, \ldots, AG_m, \text{ definirana kot}
\end{aligned}
$$

$$
\begin{aligned}
& AG = (G, A, R), \text{ kjer } G = G_2 \oplus \ldots \oplus G_m \oplus \triangle G_1, \\
& A = A_1 \ominus \ldots \ominus A_m, \\
& R = R_1 \otimes \ldots \otimes R_m \ .
\end{aligned}
$$

Primer

Kot primer dedovanja atributnih gramatik, si poglejmo gramatiko s primera 4.4 (stran 48). Atributno gramatiko, ki definira semantiko dani gramatiki, smo zapisali v poglavju 4.4.1. Osnovno gramatiko razbijemo na dve

komponenti. V prvi komponenti definiramo jezik, ki omogoča seštevanje in odštevanje realnih števil. V izpeljani komponenti smo jezik nadgradili s podporo za množenje in deljenje ter z operatorjem oklepaj.

$$
\begin{aligned}
&AG_{Expr} = (G_{Expr}, A_{Expr}, R_{Expr})\\
&A_{Expr} = \{Z.val \mapsto double, E.val \mapsto double, T.val \mapsto double\}\\
&R_{Expr} = R_{Start} \cup R_{Expr1} \cup R_{Expr2} \cup R_{Expr3} \cup R_{Term}\\
&R_{Start} = R_{Z\to E}\\
&R_{Expr1} = R_{E\to E+T}\\
&R_{Expr2} = R_{E\to E-T}\\
&R_{Expr3} = R_{E\to T}\\
&R_{Term} = R_{T\to \#Number}\\
&R_{Z\to E} = \{Z[0].val = E[0].val\}\\
&R_{E\to E+T} = \{E[0].val = E[1].val + T[0].val\}\\
&R_{E\to E-T} = \{E[0].val = E[1].val - T[0].val\}\\
&R_{E\to T} = \{E[0].val = T[0].val\}\\
&R_{T\to \#Number} = \{T[0].val = \#Number[0].value()\}
\end{aligned}
$$

$$
\begin{aligned}
&AG_{Expr} \oplus \triangle AG_{ExprMul} = (G_{Expr} \oplus \triangle G_{ExprMul},\\
&\qquad A_{ExprMul} \ominus A_{Expr}, R_{ExprMul} \otimes R_{Expr})\\
&A_{ExprMul} \ominus A_{Expr} = \{F.val \mapsto double\} \cup\\
&\qquad \{Z.val \mapsto double, E.val \mapsto double, T.val \mapsto double\}\\
&RazveljaljenId(ExprMul, Expr) = \{Expr.R_{Term}\}\\
&R_{ExprMul} \otimes R_{Expr} = merge(R_{Start}, Expr.R_{Start}) \cup\\
&\qquad R_{Expr1} \cup R_{Expr2} \cup R_{Expr3} \cup\\
&\qquad Expr.R_{Expr1} \cup Expr.R_{Expr2} \cup Expr.R_{Expr3} \cup\\
&\qquad R_{Term} \cup R_{Term1}\\
&R_{Start} = R_{Z\to E}\\
&R_{Expr1} = R_{T\to T*F}\\
&R_{Expr2} = R_{T\to T/F}\\
&R_{Expr3} = R_{T\to F}\\
&R_{Term} = R_{F\to \#Number}\\
&R_{Term} = R_{F\to (E)}\\
&R_{Z\to E} = merge(R_{Start}, Expr.R_{Start}) = \{Z[0].val = E[0].val\}\\
&R_{T\to T*F} = \{T[0].val = T[1].val * F[0].val\}\\
&R_{T\to T/F} = \{T[0].val = T[1].val / F[0].val\}\\
&R_{T\to F} = \{T[0].val = F[0].val\}\\
&Expr.R_{E\to E+T} = \{E[0].val = E[1].val + T[0].val\}\\
&Expr.R_{E\to E-T} = \{E[0].val = E[1].val - T[0].val\}
\end{aligned}
$$

$Expr.R_{E \rightarrow T} = \{E[0].val = T[0].val\}$
$R_{F \rightarrow \#Number} = \{F[0].val = \#Number[0].value()\}$
$R_{F \rightarrow (E)} = \{F[0].val = E[0].val\}$

4.6 Sorodna dela doktorski disertaciji

V zadnjem času se precej raziskovalcev ukvarja z izboljšanjem metod za načrtovanje in implementacijo programskih jezikov. Posebne pozornosti so deležni domensko specifični programski jeziki, ki se v praksi vedno bolj uveljavljajo. Poskušajo se formalizirati postopki za načrtovanje in posledično orodja za implementacijo. Obstaja veliko raziskav za izboljšanje formalnega načrtovanja programskih jezikov in orodij za avtomatsko generiranje prevajalnikov. V večini raziskav se avtorji trudijo izboljšati abstraktni nivo specifikacij ter izboljšati njihovo ponovno uporabnost. Specifikacijski jeziki temeljijo na različnih formalizmih, tako da je direktna primerjava zelo težka. Pregled vseh metod za izboljšanje formalnih metod za načrtovanje in implementacijo programskih jezikov je preobsežna in ni v sklopu disertacije. Na tem mestu se osredotočimo le na raziskave, kjer avtorji uvajajo paradigme aspektno usmerjenega programiranja v specifikacije programskih jezikov. Posebej natančno raziščemo dela, kjer se principi AOP ali podobni uporabljajo v navezavi z atributnimi gramatikami.

4.6.1 Modularne atributne gramatike

V delu [82] so opisane modularne atributne gramatike (MAG). Čeprav še koncepti AOP [28] leta 1990, ko sta avtorja predstavila svoje delo, niso obstajali, lahko MAG štejemo za enega izmed prvih poskusov uvajanja konceptov AOP v atributne gramatike. Modularizacije in posledičnega zmanjšanja kompleksnosti atributnih gramatik sta se avtorja lotila s šablonami in ujemanjem vzorcev. Specifikacije opišemo s pomočjo vzorcev, ki mu lahko pridružimo eno ali več šablon. Vzorci specificirajo produkcije kontekstno proste gramatike, s šablonami pa je definirana semantika programskega jezika. Šablone generirajo le semantične funkcije za tiste produkcije, ki se ujemajo z vzorcem, in kjer je atribut uporabljen v semantični funkciji potreben v drugih izračunih. S šablonami v MAG je mogoče opisati splošne ponovno uporabne semantične funkcije, ki niso vezane na določeno sintakso programskega jezika. Moduli torej abstrahirajo določen semantični koncept. Z moduli je mogoče opisati tudi večino znanih semantičnih vzorcev v atributnih gramatikah. Opis atributnih gramatik z moduli MAG bistveno zmanjša število vrstic

zapisa atributnih gramatik. Slabosti pristopa so jasno vidne pri implementaciji realnejših programskih jezikov, kjer postane zapis in branje specifikacij precej težaven, predvsem zaradi prilagajanja vzorcev. Uporabnik zelo težko sledi prilagajanju vzorcev in težko predvidi izračun atributov.

Predlagani pristop je bistven prispevek na področju modularizacije atributnih gramatik. Delo so povzeli in nadgradili številni raziskovalci. V disertaciji se problema lotimo na podoben način, s to razliko, da smo več poudarka namenili končnemu uporabniku in lažjemu razumevanju zapisanih specifikacij. Z MAG je uspešno rešen problem obsežnosti specifikacij AG, a nivo modularnosti, razumljivosti in ponovne uporabnosti še ni zadovoljiv. V disertaciji ponujamo boljše rešitve za omenjene probleme. Modularnost smo izboljšali s koncepti AOP, kjer smo parametrizirali aplikacijo nasvetov in tako zagotovili zapis ponovno uporabnih neodvisnih modulov za posamezen koncept programskega jezika. Razumljivost specifikacij smo zagotovili z natančno sintakso in jasnimi pravili za ujemanje vzorcev in uporabo nasvetov. Prav tako je sam razvoj podprt z orodjem, ki razvijalcu pomaga v vseh fazah razvoja programskega jezika.

4.6.2 Specifikacijski jezik APS

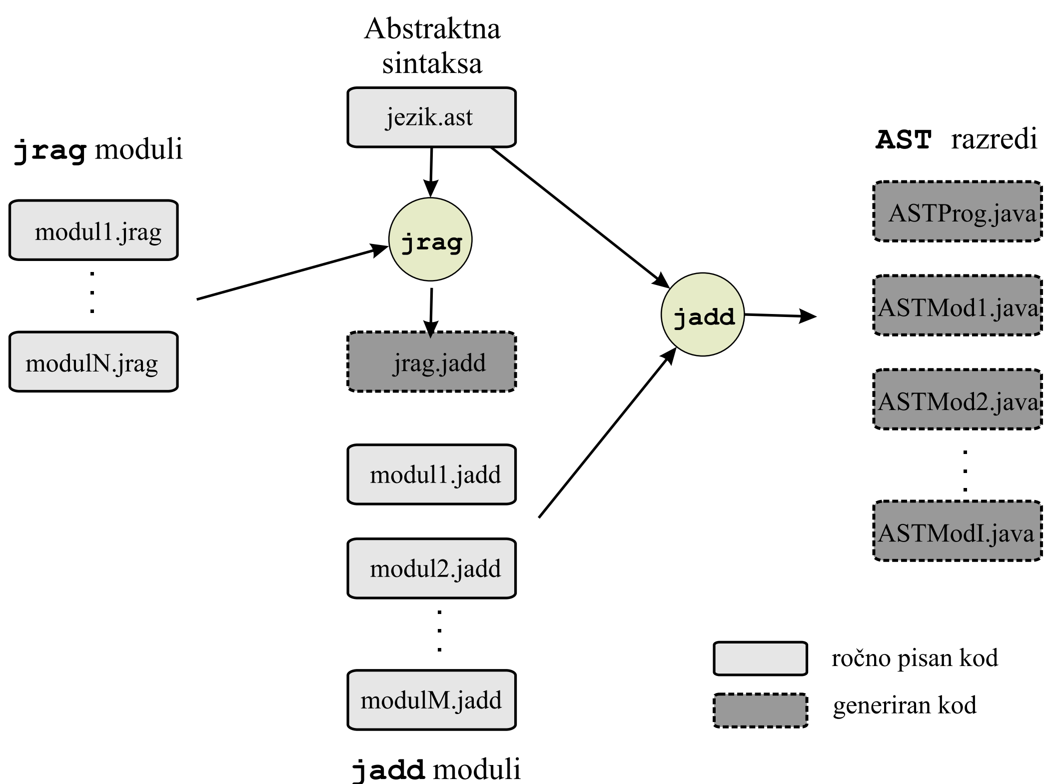
V doktorski disertaciji [91] se avtor ukvarja s problemom dekompozicije atributnih gramatik na nivoju specifikacijskega jezika. Predvsem se ukvarja s problemom podvajanja enakih specifikacij (semantičnih funkcij) v več produkcijah kontekstno proste gramatike. Avtor je razvil specifikacijski jezik APS, ki podpira zapis posameznih konceptov programskega jezika v ločene module. Dekompozicija je podprta z mehanizni, kot so: ujemanje vzorcev, logične sekvence, oddaljeni atributi in modularizacija. Najmočnejši mehanizem jezika ASP je ujemanje vzorcev, s katerim avtor rešuje problem podvajanja semantičnih funkcij. S predlagano rešitvijo avtor bistveno zmanjša specifikacije jezika ter izboljša njihovo modularnost. Na semantične specifikacije, opisane za posamezen vzorec, lahko gledamo kot na aspekt.

4.6.3 JastAdd

Avtorji orodja JastAdd [68] so se izboljšanja modularnosti in ponovne uporabe specifikacij programskih jezikov lotili z uporabo mehanizma ReRAGs (***Re****writable Circular* ***R****eference* ***A****ttributed* ***G****rammars*). Specifikacije jezika so predstavljene z objektno usmerjeno predstavitvijo abstraktnega sintaksnega drevesa (AST). Neterminalni simboli so predstavljeni kot abstraktni nadrazredi, medtem ko so produkcije, atributi in semantična pravila predstavljeni s konkretno implementacijo (izpeljavo) abstraktnih nadrazredov.

Takšna predstavitev omogoča ponovno uporabo z mehanizmom dedovanja objektno usmerjenih programskih jezikov. Mehanizem ReRaGs vključuje več mehanizmov, kot so: objektno usmerjeno programiranje, aspektno usmerjeno programiranje, deklarativno programiranje, atributne gramatike in transformacijske sisteme. Povečanje modularnosti zapisa programskega jezika se avtorji lotevajo z zapisom posameznih lastnosti jezika v ločene module. Ti so lahko razviti z različnimi pristopi (specifikacije abstraktne sintakse, deklarativne specifikacije – moduli `jrag`, imperativne specifikacije – moduli `jadd` zapisani v programskem jeziku java), ki se v fazi generiranja združijo v celoto. Arhitektura sistema je prikazana na sliki 4.5. Sintakso programskega jezika v orodju definiramo v modulih `ast`, iz katerih se zgenerira ustrezna hierarhija razredov v objektno usmerjenem programskem jeziku. Semantiko programskega jezika lahko v orodju JastAdd definiramo na dva načina. Z atributno gramatiko in s posebnim specifikacijskim jezikom, ki se definira v modulih `jrag`. Drugi način je imperativen in se definira v modulih `jadd`, kjer semantiko definiramo v programskem jeziku java. Posamezno dolžnost (aspekt) programskega jezika je mogoče zapisati v ločenem modulu. V fazi generiranja se najprej posamezni moduli `jadd` združijo (aspektno tkanje) v objektno usmerjeno konkretno implementacijo abstraktnega sintaksnega drevesa, ki predstavlja implementacijo programskega jezika. Aspektno usmerjeno specificiranje programskega jezika je podprto torej na nivoju `jadd` modulov. Pristop ne sledi klasičnemu modelu AOP, ki ga predlagajo avtorji v [37, 28]. JastAdd uporablja implicitni model stićišč in specifikatorjev stičišč, ki jih razvijalec ne more definirati. Pristop bi lahko primerjali z medtipskimi deklaracijami v programskem jeziku aspectJ, kjer so stične točke vse poimenovane lastnosti določene komponente. Arhitektura orodja je zasnovana tako, da se je mogoče v `jadd` modulu sklicevati na lastnosti v drugih `jadd` in `jrag` modulih. Pravzaprav je to za resnejši programski jezik nujno. Ravno ta lastnost se izkaže za precejšnjo slabost, saj so moduli močno sklopljeni, kar ne omogoča zadovoljivega nivoja ponovne uporabnosti. Prav tako ni mogoče razviti splošnih modulov za določene dolžnosti programskega jezika, saj moduli ne omogočajo parametrizacije, kar jih veže na konkretno implementacijo programskega jezika.

Predlagan pristop ne podpira konceptov AOP na nivoju specifikacij programskega jezika, temveč na nivoju orodja. V disertaciji se problematike lotimo celoviteje. Koncepti AOP so podprti na nivoju specifikacij, ki temeljijo na atributnih gramatikah. V ta namen smo definicijo atributnih gramatik razširili s koncepti AOP. Pristop, ki ga predlagamo v disertaciji, omogoča razvoj rahlo sklopljenih, parametriziranih in ponovno uporabnih komponent. Aplikacijo teh pa lahko definira razvijalec z definicijo ustreznih stičišč in uporabo nasvetov.



Slika 4.5: Arhitektura orodja JastAdd

4.6.4 Montages

V delu [92] avtorji predstavijo specifikacije programskih jezikov z moduli, ki jih imenujejo `Montage`. Ti moduli tvorijo ogrodje za specifikacijo imperativnih programskih jezikov, kjer vsak modul v celoti predstavlja nek koncept programskega jezika. Sintaksa jezika, statična in dinamična semantika so v vsakem modulu predstavljeni v pol-grafičnem načinu s kontrolnimi diagrami in grafi toka podatkov, ki predstavljajo abstraktni stroj stanj (ASM) in se zapišejo za vsako produkcijo gramatike. Struktura modulov je enaka hierarhiji produkcij zapisanih v notaciji EBNF, kjer je vsak modul semantična razširitev BNF produkcije. Čas učenja za takšno predstavitev sintakse in semantike programskih jezikov je dokaj kratek. Orodje `Gem-Mex` [92], ki podpira razvoj jezikov z moduli `Montage`, in predstavitev programskega jezika sta primerna za hiter razvoj prototipnih jezikov. Orodje iz podanih specifikacij zgenerira interpreter, grafični razhroščevalnik in druga orodja.

V doktorski disertaciji [93] avtor predstavi *Montage Component System* (MCS), ki je nadgradnja Montage in za specifikacijski jezik uporablja programski jezik java. MCS omogoča ločen razvoj modulov programskega jezika za različen nivo implementacije programskega jezika (razpoznavanje, statična semantika, generiranje koda in izvajanje (dinamična semantika)). Kombiniranje modulov vseh nivojev in njihova ponovna uporaba je eden pomembnejših prispevkov disertacije. Ponovna uporaba je zagotovljena na nivoju modulov, tudi v binarni obliki, ki jih lahko uporabimo pri specifikacijah drugih programskih jezikov.

Avorji [92, 93] se modularnosti specifikacij lotevajo na nivoju produkcij gramatike programskega jezika, kar je že znan koncept. Predlagana ponovna uporaba modulov v tej disertaciji je zastavljena širše, saj je mogoče pri razširjanju in ponovni uporabi posameznih modulov uporabiti prednosti aspektno-usmerjenega programiranja. Modulov ni mogoče le ponovno uporabiti, ampak jih lahko ob ponovni uporabi razširimo z uporabo dedovanja, aspektno-usmerjenega programiranja ali kombinacijo obeh tehnik.

The Answer to the Great Question...Of Life, the Universe and Everything...Is...Forty-two.

– The computer Deep Thought –

Poglavje 5

Aspektno usmerjene atributne gramatike

Atributne gramatike, ki smo jih spoznali že v prejšnjih poglavjih, zraven mnogo dobrih lastnosti, prinašajo tudi nekatere slabosti. Le–te so razlog, da atributne gramatike niso uporabljene v širšem obsegu pri specifikacijah programskih jezikov in tudi na drugih področjih. Z izboljšanjem nekaterih slabosti se ukvarjamo v tej disertaciji. V poglavju uvajamo aspektno usmerjene atributne gramatike, ki so povsem nov pristop in predstavljajo enega izmed izvirnih znanstvenih prispevkov disertacije.

5.1 Uvod in motivacija

Atributne gramatike je leta 1968 predstavil D. E. Knuth [48]. V tem času so se atributne gramatike izkazale za zelo uporabne pri specificiranju semantike programskih jezikov, kakor tudi pri implementaciji orodij za avtomatsko generiranje prevajalnikov/interpreterjev za programske jezike. Čeprav je implementacija programskih jezikov izvirno področje atributnih gramatik in tudi

področje, kjer so atributne gramatike najpogosteje uporabljene, jih srečamo na mnogih drugih področjih [94]: vmesniki naravnih jezikov, grafični uporabniški vmesniki, programsko inženirstvo, komunikacijski protokoli, vizualno programiranje, statična analiza programov, podatkovne baze itd. Kljub širokemu spektru uporabnosti najdemo zgolj nekaj komercialnih prevajalnikov, ki so načrtovani in implementirani z atributnimi gramatikami. Avtor v delu [95] ugotavlja, da atributne gramatike niso primerne za izdelavo hitrih komercialnih prevajalnikov za splošno namenske programske jezike. Kot razlog za to ugotovitev navaja preprostost njihovega modela prevajanja ter nezmožnost direktne podpore generiranju in optimiziranju strojnega koda. Vse te ugotovitve ciljajo na pragmatičen aspekt atributnih gramatik. V atributnih gramatikah (vsaj v svoji osnovni obliki, kot jih je predlagal D.E. Knuth) najdemo precej slabosti, ki so opazne pri specifikaciji realnih (velikih) splošno namenskih programskih jezikov. Specifikacije so v teh primerih dokaj velike, slabo strukturirane in težko berljive. Še slabše se atributne gramatike odrežejo pri nadgradnji specifikacij programskega jezika. V tem primeru lahko majhne spremembe v programskem jeziku povzročijo spremembo skoraj celotnih specifikacij. V preteklosti je bilo opravljenih precej raziskav, ki poskušajo odpraviti slabosti atributnih gramatik, kot so pomanjkanje modularnosti, razširljivosti in ponovne uporabnosti. Posamezni koncepti, o katerih smo v delu že govorili, kot so: dostop do oddaljenih atributov, objektna usmerjenost, šablone v atributnih gramatikah [54, 96, 97, 98] itd., izboljšujejo nekatere slabosti, a ne zadovoljivo.

Te slabosti poskušamo v delu izboljšati z uvedbo konceptov AOP [28] v atributne gramatike. Aspektno usmerjeno programiranje se je v zadnjih letih izkazalo za obetajoč pristop pri opisovanju prekrivanja dolžnosti in prepletenega koda na modularen, ponovno uporaben način. Te lastnosti smo integrirali v atributne gramatike, pristop pa poimenovali aspektno usmerjene atributne gramatike (AOAG).

5.2 Aspektno usmerjene atributne gramatike

Aspektno usmerjeno programiranje se ukvarja z modularizacijo prekrivanja dolžnosti in prepletenega programskega koda. Prekrivanje dolžnosti najdemo v različnih programskih strukturah in v različnih fazah življenskega cikla (npr. izvorni kod, modeli, zahteve, gramatike). Atributne gramatike pri tem niso izjema. Podrobnejši pregled specifikacij atributnih gramatik nakazuje nezmožnost atributnih gramatik za modularen opis posameznih konceptov programskih jezikov, kakor tudi nezmožnost za modularno nadgrajevanje specifikacij, saj se specifikacije posameznih konceptov prekrivajo z ostalimi.

Posamezne razširitve programskih jezikov (preverjanje tipov, okolje, generiranje koda) zahtevajo ob uvedbi spremembe semantičnih funkcij v večini produkcij atributne gramatike. Kot smo že omenili, se atributne gramatike uporabljajo tudi za avtomatsko generiranje prevajalnikov/interpreterjev ter drugih orodij (npr. urejevalniki, razhroščevalniki itd.) [99]. Pogosto je praksa, da je potrebno del specifikacij za ta orodja zapisati z atributnimi gramatikami. Specifikacije za posamezna orodja so ponavadi razdrobljene čez celotne specifikacije programskega jezika in s tem posegajo v modularno strukturo specifikacij. To so le nekateri primeri, ki so nas opomnili na resnost problema. Aspektno programiranje se v tem primeru ponuja kot idealna rešitev in smo jo poskušali za opisane probleme tudi uporabiti.

V disertaciji se ukvarjamo zgolj s prekrivanjem dolžnosti pri specifikacijah semantike programskih jezikov. S sintakso programskih jezikov, ki jo z atributnimi gramatikami definiramo v notaciji BNF, se v disertaciji ne ukvarjamo. Menimo, da so modeli za razširitve/modularnost sintakse programskih jezikov z atributnimi gramatikami že uspešno rešene s pristopi kot so: večkratno dedovanje, šablone v atributnih gramatikah [88] in ostalimi [100].

Preden se lotimo opisa formalnega modela aspektno usmerjenih atributnih gramatik, si poglejmo še model stičišč, ki je pomemben del vseh aspektnih jezikov. Glede na to da se specifikacije atributnih gramatik ne izvajajo in so deklarativne, ter da se z aspektnim pristopom lotevamo zgolj semantike jezika, smo stičišča definirali kot statične točke v specifikacijah atributnih gramatik, kjer lahko dodamo semantične konstrukte. Semantika je v atributnih gramatikah specificirana v sklopu sintaksne produkcije. Stičišča v aspektno usmerjenih atributnih gramatikah so torej produkcije. Semantične koncepte, ki se prepletajo z ostalimi, definiramo v nasvetih, definirano semantiko v le–teh pa naknadno pripnemo "originalni" semantiki v sklopu željene produkcije. To definiramo s specifikatorjem stičišč.

Aspektno usmerjena atributna gramatika (AOAG) je atributna gramatika razširjena z definicijo stičišč, definicijo nasvetov in definicijo aplikacije nasvetov. Aspektno usmerjena atributna gramatika je torej definirana kot: $AOAG = (G, A, R, Pc, Ad, Aa)$.

Množica definicij stičišč Pc definira končno množico specifikatorjev stičišč, $Pc = \{pc_1, ..., pc_m\}$, kjer je specifikator stičišč $pc_i, 1 \leq i \leq m$, uporabljen za definicijo ujemanja množice produkcij. Definicija specifikatorja stičišč ima obliko: $pName : LeftS \rightarrow RightS$, kjer je $pName$ unikatni identifikator definicije, $LeftS$ pravilo (vzorec) ujemanja za neterminalni simbol na levi strani produkcije in $RightS$ pravilo ujemanja za desno stran (terminalni in neterminalni simboli) produkcije. Za definicijo pravila ujemanja

produkcije smo uvedli dva nadomestna znaka (*wildcard symbols*) '..' in '$*$', ki ju lahko uporabimo pri definiranju vzorca za ujemanje $LeftS$ in $RightS$ v specifikatorju stičišč pc_i. Nadomestni znak '..' se ujema z nič ali več simboli $\alpha \in V$ in ga lahko uporabimo pri definiciji vzorca za ujemanje $RightS$. Z nadomestnim znakom '$*$' definiramo vzorec za ujemanje celotnega imena oz. dela imena terminalnega in neterminalnega simbola. Ta nadomestni znak lahko uporabimo za definicijo vzorca ujemanja $LeftS$ in $RightS$. Specifikator stičišč ($pName\ :\ LeftS \rightarrow\ RightS$) definira ujemanje produkcije $p \in P, p : X_0 \rightarrow X_1 ... X_n (n \geq 0)$, če se X_0 ujema z vzorcem $LeftS$,in če se $X_1\ ...\ X_n$ ujema z vzorcem $RightS$. S Pm_i označimo množico produkcij definiranih s pc_i, $Pm_i = \{p_i \mid p_i \in P \wedge matched(p_i, pc_i)\}$. Ujemajoče produkcije Pm (množica stičišč) izbrane s specifikatorji stičišč Pc so tako definirane kot $Pm = \bigcup_{i=1...m} Pm_i$ in velja $Pm \subseteq P$.

Ad definira množico definicij nasvetov $Ad = \{ad_1, ..., ad_l\}$, kjer je nasvet ad_k, $1 \leq k \leq l$ naslednje oblike: $aName\ < F_1, ..., F_r >\ \{\ Rs_v\ \}$. $aName$ predstavlja unikatni identifikator nasveta, medtem ko so simboli $F_r \in V \cup A, r \geq 0$, formalni parametri semantičnih funkcij Rs_v, ki so definirane v nasvetu ad_k. $Rs_v, v \geq 0$ je množica semantičnih funkcij naslednje oblike: $Rs_v = \{X_j.a = f(y_1, ..., y_n) \mid a \in A(X_j), y_i \in A(X_0) \cup ... \cup A(X_n),\ 0 \leq i \leq n\}$, kjer so $X_0, ..., X_n$ neterminalni simboli produkcije, ki se ujemajo na vzorec ujemanja definiran s specifikatorjem stičišč. V telesu nasveta lahko definiramo tudi abstrakcije semantičnih funkcij, ki so neodvisne od strukture produkcij (definirane sintakse). Te abstrakcije lahko uporabimo za definicijo enostavnih semantičnih funkcij, kakor tudi za definicijo vzorcev (distribucija seznama, distribucija vrednosti, konstrukcija seznama itd.), ki se pogosto pojavljajo pri specifikaciji programskih jezikov z atributnimi gramatikami. V ta namen smo definirali dva psevdo-identifikatorja LHS in RHS, ki označujeta neterminalni simbol na levi strani produkcije in seznam neterminalnih simbolov na desni strani produkcije.

Definicijo aplikacij nasvetov definira množica $Aa = \{aa_1, ..., aa_t\}$. Posamezna definicija aplikacije nasveta $aa_u, 1 \leq u \leq t$ ima naslednjo obliko: *apply* $aName < S_1, ..., S_q >$ *on* $pName$, kjer sta *apply* in *on* rezervirani besedi, $aName$ je ime obstoječe definicije nasveta, $pName$ je ime obstoječe definicije specifikatorja stičišč in $S_q \in V \cup A, q \geq 0$ je množica dejanskih parametrov, ki se zamenjajo s formalnimi parametri nasveta Ad v času tkanja.

Tkanje

Aspektno tkanje je proces združevanja funkcionalnosti definirane v osnovnih (komponentnih) modulih s funkcionalnostjo zapisano z aspekti. Na ta način dobimo končno funkcionalnost sistema. Proces tkanja v aspektno usmerjenih atributnih gramatikah se bistveno ne razlikuje od tkanja v drugih aspektno usmerjenih jezikih. V aspektno usmerjenih atributnih gramatikah se v procesu tkanja iz atributnih gramatik in iz definicije dodatnih semantičnih funkcij v aspektnem delu generirajo monolitne atributne gramatike. V literaturi poznamo kar nekaj različnih načinov tkanja. Za aspektno usmerjene atributne gramatike predlagamo statično tkanje, kar pomeni, da se končna monolitna oblika atributnih gramatik generira iz aspektnih specifikacij, pred nadaljno obdelavo atributnih gramatik (npr. s strani generatorja prevajalnikov, ki iz atributnih gramatik avtomatsko zgenerira prevajalnik za podan jezik). Proces tkanja v aspektno usmerjenih atributnih gramatikah je definiran na naslednji način.
V nasvetu Ad definirani atributi, pridruženi simbolom $X_j, 0 \leq j \leq n$, so definirani s semantičnimi funkcijami v Rs_v. Semantika definirana v nasvetu ad_k ($aName$) se pripne (pridruži) semantiki množice produkcij Pm_i, ki ustreza vzorcu definiranim s specifikatorjem stičišč pc_i ($pName$). Za vsako produkcijo $p_i \in Pm_i$, ki se ujema z vzorcem specifikatorja stičišč, dobimo dejansko množico semantičnih funkcij Ra_{ki} z zamenjavo formalnih parametrov R_j (definirani v ad_k) z dejanskimi parametri S_j (definirani v aa_u) v Rs_v (število dejanskih parametrov aplikacije nasveta aa_u in formalnih parametrov nasveta ad_k mora biti enako; $q = r$). Množico semantičnih pravil Ra definiranih s semantiko nasvetov Ad in produkcij, ki se ujemajo na vzorec specifikatorja stičišč Pc, je definirana kot $Ra = \bigcup_{k=1..l,i=1..m} Ra_{ki}$ in jo je potrebno pridružiti originalnim semantičnim funkcijam Rp_i. Tako pridobimo pravilno definirano atributno gramatiko $AG = (G, A, R')$ na naslednji način:

$$Rp'_i = Rp_i \cup (\bigcup_{k=1..l} Ra_{ki}), R' = \bigcup_{i=1..z} Rp'_i((G, A, R, Pc, Ad, Aa) = (G, A, R'))$$

Algoritem tkanja, kot smo ga opisali, je predstavljen s psevdokodom v algoritmu 5.1. Vhod v algoritem so specifikacije aspektno usmerjenih atributnih gramatik ter seznam definicij aplikacij nasvetov, ki definirajo uporabo nasveta na neki množici stičišč. Čeprav so ti stavki del specifikacij posameznih modulov celotnih specifikacij, jih v algoritem podajamo ločeno, saj so neodvisni od hierarhije komponent. Za vsako definicijo nasveta se potem izvrši identičen algoritem:

- iskanje (*lookup*) nasveta in specifikatorja stičišč (definirana v definicij nasveta),

- iskanje množice produkcij, ki se ujemajo na vzorce iz specifikatorja stičišč in
- uporaba (aplikacija) semantike nasveta na vse produkcije (zamenjava formalnih in dejanskih parametrov ter dodajanje novih semantičnih funkcij).

Algoritem 5.1 Algoritem tkanja AOAG

```
method tkanje(apply, jezik)
  Vhod: apply // množica "apply" stavkov
        jezik // končni jezik – najnižje v hierarhiji jezikov
  Izhod: monolitna atributna gramatik

  for u ← 1 to t do
    // najdi ujemanje parov (nasvet, specifikator stičišč)
    nasvet_specifikator_stičišč ← najdi_par_ujemanja(aa_u)
    ad ← nasvet_specifikator_stičišč.vrni_advice()
    pc ← nasvet_specifikator_stičišč.vrni_pointcut())
    // najdi seznam produkcij, ki se ujemajo na vzorec
    P ← ujemanje(pc)
    for x ← 1 to P.velikost() do
      // zamenjava formalnih parametrov nasveta in dejanskih parame-
      trov
      // aplikacija semantičnih funkcij na produkcije,
      // ki jih dobimo s P.vrni(x)
      tkanje(ad, P.vrni(x))
    end for
  end for
end method
```

5.2.1 Primer

Nekatere prednosti aspektno usmerjenih gramatik so predstavljene na primeru, ki ga opišemo v tem poglavju. Iz primera je razvidno, da je mogoče z aspektnimi lastnostmi atributnih gramatik enostavneje in na bolj modularen način načrtovati razširitve specifikacij programskih jezikov. Predvsem se na ta način izognemu podvajanju istih semantičnih funkcij v večih produkcijah. Podvajanje istih semantičnih funkcij je monotono opravilo, ki je pogost vir napak, saj je potrebno ob spremembi semantike isto semantično funkcijo

Primer 5.1 Prikaz konceptov aspektno usmerjenih atributnih gramatik

```
Specifikacije jezika v atributnih gramatikah:
p0:  A → B C {A.x = B.x + C.x; B.y = 0; C.y = 1;}   // Rp0
p1:  B → a B {B0.x = B1.x; B1.y = B0.y + 1;}        // Rp1
p2:  B → b B {B0.x = B1.x + 1; B1.y = B0.y;}        // Rp2
p3:  B → ϵ {B.x = B.y;}                             // Rp3
p4:  C → c {C.x = C.y + 2;}                         // Rp4
p5:  C → d {C.x = C.y + 2;}                         // Rp5
Specifikatorji stičišč:
pc1 :  A → B C                                      // ujemanje p0
pc2 :  B → * B                                      // ujemanje p1 in p2
pc3 :  B → ϵ                                        // ujemanje p3
pc4 :  C → ..                                       // ujemanje p4 in p5
pc5 :  * → ..                                       // ujemanje vseh produkcij
Nasveti:
ad1 <X, Y, Z, val> {X.val = Y.val + Z.val;}
ad2 <X, val> { X0.val = X1.val + 1;}
ad3 <X, val, value> {X0.val = value;}
ad4 <value> { RHS.value = LHS.value;}               // vzorec distribucija vrednosti
Uporaba nasvetov:
apply ad1 <A, B, C, cost> on pc1                    // Ra10 = {A.cost = B.cost + C.cost;}
apply ad2 <B, cost> on pc2                          // Ra21 = {B0.cost = B1.cost + 1;}
                                                    // Ra22 = {B0.cost = B1.cost + 1;}
apply ad3 <B, cost, 0> on pc3                       // Ra33 = {B.cost = 0;}
apply ad3 <C, cost, 2> on pc4                       // Ra34 = {C.cost = 2;}
                                                    // Ra35 = {C.cost = 2;}
apply ad3 <A, val, 0> on pc1                        // Ra30 = {A.val = 0;}
apply ad4 <val> on pc5                              // Ra40 = {B.val=A.val; C.val=A.val;}
                                                    // Ra41 = {B1.val = B0.val;}
                                                    // Ra42 = {B1.val = B0.val;}

Končne semantične funkcije (po tkanju):
Rp'0  =  Rp0 ∪ Ra10 ∪ Ra30 ∪ Ra40   =  {A.x = B.x + C.x; B.y = 0; C.y = 1;
                                        A.cost = B.cost + C.cost;
                                        A.val = 0; B.val = A.val; C.val = A.val;}
Rp'1  =  Rp1 ∪ Ra21 ∪ Ra41          =  {B0.x = B1.x; B1.y = B0.y + 1; B0.cost =
                                        B1.cost + 1; B1.val = B0.val;}
Rp'2  =  Rp2 ∪ Ra22 ∪ Ra42          =  {B0.x = B1.x + 1; B1.y = B0.y; B0.cost =
                                        B1.cost + 1; B1.val = B0.val; }
Rp'3  =  Rp3 ∪ Ra33                 =  {B.x = B.y; B.cost = 0;}
Rp'4  =  Rp4 ∪ Ra34                 =  {C.x = C.y + 2; C.cost = 2;}
Rp'5  =  Rp5 ∪ Ra35                 =  {C.x = C.y + 2; C.cost = 2;}
```

spremeniti v celotnih specifikacijah programskega jezika. Dodatno prednost aspektno usmerjenih atributnih gramatik prinaša možnost zapisa generičnih specifikacij v nasvetih. Te specifikacije so neodvisne od produkcij, katerim dodamo semantiko. Na ta način še dodatno izboljšamo ponovno uporabnost aspektnega dela specifikacij. Prednosti aspektno usmerjenih atributnih gramatik prikazujemo na dokaj majhnem primeru 5.1, ki prikazuje zgolj koncepte. Prednosti AOAG so očitnejše na večjih, realnejših primerih, kar je ponazorjeno z analizo v poglavju 7.

Prvi del specifikacij na primeru 5.1 je zapisan v osnovnih atributnih gramatikah. Semantika posamezne produkcije je zapisana s semantičnimi funkcijami, ki definirajo atributa `x` in `y`. Specifikatorji stičišč ($\mathtt{pc}_1$, ..., $\mathtt{pc}_5$) defi-

nirajo vzorce, ki se ujemajo na produkcije, ki jim lahko dodamo semantiko definirano v nasvetih. Definicija semantike v nasvetih je prikazana v naslednjem delu specifikacij ($\texttt{ad}_1$, ..., $\texttt{ad}_4$). Semantične funkcije nasveta ad_4 predstavljajo generične abstrakcije, ki jih preslikamo v konkretne semantične funkcije v času tkanja (apliciranje semantike nasveta na produkcijo). Nasvet ad_4 predstavlja vzorec "distribucija vrednosti"[1]. Konkretna aplikacija semantičnih funkcij v nasvetih je prikazana v delu aspektnih specifikacij, kjer definiramo, kateri nasvet želimo aplicirati na določen specifikator stičišč (*apply* stavki). V zadnjem delu primera je prikazana celotna atributna gramatika, ki jo dobimo po tkanju. Kot je razvidno iz primera, se semantika nasveta ad_3, kljub temu da nasvet ne definira generičnih semantičnih funkcij, aplicira na tri različne produkcije (produkcije predstavljajo različno sintakso). To je možno zaradi podpore parametrizaciji nasvetov in "apply" dela specifikacij. V nasvetih definiramo formalne parametre (neterminalni in terminalni simboli ter atributi), ki se v času tkanja zamenjajo z dejanskimi, ki jih definiramo v "apply" stavku.

Sliki 5.1 in 5.2 prikazujeta semantično drevo stavka *abad* za originalne specifikacije in za specifikacije po apliciranju aspektnih specifikacij na osnovno gramatiko. Že na preprostem primeru je razvidno, da lahko z aspektno usmerjenim pristopom na dokaj preprost, pregleden in modularen način dodajamo posamezne semantične koncepte programskim jezikom.

Algoritem tkanja (algoritem 5.1) poskusimo razložiti na primeru definicije aplikacije nasveta $\texttt{apply ad}_3 <\texttt{C, cost, 2}> \texttt{ on pc}_4$ (identičen algoritem se izvede tudi za ostale definicije aplikacije nasveta).

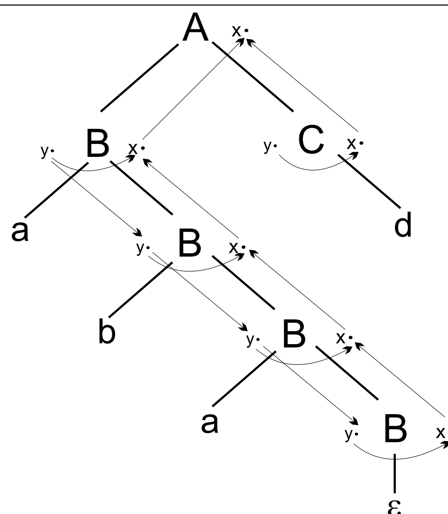
1. Iskanje nasveta in specifikatorja stičišč

   Pri iskanju specifikatorja stičišč je postopek enostaven, saj je mora biti ime specifikatorja stičišč unikatno. Pri nasvetih je potrebno pri iskanju upoštevati podpis nasveta, ki ga sestavljata ime in število formalnih parametrov nasveta. V našem primeru smo najdli nasvet ad_3 in specifikator stičišč pc_4.

2. Iskanje ujemanja vzorca

   V tem koraku se izvaja iskanje ujemanja vzorca, ki ga definira specifikator stičišč, in produkcij. V našem primeru se z vzorcem $pc_4 : C \rightarrow ..$

[1]Podedovane atribute levega neterminalnega simbola prenesemo na vse podedovane atribute neterminalnih simbolov na desni strani produkcije.



Slika 5.1: Drevo ovrednotenja osnovne gramatike za stavek *abad*.

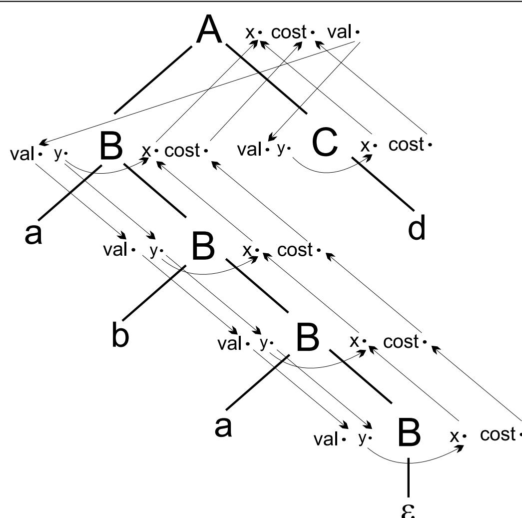
ujemajo vse produkcije, ki imajo na na levi neterminalni simbol C. Desna stran produkcije ni pomembna, saj je '..' splošen vzorec ujemanja. S specifikatorjem stičišč se torej ujemata produkciji p_4 in p_5.

3. Uporaba nasveta (generiranje ustreznih semantičnih funkcij)

   Ta korak je od vseh najzahtevnejši in je sestavljen iz več faz. V prvi fazi je potrebno dejanske parametre naveta, definirane v definiciji aplikacije nasveta, zamenjati z formalnimi. Le–ti se kasneje uporabljajo pri generiranju semantičnih funkcij. V našem primeru se neterminalni simbol `X` nadomesti z neteminalnim simbolom `C`, terminalna simbola `val` in `value` pa se zamenjata z neterminalnima simboloma `cost` in `2`. V drugi fazi se obstoječe semantične funkcije razširijo z dejanskimi parametri (vrši se zamenjava). V kolikor je semantična funckija zapisana generično, se zraven zamenjava izvrši tudi algoritem za razširjanje generičnega zapisa semantične funkcije. V našem primeru se produkcijama p_4 in p_5 doda semantična funkcija `C.cost = 2`.

5.3 Večkratno dedovanje aspektno usmerjenih atributnih gramatik

Prednosti dedovanja kot osnovnega mehanizma ponovne uporabe in inkrementalnega razvoja programske opreme smo spoznali že v poglavju 2.3.1,



Slika 5.2: Drevo ovrednotenja po tkanju za stavek *abad*.

kjer smo se ukvarjali z objektno usmerjenim programiranjem. Predstavljene mehanizme (večkratnega) dedovanja so avtorji [88] uspešno uporabili tudi na področju atributnih gramatik (poglavje 4.5.2).

Aspektno usmerjene atributne gramatike so naravna razširitev atributnih gramatik z definicijo stičišč, definicijo nasvetov in definicijo uporabe nasvetov. Te lastnosti smo vključili v sprecifikacije atributnih gramatik in so kot take tudi predmet dedovanja. Specifikacije aspektno usmerjenih atributnih gramatik so sestavljene iz naslednjih lastnosti (aspektne lastnosti, ki jih bomo obravnavali v tem poglavju so zapisane krepko):

$$Jezik = RegDefIme + AtributiIme + PraviloIme + \mathbf{PointcutIme} \\ + \mathbf{NasvetIme} + \mathbf{UporabaNasveta}$$

Sledi formalna definicija večkratnega dedovanja posameznih aspektnih lastnosti.

Definicija stičišč

Za vsak specifikator stičišč pc v jeziku l, $Pointcuti(l)(pc)$ definira končno množico ujemajočih produkcij P, ki se ujemajo z vzorcem specifikatorja stičišč pc. Ujemanje se vrši nad celotno hierarhijo jezikov, začenši iz jezika, v katerem je definirana začetna produkcija.

$$Pointcuti : Jezik \rightarrow PointcutIme \rightarrow UjemajoceProdukcije$$
$$Pointcuti(l)(pc) = \{p_i \mid p_i \in P,\, p_i : X_{i0} \rightarrow X_{i1}X_{i2}...X_{in}, ujemanje(p_i, pc)\}$$

$S_1, S_2, \ldots, S_m$ označuje množico preslikav specifikatorjev stičišč, definiranih v jezikih $l_1, l_2, \ldots l_m$, v množice ujemajočih produkcij. Formalno preslikave definiramo kot:

$$\begin{aligned} S_1 &= \{pt_{11} \mapsto \{pr_{11}\}, pt_{12} \mapsto \{pr_{12}\}, \ldots, pt_{1k} \mapsto \{pr_{1k}\}\} \\ S_2 &= \{pt_{21} \mapsto \{pr_{21}\}, pt_{22} \mapsto \{pr_{22}\}, \ldots, pt_{1l} \mapsto \{pr_{1l}\}\} \\ &\vdots \\ S_m &= \{pt_{m1} \mapsto \{pr_{m1}\}, pt_{m2} \mapsto \{pr_{m2}\}, \ldots, pt_{mn} \mapsto \{pr_{mn}\}\}, \end{aligned}$$

kjer je pt_{ij} specifikator stičišč in pr_{ij} množica ujemajočih produkcij. Entiteto $S = S_2 \ominus \ldots \ominus S_m \ominus \triangle S_1$, kjer S_1 deduje od $S_2, \ldots, S_m$, formalno definiramo kot unijo vseh podedovanih produkcij, ki jih definirajo specifikatorji stičišč v celotni hierarhiji (vertikalno prekrivanje). Dedovanje specifikatorjev stičišč bi lahko definirali tudi kot preprosto unijo, a zaradi konsistence z ostalimi lastnostmi horizontalno prekrivanje preprečujemo (obravnavamo kot napako). Vertikalno prekrivanje razrešujemo z asimetričnim naslednikškim dedovanjem. V kolikor določen specifikator stičišč zbriše definicije starša, se prepreči dostop do starševskih definicij, zbriše pa se tudi množica že definiranih produkcij.

Nasveti

Za vsak nasvet ad definiran v jeziku l, predstavlja $Nasvet(l)(ad)$ končno množico ($ProdukcijaSemantika$) parov (p, R_p), kjer je p sintaksna produkcija in R_p končna množica generiranih[2] semantičnih funkcij pripadajočih produkciji p.

$$Nasvet : Jezik \rightarrow NasvetIme \rightarrow ProdukcijaSemantika$$
$$Nasvet(l)(ad) = \{(p, R_p) | p \in P,$$
$$p : X_0 \rightarrow X_1X_2...X_n,$$

[2]Kot smo govorili že v poglavju o aspektno usmerjenih atributnih gramatikah (poglavje 5.2), se končne semantične funkcije generirajo iz generičnih specifikacij le teh, ki so odvisne od produkcije, kateri pridružujemo semantične funkcije nasveta.

$R_p = \{X_i.a = f(X_{0.b}, \ldots, X_{j.c}) \mid X_i.a \in DefAtr(p)\}\}$

Dedovanje nasvetov je definirano na podoben način kot definiranje sintaksnih pravil s to razliko, da signaturo nasveta ne predstavlja zgolj ime, temveč tudi število formalnih parametrov nasveta. Pomembno pri nasvetih je to, da so semantične funkcije odvisne od dejanske produkcije na kateri želimo uporabiti nasvet. Te semantične funkcije se generirajo ob tkanju in pred operacijo *zdruzi*, ki smo jo definirali pri dedovanju sintaksnih pravil. Definicije dedovanja zaradi identičnosti z dedovanjem sintaksnih pravil in združevanja semantičnih funkcij ne bomo ponovno zapisovali.

Uporaba nasvetov

Ker se aplikacija nasvetov vedno vrši v ciljnem jeziku, ta lastnost aspektno usmerjenih atributnih gramatik ni predmet dedovanja. Kljub temu ima dedovanje vseh lastnosti, tudi neaspektnih, za aplikacijo nasvetov pomembno vlogo.

Večkratno dedovanje AOAG

Lastnosti dedovanja posameznih aspektnih lastnosti smo že navedli. Ostane nam torej še formalni zapis dedovanja aspektno usmerjenih atributnih gramatik.

Naj bo $AspectAG_1$, $AspectAG_2$, ..., $AspectAG_m$ aspektno usmerjena atributna gramatika, formalno definirana kot:

$AspectAG_1 = (G_1, A_1, R_1, Pc_1, Ad_1)$,
$AspectAG_2 = (G_2, A_2, R_2, Pc_2, Ad_2)$,
$\vdots$
$AspectAG_m = (G_m, A_m, R_m, Pc_m, Ad_m)$, potem je

$AspectAG = AspectAG_2 \oplus \ldots \oplus AspectAG_m \oplus \triangle AspectAG_1$, kjer $AspectAG_1$ deduje od $AspectAG_2, \ldots, AspectAG_m$ in je definirana kot:

$AspectAG = (G, A, R, Pc, Ad)$, kjer

$$\begin{array}{lcl} G & = & G_2 \oplus \ldots \oplus G_m \oplus \triangle G_1, \\ A & = & A_1 \ominus \ldots \ominus A_m, \\ R & = & R_1 \otimes \ldots \otimes R_m, \\ Pc & = & Pc_1 \ominus \ldots \ominus Pc_m, \\ Ad & = & Ad_1 \otimes \ldots \otimes Ad_m \ . \end{array}$$

5.4 Prednosti AOAG

Posamezni sklopi specifikacij programskega jezika so v atributnih gramatikah močno sklopljeni, kar ovira ponovno uporabo in modularno načrtovanje in implementacijo programskih jezikov. Prav tako je semantika programskega jezika vezana na sintakso programskega jezika, ki je definirana s produkcijami v notaciji BNF. Obe lastnosti atributnih gramatik lahko štejemo med glavne pomanjkljivosti le–teh. Predlagani pristop, aspektno usmerjene atributne gramatike, uspešno rešuje oba problema. Semantiko programskega jezika je s predlaganim pristopom mogoče načrtovati ločeno od sintakse. Semantiko programskega jezika zapišemo v nasvete, ki jih lahko uporabimo na več sintaksnih produkcijah v eni ali več komponentah. Modularnost in neodvisnost od definirane sintakse je zagotovljena s parametrizacijo nasvetov in definicijami uporabe le–teh ter z možnostjo definiranja generičnih semantičnih funkcij v nasvetih. S tem se bistveno zmanjša obseg specifikacij kot tudi modularnost. Ponovna uporaba je v veliki meri omogočena že z rahlo sklopljenostjo komponent ter sintakse in semantike specifikacij programskega jezika. Za izboljšanje modularnosti pa smo koncept AOAG razširili z možnostjo večkratnega dedovanja atributnih gramatik, ki sledi principom večkratnega dedovanja atributnih gramatik, kot ga predlagajo avtorji v [88].

Če povzamemo, so prednosti AOAG naslednje:

- možnost ločenega načrtovanja in implementacije sintakse in semantike programskega jezika,
- rahla sklopljenost komponent,
- ponovna uporaba komponent z mehanizmom večkratnega dedovanja in
- možnost generičnega zapisa semantike v nasvetih in parametrizacija njohove uporabe (možnost razvoja knjižnice aspektov).

If the only tool you have is a hammer, you tend to see every problem as a nail.

– Abraham Maslow –

Poglavje 6

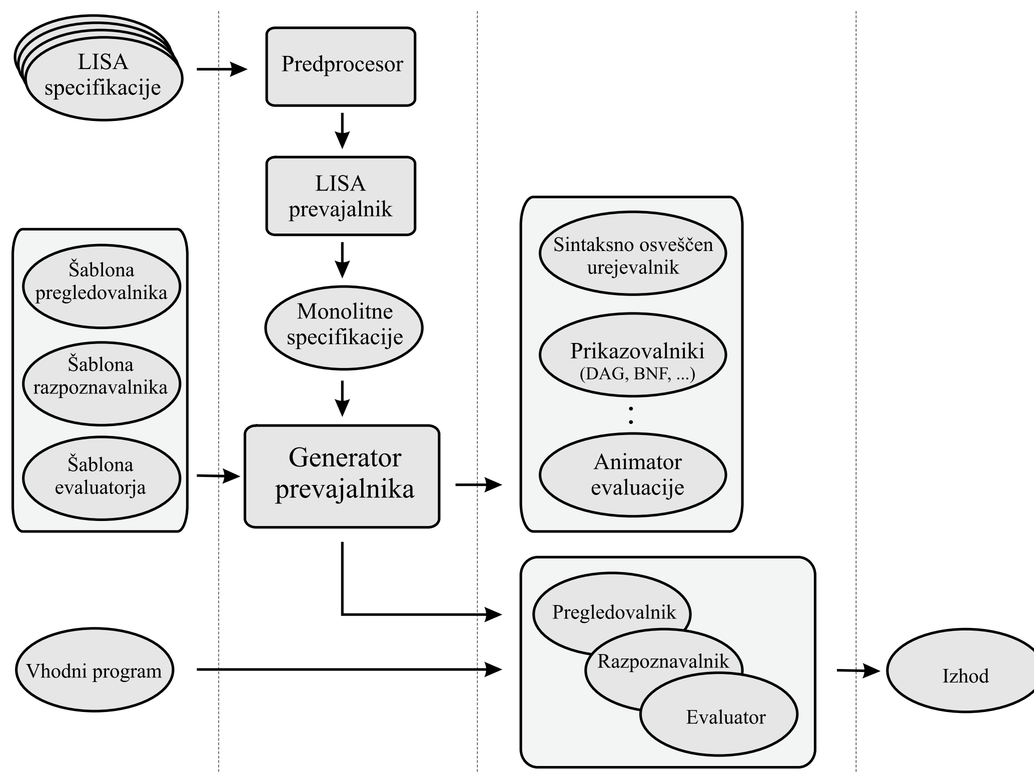
Orodje $LISA_{ver3.0}$

*V poglavju spoznamo orodje LISA$_{ver3.0}$, v nadaljevanju LISA (**L**anguage **I**mplementation **S**ystem Based on **A**ttribute Grammars). Orodje je avtomatski generator prevajalnikov in ostalih orodij za programske jezike. Razvito je bilo, in se še razvija, na Fakulteti za elektrotehniko, računalništvo in infromatiko, Univerze v Mariboru, v skupini pod vodstvom dr. Marjana Mernika. Orodje se uporablja v pedagoške namene, kakor tudi za razvoj prototipnih jezikov, preiskušanju novih koceptov pri razvoju programskih jezikov [101], in tudi uvajanju novih konceptov v same programske jezike [102]. V uvodu se seznanimo z orodjem in specifikacijskim jezikom orodja, ki temelji na formalnem modelu atributnih gramatik. V nadaljevanju so predstavljene novosti, ki jih uvaja disertacija. Specifikacijski jezik smo nadgradili s konceptom aspektno usmerjenih atributnih gramatik ter večkratnim dedovanjem le teh.*

6.1 Uvod

Orodje LISA [69] je sistem za avtomatsko generiranje prevajalnika (interpreterja) iz podanih formalnih specifikacij programskega jezika. Za specifikacije

programskega jezika orodje uporablja poseben domensko specifični jezik, ki temelji na formalnem modelu (aspektno usmerjenih) atributnih gramatik. Jezik podpira večkratno dedovanje. Ponovno uporabne komponente specifikacij programskega jezika so shranjene v eni ali več datotekah `.lisa`. V več korakih (odvisno od zapisanih specifikacij) se komponente pretvorijo v ločeno monolitno obliko specifikacij programskega jezika z atributnimi gramatikami za leksikalni, sintaksni in semantičen del programskega jezika. Iz monolitnih specifikacij se nato generira prevajalnik (pregledovalnik/razpoznavalnik) ter ostala orodja [99]. Statičen del prevajalnika in ostalih orodij je zapisan v predpripravljenih datotekah (šablonah), ki se uporabljajo pri generiranju. Izhod orodja je izvorni kod prevajalnika in ostalih orodij v programskem jeziku java. Arhitektura orodja s potekom generiranja prevajalnika je prikazana na sliki 6.1.



Slika 6.1: Arhitektura orodja LISA

Progami zapisani v novo definiranem programskem jeziku lahko z orodjem LISA tudi prevedemo in izvajamo. V ta namen lahko uporabniki uporabljajo razvojno okolje (IDE). Grafično razvojno okolje ponuja uporabnikom

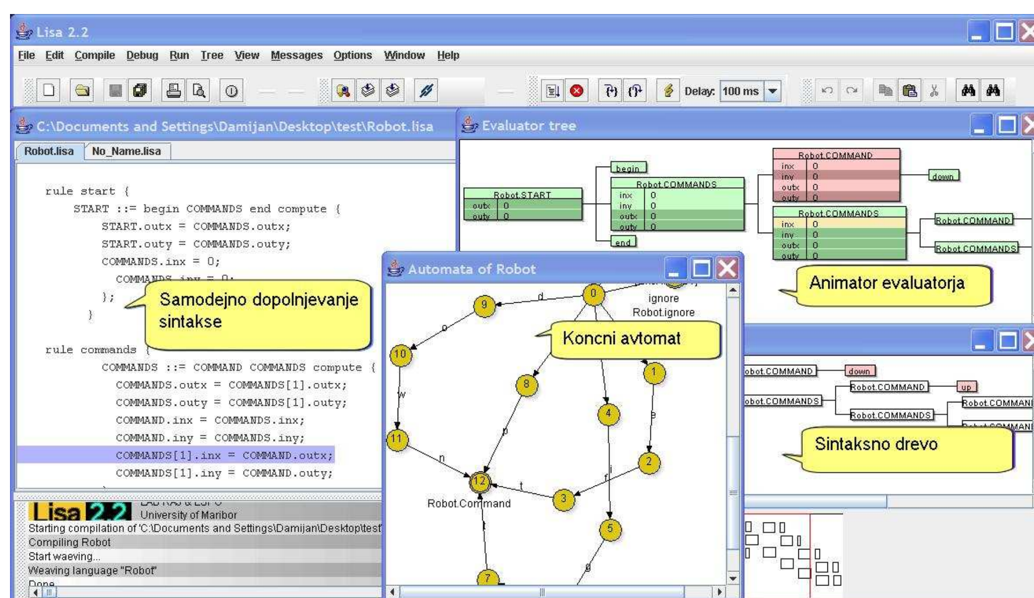
vizualno predstavitev nekaterih internih podatkovnih struktur. Tako je npr. mogoče opazovati korake izvajanje sintaksnega in semantičnega analizatorja. Animacija prikazuje posamezne korake izvajanja ovrednotenja semantičnega drevesa. Vizualizacija evaluatorja pomaga pri razvoju programskega jezika in lahko služi kot razhroščevalnik v fazi razvoja. Prav tako pa se ta pomožna orodja uporabljajo v didaktične namene, saj dobimo z opazovanjem delovanja generatorja boljše razumevanje delovanja prevajalnikov. Na ta način postane orodje LISA pomemben didaktični pripomoček pri učenju konceptov programskih jezikov in delovanju prevajalnikov.

Če povzamemo, so glavne značilnosti orodja naslednje:

- specifikacijski jezik podpira večkratno dedovanje atributnih gramatik in šablone v atributnih gramatikah, kar omogoča ponovno uporabnost specifikacij in inkrementalen razvoj programskih jezikov,
- orodje je arhitekturno neodvisno, saj je razvito v programskem jeziku java,
- orodje omogoča delo v tekstovnem (konzolnem) načinu ali pa delo z grafičnim vmesnikom, kjer lahko uporabniki specificirajo – generirajo – prevajajo – izvajajo programe v novo specificiranem programskem jeziku,
- generira različne tipe leksikalnih, sintaksih in semantičnih analizatorjev, ki jih lahko uporabniki uporabljajo tudi samostojno; trenutna različica podpira LL, SLR, LALR in LR razpoznavalnike ter *tree-walk*, *parallel*, *L-attribute* in *Katayama* evaluatorje,
- grafična predstavitev podatkovnih struktur: končni avtomat, BNF, sintaksno drevo, semantično drevo, graf odvisnosti itd. in
- animacija sintaksnega in semantičnega analizatorja.

Orodje LISA je dostopno tudi na spletu (*http://www.cs.feri.uni-mb.si*) [103]. Prav tako smo razvili spletno aplikacijo, ki je vmesnik do spletne storitve[1]. Podobno kot z orodjem je možno s spletno aplikacijo oz. storitvijo, ki se enostavno vgradi v aplikacijo, generirati razpoznavalnik/prevajalnik/... Generiran izvorni kod oz. preveden kod je mogoče prenesti na odjemalčev računalnik in ga uporabljati samostojno. Pri razvoju novega programskega

[1]Spletna storitev je dostopna na naslovu:
`http://marx.uni-mb.si:8080/LisaWebService/services/CServiceBean?wsdl`.



Slika 6.2: Grafični vmesnik orodja LISA

jezika ima uporabnik na voljo tri možnosti. Razvoja se lahko loti "od začetka" s pisanjem specifikacij oz. nalaganju le teh iz odjemalčevega računalnika. Druga možnost je nadgradnja specifikacij programskih jezikov, ki so shranjeni v repozitoriju, kamor lahko uporabniki tudi shranijo ter primerno dokumentirajo svoje rešitve. Tretja možnost je generiranja razpoznavalnika iz podanih pravilnih in nepravilnih programov [77]. Spletna aplikacija vodi uporabnika skozi korake generiranje razpoznavalnika. Generiran razpoznavalnik se lahko uporablja samostojno ali pa ga lahko pretvorimo v LISA komponento in jo nadalje razširimo še s semantiko jezika.

Spletna aplikacija/storitev je primerna za začetnike, saj jim zelo olajša delo (ni potrebno nalaganje orodja, delo s spletno aplikacijo je zelo intuitivno, ...), kakor tudi za eksperte. Začetniki lahko prve korake pri načrtovanju programskih jezikov storijo z uporabo dobro dokumentiranih rešitev v repozitoriju rešitev. Ekspertom pa je zraven skoraj celotne funkcionalnosti orodja LISA omogočena uporaba spletne storitve. Na ta način lahko spletno storitev uporabijo v svojih aplikacijah kot pregledovalnik/razpoznavalnik/... ali celo kot generator prevajalnika, kjer se jezik v aplikaciji pogosto spreminja [104].

6.2 Specifikacijski jezik

Kot smo omenili že uvodoma, smo za generator prevajalnikov LISA razvili poseben domensko specifični specifikacijski jezik. Pri načrtovanju jezika smo se trudili doseči čim večjo podporo modularnosti, ponovni uporabnosti in inkrementalnemu razvoju programskih jezikov. V grobem bi lahko specifikacijski jezik razdelili na komponentni del in na aspektni del. Komponentni del podpira modularen razvoj na nivoju sintaksne, kakor tudi semantične specifikacije jezikov z mehanizmi, kot so večkratno dedovanje atributnih gramatik in šablonami v atributnih gramatikah [88]. O večkratnem dedovanju atributnih gramatik smo že govorili, šablonam pa ne bomo posvetili veliko pozornosti, saj se šablone ne nanašajo na področje disertacije. Šablone zagotavljajo modularnost in ponovno uporabnost specifikacij jezika na sintaksnem nivoju, mi se pa v disertaciji ukvarjamo predvsem s semantičnim delom specifikacij programskih jezikov. Podrobnejši pregled opisanih konceptov je dostopen na spletni strani [69] in v delih [1, 88]. Drugi del specifikacij je aspektni del, ki predstavlja enega od izvirnih prispevkov disertacije. Specifikacije smo uspešno nadgradili s koncepti aspektno usmerjenih atributnih gramatik (poglavje 5, stran 75) in večkratnim dedovanjem aspektno usmerjenih atributnih gramatik (poglavje 5.3, stran 83) ter jih integrirali v orodje LISA.

Vsaka datoteka `.lisa` lahko vsebuje enega ali več jezikov (komponent oz. razredov), lahko pa vključuje tudi komponente v drugih datotekah. Primer definicije komponente je prikazan na primeru 6.1.

Primer 6.1 Osnovna sintaksa komponente v orodju LISA

```
import "Robot.lisa";       // vključitev drugih komponent -- opcijsko
language RobotExtensions  extends Robot {
   // leksikalne specifikacije
   // specifikacije atributov
   // sintaksne specifikacije
   // semantične specifikacije
   // operacije nad domeno
   // definicija stičišč
   // definicija nasvetov (advice)
   // definicija uporabe nasvetov na ustrezna stičišča
}
```

6.2.1 Komponentni del jezika

Komponentni del jezika je sestavljen iz: leksikalnih definicij, sintaksnih pravil, definicij atributov in operacij nad domenami.

Leksikalne definicije

Leksikalne definicije definiramo z rezervirano besedo **lexicon**. Iz tega dela specifikacij LISA generira izvorni kod pregledovalnika v programskem jeziku java. Osnovne leksikalne simbole definiramo s poimenovanimi regularnimi izrazi (regularne definicije). Vsaka regularna definicija mora imeti unikatno ime in je lahko izpeljana (*extends*), iz enega ali več staršev, oz. redefinirana (*overrides*) v podedovanem jeziku. V primeru 6.2 sta definirani dve regularni definiciji: `Commands` in `ReservedWord`. Rezervirana beseda **ignore** je uporabljena za definicijo znakov in osnovnih simbolov, ki naj jih pregledovalnik ignorira. Ti znaki oz. leksikalni simboli niso vključeni v seznam osnovnih simbolov. Formalna sintaksa leksikalnih definicij je naslednja:

```
lexicon {
   [[Q] overrides | [Q] extends] R RegularniIzraz
  ⋮
}
```

Sintaksna pravila

Sintaksne (BNF produkcije) in semantične (semantične funkcije) specifikacije so kapsulirane v LISA sintaksnih pravilih (pravilo definira rezervirana beseda **rule**). Za specifikacije sintakse programskega jezika je v LISA specifikacijskem jeziku uporabljena dobro uveljavljena notacija BNF [17]. Produkcije konteksno neodvisne gramatike so navedene v "rule" delu sintaksnega pravila (npr. `START ::= begin COMMANDS end` ; po dogovoru pišemo prvo črko neterminalnega simbola z veliko črko, terminalni simboli so zapisani z malimi črkami). Sintaksna pravila predstavljajo vmesnik specifikacij programskega jezika in so lahko v izpeljanem jeziku dopolnjene oz. izbrisane (*extends/overrides*). Semantika programskega jezika je opisana s semantičnimi funkcijami (atributna gramatika), ki so zapisane v "compute" (definiran z rezervirano besedo **compute**) delu. Neterminalnim simbolom so v atributni gramatiki pridruženi atributi, katerih vrednosti se s pomočjo semantičnih funkcij spreminjajo in prenašajo po semantičnem drevesu. Prav zato morajo biti semantične funkcije definirane za vse produkcije. Semantične funkcije so zapisane v programskem jeziku java, v zapisu pa je mogoče uporabiti vse razrede aplikacijskega programskega vmesnika java (*Java API*). To je mogoče, ker je izhod generatorja izvorni kod v

programskem jeziku java. Prinaša pa tudi nekaj slabosti, saj vseh napak v semantičnih funkcijah tako ni mogoče javljati na nivoju orodja LISA. Formalna sintaksa sintaksnih pravil je naslednja:

```
rule [[Y] extends | [YY] overrides] Z {
    X ::= X_11 X_12 ... X_1p compute {
            semantične funkcije
          }
    ⋮
    |
          X_r1 X_r2 ... X_rt compute {
            semantične funkcije
          };
    ⋮
}
```

Kot je razvidno iz zapisa sintaksnega pravila, lahko sintaksno pravilo vsebuje več BNF produkcij.

Definicija atributov

Semantika programskega jezika je v atributnih gramatikah določena z atributi, ki so pridruženi vsem neterminalnim simbolom. Atributi v orodju LISA so lahko poljubnega javanskega tipa (primitivnega ali razrednega), tudi uporabniško določenega (definiranega v "method" delu oz. v zunanjih knjižnicah). Ponavadi enako ime atributa uporabimo pri več neterminalnih simbolih. V ta namen lahko uporabimo nadomestni znak (*wildcard*) '*', ki nadomesti ime neterminalnega simbola. Tak atribut lahko pridružimo poljubnemu, lahko pa tudi vsem neterminalnim simbolom. LISA od razvijalca ne zahteva podajanja vrste atributov (podedovani, pridobljeni), to orodje izračuna samodejno. Atribute definiramo z rezervirano besedo **attributes**, formalna sintaksa je naslednja:

```
attributes Java_type At_1, ..., At_M ;
⋮
```

Operacije nad domenami

Semantične domene in operacije nad domenami so v orodju LISA zapisane v "method" delu specifikacij. Operacije so označene z rezervirano besedo **method**. Vsaka metoda ima unikatno ime in jo lahko v

izpeljanem razredu razširimo, brišemo ali spremenimo. Kot že rečeno, LISA generira izvorni kod v programskem jeziku java. Prav zato so tudi celotne operacije nad domenami specificirane v programskem jeziku java. Ves kod, ki ga specificiramo v "method" delu, se vključi v razred, ki implementira evaluator za specificiran jezik. Operacije nad domenami so lahko delno ali v celoti uvožene iz že obstoječih komponent (npr., `import Robot.calculations.* ;`). Sintaksa zapisa operacij nad domenami je naslednja:

```
method [[N] overrides | [N] extends] M {
   Javanski kod operacij nad domenami
}
⋮
```

Za boljšo ilustracijo sintakse LISA specifikacij podajamo primer specifikacij prototipnega jezika za krmiljenje robota (primer 6.2). V jezik so vgrajeni ukazi za premikanje robota (*right, left, up, down*). Pomen (semantika) programa je izračun robotove končne lokacije. Izhod primera programa `begin down down left end` je tako `{outp.x=-1, outp.y=-2}` (`in.x` in `in.y` sta inicializirana na 0).

6.2.2 Domensko specifični aspektni jezik

Pri načrtovanju in zapisu specifikacij programskih jezikov pogosto naletimo na koncepte, ki jih ni moč zapisati modularno, temveč je zapis teh konceptov prepleten z drugimi specifikacijami. To v praksi pomeni, da je potrebno nek koncept oz. nek del semantike programskega jezika zapisati v več modulov specifikacij, kar bistveno oteži sam razvoj, še bolj pa vzdrževanje in nadaljni razvoj programskega jezika. Ta slabost je posebej vidna, ko želimo v že obstoječ programski jezik vključiti nove koncepte. Npr., če želimo v jezik aritmetičnih izrazov vključiti prireditveni stavek, je zraven nove sintakse in pripadajoče semantike (sintaksa in semantika vključenega koncepta) potrebno zapisati semantične funkcije za prenos atributov okolja spremenljivke po celotnem semantičnem drevesu. Takšne spremembe so zelo zahtevne in zahtevajo spremembo celotnih specifikacij (kljub uporabi dedovanja). Z željo po izboljšanju modularnosti in zmanjšanju prepletanja semantičnih konceptov v LISA specifikacijah smo v specifikacije integrirali koncepte aspektno usmerjenih atributnih gramatik [101] in dedovanja aspektno usmerjenih atributnih gramatik, ki smo jih predstavili v poglavju 5.

Primer 6.2 Specifikacije programskega jezika Robot

```
language Robot {
 lexicon {
  Commands left | right | up | down
  ReservedWord  begin | end
  ignore [\0x0D\0x0A\ ]   // ignoriraj
 }
 attributes Point *.inp, *.outp;

 rule start {
 START ::= begin COMMANDS end  compute {
   START.outp = COMMANDS.outp;
    // robot position in the beginning
   COMMANDS.inp = new Point(0, 0); };
 }

 rule moves {
 COMMANDS ::= COMMAND COMMANDS  compute {
   COMMANDS[0].outp = COMMANDS[1].outp;
   COMMAND.inp = COMMANDS[0].inp;
   COMMANDS[1].inp = COMMAND.outp; }
 |  epsilon  compute {  // epsilon (prazna) produkcija
   COMMANDS.outp = COMMANDS.inp; };
 }

 rule move {
 COMMAND ::= left  compute {
   COMMAND.outp = new Point((COMMAND.inp).x-1,(COMMAND.inp).y); };
 COMMAND ::= right  compute {
   COMMAND.outp = new Point((COMMAND.inp).x+1,(COMMAND.inp).y); };
 COMMAND ::= up  compute {
   COMMAND.outp = new Point((COMMAND.inp).x,(COMMAND.inp).y+1); };
 COMMAND ::= down  compute {
   COMMAND.outp = new Point((COMMAND.inp).x,(COMMAND.inp).y-1); };
 }
}
```

Pri integraciji konceptov aspektno usmerjenih atributnih gramatik smo komponentni del LISA specifikacijskega jezika razširili z definicijo stičišč (*pointcuts*), definicijo nasvetov (*advice*) in uporabo nasvetov (*advice application*). Pomemben del vsakega aspektnega jezika je model stičišč (JPM), o katerem smo razpravljali že v poglavju 3.5.1. Glede na vprašanja, ki smo si ji jih zastavili v dotičnem poglavju so stičišča v LISA specifikacijskem jeziku statične točke v specifikacijah, kjer lahko dodamo določen semantičen koncept. Te točke so v LISA specifikacijah definicije BNF produkcij.

Eden izmed pomembnejših lastnosti LISA specifikacijskega jezika je možnost večkratnega dedovanja, ki bistveno pripomore k modularnosti in ponovni uporabnosti specifikacij programskega jezika. Da bi zagotovili konsistentnost celotnega jezika, so tudi aspektne lastnosti predmet (večkratnega) dedovanja. V ta namen je bilo potrebno nekoliko prirediti formalni model dedovanja aspektno usmerjenih atributnih gramatik, ki smo jih uspešno integrirali v LISA specifikacijski jezik.

Preden se lotimo opisa posameznih aspektnih lastnosti jezika ,si poglejmo primer programa zapisanega z aspektnim LISA specifikacijskim jezikom. Primer 6.3 prikazuje razširitev jezika Robot (primer 6.2) s konceptom merjenja časa. Pomen določenega stavka, tako niso samo končne koordinate robota, ampak tudi čas, ki ga je robot potreboval do cilja. Za vsak premik (ukaz) potrebuje robot enoto časa. Na tem mestu se še ne bomo spuščali v podroben opis jezika, temveč se bomo, na primer, sklicevali pri opisu posameznih aspektnih lastnosti LISA jezika.

Definicija (opis) stičišč

Množico stičišč (ponavadi se uporablja zapis za opis več stičišč hkrati) v aspektnih jezikih opišemo s specifikatorjem stičišč. Stičišča v LISA specifikacijah so BNF produkcije, ki imajo pridružene semantične funkcije. Omogočiti moramo torej mehanizem, ki bo na unikaten način zajel množico BNF produkcij. BNF produkcije so sestavljene iz levega dela (neterminalni simbol) in desnega dela (sestavljajo ga neterminalni, terminalni in prazen simbol ε), le–te pa so zapisane v obliki niza (*string*). Na dlani je torej rešitev z ujemanjem vzorcev (*pattern matching*). Definirati je potrebno vzorec, ki se bo ujemal z željeno množico BNF produkcij. Ker so BNF produkcije kapsulirane v sintaksnih pravilih, ta pa so definirana v nekem jeziku v celotni hiearhiji le–teh, je potrebno v vzorec zajeti še ti dve informaciji. Za opis vzorca smo uvedli dva nadomestna znaka. Nadomestni znak '..' se ujema na nič ali več neterminalnih ali terminalnih simbolov in ga lahko uporabimo za definicijo desne strani produkcije. Nadomestni znak '$*$' pa se ujema na del oz. celi niz, ki lahko predstavlja terminalni ali neterminalni

Primer 6.3 Specifikacije programskega jezika `RobotTime`

```
language RobotTime  extends Robot {
 attributes  int *.time;

 // vse produkcije v pravilu "move"
 pointcut Command   *.move : COMMAND  ::= .. ;

 // startno pravilo je potrebno vedno (re)definirati
 // tudi v primeru, ko temu pravilu ne spreminjamo semantike
 rule  extends start {
  START ::= begin COMMANDS end  compute {
    START.time = COMMANDS.time;
  };
 }
 rule  extends moves {
  COMMANDS ::= COMMAND COMMANDS  compute {
    COMMANDS[0].time = COMMAND.time + COMMANDS[1].time;
  }
  |  epsilon  compute {
    COMMANDS.time = 0;
   };
  }
   advice CommandTime<C> {
    C.time=1;
  }

  // semantiko nasveta "CommandTime" združimo s semantiko vseh pravil,
  // ki se ujemajo na pointcut "Command"
  // Dejanski parameter "COMMAND" se ob tkanju zamenja s formalnim
  // parametrom "C"
  apply CommandTime<COMMAND>  on Command;
}
```

simbol, ime sintaksnega pravila in jezika. Uporabljamo ga lahko v katerem koli nizu v definiciji specifikatorja stičišč. Za boljše razumevanje podajmo nekaj krajših primerov, ki se nanašajo na specifikacije jezika s primera 6.2 (rezervirano besedo in identifikator specifikatorja stičišč smo izpustili).

`*.* : * ::= .. ;`	*označi (ujema) vse produkcije, v vseh pravilih in jezikih, ne glede na ime jezika (celotna hiearhija jezikov)*
`Robot.m* : * ::= .. ;`	*označi vse produkcije v pravilih, katerih ime se začne z* `m` *, v jeziku* `Robot`
`Robot.move : COMMAND ::= left ;`	*označi natančno eno produkcijo:* `COMMAND ::= left` *v pravilu* `move` *jezika* `Robot`

Specifikator stičišč v orodju LISA definiramo z rezervirano besedo **pointcut**. Vsak specifikator stičišč mora imeti, znotraj ene komponente, unikatno ime. S poimenovanjem dosežemo možnost apliciranja večih nasvetov na isto množico stičišč, omogočimo pa tudi ponovno uporabo te lastnosti jezika (dedovanje). Formalna sintaksa za opis množice stičišč je naslednja:

pointcut `[[`*`NovoIme`*`]` **extends** | **overrides**`]` `[L.]`*`Ime`* *Vzorec*;

Specifikatorje stičišč z enakim imenom lahko ponovno uporabimo. Specifikator stičišč lahko razširimo (*extends*) ali redefiniramo (*overrides*). To lahko storimo v istem ali izpeljanem razredu. Specifikator stičišč definira množico stičišč, tako se v primeru izpeljave izračuna unija množic stičišč, definiranih z vsemi definicijami v hiearhiji. V spodnjem primeru je množici stičišč, ki se ujamejo na specifikator stičišč `Command` dodana novo definirana množica stičišč.

pointcut `Command1` **extends** `Command` *`Vzorec`* ;

Redefinirani specifikatorji stičišč v izpeljanih razredih niso vidni (dostopni). V spodnjem primeru definiramo specifikator stičišč `Command2`, ki redefinira že obstoječega `Command1`.

pointcut `Command2` **overrides** `Command1` *`Vzorec1`* ;

V praksi to pomeni, da se že obstoječa množica stičišč pobriše, v novo množico stičišč pa se dodajo le produkcije, ki se ujemajo z novo definiranim vzorcem. V kolikor bi v izpeljanih jezikih želeli razširiti specifikator stičišč `Command1`, bi LISA prevajalnik javil napako, da specifikator

stičišč s tem imenom ne obstaja. Seveda pa to ne pomeni, da teh specifikatorjev stičišč ni možno ponovno uporabiti. Pri ponovni uporabi je potrebno navesti le celotno ime `Jezik.Pointcut`.

Definicija nasveta

Nasveti (*advice*) v LISA specifikacijskem jeziku definirajo ponovno uporabne polimorfne abstrakcije semantičnih funkcij, ki niso vezane zgolj na eno BNF produkcijo. Semantične funkcije so parametrizirane s poljubnim številom terminalnih oz. neterminalnih simbolov ali atributov. Način podajanja semantike v nasvetih je podoben kot pri šablonah v atributnih gramatikah [88]. V nasvetu definiramo dodatno semantiko (razširitev), ki jo lahko dodamo že definirani semantiki na določenem stičišču (produkciji) in ne posega v strukturo (sintakso) programskega jezika. V aspektnih jezikih imajo razvijalci ponavadi možnost zapisa več nasvetov oz. možnost različne aplikacije nasveta na določen *pointcut*, kot npr: *before, after, around, ...* V orodju LISA imamo na voljo samo en način uporabe nasvetov. Vzroke zato najdemo v deklarativnosti LISA specifikacijskega jezika. Vrstni red semantičnih funkcij znotraj določene produkcije ni pomembem, zato tudi več vrst aplikacij ni podprto.

Formalna sintaksa definicije nasveta je naslednja:

```
advice [[NovoIme] extends | overrides] [JezikIme.]Ime<arg> ;
```

Na primeru 6.3, kjer obstoječi jezik za krmiljenje robota razširjamo s konceptom izračuna časa; enako semantično funkcijo uporabimo na več produkcijah. S tem se izognemo nepotrebnemu ponavljanju enakih semantičnih funkcij, ki so v atributni semantiki pogost pojav.

```
advice CommandTime<C> { C.time = 1; } ;
```

V orodju LISA lahko definiramo tudi abstraktne semantične funkcije, ki so neodvisne od strukture produkcije, na katero jih želimo aplicirati. Konkretne semantične funkcije se generirajo ob času tkanja in so odvisne od oblike produkcije, ki ji želimo dodati semantiko nasveta in uporabljenih dejanskih parametrov. Te abstrakcije lahko uporabimo za definicijo vzorcev, ki se pogosto pojavljajo v specifikacijah programskih jezikov. Ti vzorci so naslednji: distribucija vrednosti, distribucija seznama, konstrukcija vrednosti, konstrukcija seznama, *bucket brigade* itd. Kot primer si poglejmo vzorec *bucket brigade left*, ki je definiran na naslednji način:

```
Y ::= X1 X2 ... XN
 { X1.in = Y.in; X2.in = X1.out;
 ⋮
 XN.in = XN-1.out; Y.out = XN.out; }
```

Enak vzorec definiramo še v LISA specifikacijskem jeziku.

```
advice bucketBrigadeLeft<inAtt, outAtt> {
   if (empty) {
      LHS.outAtt = LHS.inAtt ;
   }
   else {
     first.inAtt = LHS.inAtt;
     RHS.inAtt = pred.outAtt;
     LHS.outAtt = last.outAtt;
   }
}
```

Rezervirani besedi LHS in RHS označujeta neterminalni simbol na levi strani produkcije in seznam neterminalnih simbolov na desni strani produkcije. Za obdelavo seznama neterminalnih simbolov smo definirali naslednje funkcije:

- *empty* – vrne *true*, če je seznam prazen,
- *pred* – vrne prejšnji element seznama (za prvi element seznama funkcija ni definirana),
- *succ* – vrne naslednji element seznama (za zadnji element seznama funkcija ni definirana),
- *last* – vrne zadnji simbol seznama in
- *first* – vrne prvi element seznama.

V primerih, ko funkcija ni definirana, se semantična funkcija ne generira. Za obdelavo seznamov smo v jezik vključili tudi pogojni stavek (**if**..**else**). Kot je razvidno iz primera vzorca, bi za obdelavo seznama potrebovali tudi zanke, a jih v sam jezik nismo vključili. V kolikor leva stran produkcije v nasvetu vsebuje rezervirano besedo RHS (nekaj želimo prirediti seznamu oz. v našem primeru vsem neterminalnim simbolom v seznamu), se implicitno izvrši zanka, ki generira semantično funkcijo za vse elemente seznama. Edini pogoj za generiranje semantične funkcije je, da se vse uporabljene funkcije nad seznami v tej semantični funkciji izvedejo.

Dedovanje nasvetov je implementirano podobno kot dedovanje sintaksnih pravil, kar ne bi smelo presenečati, saj z obema formalizmoma definiramo semantiko programskega jezika. Ob razširjanju nasveta je torej potrebno združiti semantiko dveh ali več nasvetov/produkcij. V izpeljanih razredih lahko razširimo samo nasvete z enako signaturo (enakim imenom in številom formalnih parametrov). V primeru redefiniranja nasveta je postopek enak kot pri ostali lastnostih LISA jezika.

Uporaba nasvetov

V tem delu aspektnih LISA specifikacij definiramo uporabo nasvetov na definiranih specifikatorjih stičišč. Uporabo nasveta definiramo z rezervirano besedo **apply**, ki ji sledi ime nasveta z dejanskimi parametri (število dejanskih parametrov in formalnih parametrov nasveta se mora ujemati), rezervirano besedo **on** in imenom specifikatorja stičišč. Prvi korak apliciranja nasvetov je iskanje (*lookup*) nasvetov in specifikatorjev stičišč ter primerjava parametrov. Iskanje poteka po enakem principu kot pri vseh objektno usmerjenih jezikih z implementiranim nasledniškim dedovanjem. Po uspešnem iskanju sledi združevanaje semantike definirane v nasvetu s semantiko definirano v sintaksnih pravilih za vse produkcije, ki jih definira vzorec specifikatorja stičišč. Ta korak natančneje razložimo v poglavju, kjer se ukvarjamo s tkanjem.

Formalna sintaksa aplikacije nasvetov je naslednja:

```
apply [overwrite] [JIme.]AdviceIme<args> on [JIme1.]PointcutIme ;
```

Opisano si poglejmo še na primeru. Npr. stavek

```
apply CommandTime<COMMAND>  on Command;
```

definira aplikacijo nasveta `CommandTime` na specifikator stičišč `Command`. Semantiko nasveta `CommandTime` želimo torej združiti s semantiko produkcij, ki so definirane v sintaksnem pravilu `move`. Če ne bi uporabili aspektnih lastnosti jezika LISA, bi bilo potrebno v nasvetu definirano semantično funkcijo zapisati večkrat oz. ponovno redefinirat več sintaksnih pravil in jim dodati ustrezno semantiko. “Originalne” specifikacije sintaksnega pravila `move` bi v tem primeru bile naslednje:

```
rule  extends move {
 COMMAND ::= left  compute {
   COMMAND.time = 1;
 };
```

```
  COMMAND ::= right  compute {
    COMMAND.time = 1;
  };
  COMMAND ::= up  compute {
    COMMAND.time = 1;
  };
  COMMAND ::= down  compute {
    COMMAND.time = 1;
  };
}
```

Pri združevanju semantike nasvetov in sintaksnih pravil lahko pride v primeru definirane iste semantične funkcije do konfliktne situacije, saj je lahko za vsak atribut določenega neterminalnega simbola definirana zgolj ena semantična funkcija. Pri tem se poraja vprašanje, katero semantično funkcijo uporabiti. V ta namen smo pri definiciji aplikacije nasvetov uvedli novo rezervirano besedo **overwrite**. S to rezervirano besedo lahko razvijalci definirajo, da bodo vse semantične funkcije definirane v nasvetu vedno vključene v semantiko sintaksnih pravil. V primeru podvajanja se že obstoječe semantične funkcije prepišejo. Privzeto (kadar ne definiramo "overwrite") se takšne semantične funkcije ignorirajo in se ne prepišejo. Npr., pri uporabi rezervirane besede **overwrite** v naslednjem stavku:

```
apply  overwrite bucketBrigadeLeft<inEnv, outEnv>  on vseProd ;
```

se določene semantične funkcije, ki ustrezajo vzorcu specifikatorja stičišč `pointcut vseProd *.* : * ::= .. ;` (v tem primeru vse produkcije) prepišejo. Pri razvoju nasvetov in njihovi uporabi mora biti razvijalec zelo pozoren na to, ali naj se v končni semantiki uporabijo semantične funkcije nasveta ali "originalne" semantične funkcije definirane v produkcijah.

Na tej točki so določeni nasveti zgolj označeni za tkanje. Dejansko tkanje se izvaja na končnem jeziku (jezik, ki ima definirano startno produkcijo in se nahaja na najvišjem nivoju hierarhije jezikov) ob prevajanju celotnih specifikacij.

Čeprav je dedovanje pri uporabi nasvetov zelo pomembno, ta lastnost jezika ni predmet dedovanja. Ko nekje v specifikacijah definiramo aplikacijo nasveta, se le–ta izvrši v končnem jeziku ob prevajanju specifikacij in generiranju monolitne oblike specifikacij atributne gramatike. Na samo aplikacijo nasveta lahko posredno vplivamo torej z dedovanjem ostalih lastnosti LISA jezika. Na tem mestu se tudi izkažejo

prednosti, ki jih prinaša uvedba dedovanja v aspektno usmerjene specifikacije atributnih gramatih. Pri aplikaciji nasvetov lahko uporabimo vse aspektne, posredno pa tudi ostale, lastnosti jezikov. Če uporabimo kratka imena[2], je potreben *lookup* za iskanje ustrezne lastnosti. Redefinirane (*overriden*) lastnosti so v tem primeru nedosegljive in jih ignoriramo. V nasprotnem primeru *lookup* ni potreben, prav tako pa lahko uporabimo vse lastnosti, tudi redefinirane. S tem mehanizmom lahko zagotovimo, da uporabljene lastnosti v nadaljevanju specifikacij (izpeljanih jezikih) ne bodo spremenjene. V osnovi ločimo štiri različne načine uporabe nasvetov:

apply `AdviceName<args>` **on** `PointcutName ;`

V tem primeru sta obe lastnosti (nasvet in specifikator stičišč) razrešena v končnem jeziku. Posledica tega je možnost nadaljne spremembe obeh lastnosti v izpeljanih jezikih po definiranju aplikacije nasveta.

apply `LanName.AdviceName<args>` **on** `PointcutIme ;`

V tem primeru je nasvet natačno znan, specifikator stičišč pa lahko v izpeljanih jezikih še spremenimo in bo razrešen v končnem jeziku.

apply `AdviceName<args>` **on** `LanName.PointcutName ;`

Podobno kot v prejšnjem primeru, le da tokrat nasvet razrešujemo v končnem jeziku, množica stičišč pa je znana vnaprej.

apply `LanName.AdviceName<args>` **on** `LanName1.PointcutName ;`

V tem primeru za obe lastnosti preprečimo nadaljne spreminjanje. Posredno lahko na aplikacijo v tem primeru vplivamo zgolj s spreminjanjem neaspektnih lastnosti LISA specifikacijskega jezika.

Za boljšo ilustracijo konceptov aspektnih lastnosti LISA specifikacijskega jezika in (večkratnega) dedovanja le–teh si poglejmo razširitev jezikov `RobotTime` (6.3) in `Expressions` (jezik preprostih aritmetičnih izrazov). Primer programa v definiranem programskem jeziku je: **begin** `down 5 right 3*(2+1)` **end** s pomenom `{time=2; outp.x=9; outp.y=-5}`. Jezik za krmiljenje robota smo želeli razširiti z možnostjo podajanja velikosti premika. Tako lahko

[2] Vsaka lastnost v LISA specifikacijskem jeziku ima dve imeni. Kratko ime je zgolj ime lastnosti (sintaksno pravilo, atribut, regularna definicija, nasvet, ...), dolgo ime pa je sestavljeno iz imena jezika (komponente specifikacij) in imena lastnosti, ločenih s piko (`Jezik.Lastnost`).

Primer 6.4 Specifikacije programskega jezika `RobotCalc`

```
language RobotCalc  extends RobotTime, Expressions {
 lexicon {
   ignore  [\ \0x0D\0x0A\0x09]+
 }
 rule  extends start {
   START ::= begin COMMANDS end  compute { };
 }
 rule  extends move {
   COMMAND ::= left EXPR  compute {
     COMMAND.outp =
       new Point((int)COMMAND.inp.getX()-EXPR.val, (int)COMMAND.inp.getY());
   };
   COMMAND ::= right EXPR  compute {
     COMMAND.outp =
       new Point((int)COMMAND.inp.getX()+EXPR.val, (int)COMMAND.inp.getY());
   };
   COMMAND ::= up EXPR  compute {
     COMMAND.outp =
       new Point((int)COMMAND.inp.getX(), (int)COMMAND.inp.getY()+EXPR.val);
   };
   COMMAND ::= down EXPR  compute {
     COMMAND.outp =
       new Point((int)COMMAND.inp.getX(), (int)COMMAND.inp.getY()-EXPR.val);
   };
 }
}
```

specificiramo, da naj se robot premakne za `3 * 5` v levo (`left 3*5`). Primer specifikacij je prikazan na primeru 6.4.

Neterminalni simbol `EXPR` je podedovan iz jezika `Expressions`, ki implementira preprost jezik aritmetičnih izrazov (primer programčka v tem jeziku je `3*(5+2)` s pomenom `{val=21}`). Večina lastnosti v primeru je podedovanih iz jezika `RobotTime`. V jeziku `RobotCalc` smo redefinirali startno produkcijo, ki pa ji nismo dodali semantike. Prav tako smo redefinirali vse produkcije v sintaksnem pravilu `move`, kjer smo razširili (spremenili) tudi semantične funkcije. Razlog za to je razširitev sintakse programskega jezika. Zelo pomembno prednost aspektno usmerjenih specifikacij opazimo v definiciji semantičnih funkcij za izračun časa, ki jih v novem jeziku ni potrebno ponovno definirati. Razlog leži v definiciji specifikatorja stičišč, ki ustreza tudi spremenjeni sintaksi produkcij v sintaksnem pravilu `move`. Če ne bi uporabili aspektnega pristopa, bi bilo potrebno semantiko za izračun časa ponovno (pre/za)pisati v novih produkcijah, kar vodi v nepotrebno podvajanje zapisa semantičnih funkcij in nepotrebnih napak. Otežuje pa tudi vzdrževanje samega program-

skega jezika. Koncept smo prikazali na zelo majhnem primeru, kjer pa lahko zasledimo morebitne pasti specificiranja programskih jezikov z atributnimi gramatikami in prednostmi, ki jih prinaša naš pristop. Celotni koncept podrobneje prikažemo na večjem reprezentativnejšem primeru (poglavje 6.4).

Implementacija dedovanja

Pri implementaciji večkratnega dedovanja AOAG smo ohranili isti koncept, ki je implementiran za ostale neaspektne lastnosti specifikacijskega jezika. Vertikalno prekrivanje identifikatorjev razrešujemo z asimetričnim nasledniškim dedovanjem, medtem ko je razreševanje horizontalnega prekrivanja prepuščeno razvijalcu programskega jezika. V kolikor dva ali več nadjezikov vsebuje več enakih identifikatorjev (npr. nasvetov z enakimi imeni), je prekrivanje mogoče rešiti s kanoničnim zapisom identifikatorja (`Jezik.ImeId`).

Pri določenih lastnostih (npr. specifikator stičišč) je možno lastnost razširiti z več starši, saj je izpeljana lastnost unija lastosti definiranih v nadjezikih. Podobna implementacija je možna tudi pri definiciji nasvetov. Rešitve nismo uporabili, ker se s takšno rešitvijo poslabša berljivost specifikacij in poveča njihovo kompleksnost. Težko je namreč slediti uporabi nasvetov, kar lahko rezultira k pogostejšim napakam pri specificiranju semantike programskega jezika.

6.2.3 Aspektno tkanje

Srce vsakega prevajalnika za aspektne jezike je aspektni tkalec (*aspect weaver*). Naloga tkalca je kod v aspektnih modulih preplesti s kodom zapisanim v komponentnem delu jezika. Kod v nasvetih je torej potrebno združiti s kodom v modulih na mestu, ki jih definiramo s specifikatorjem stičišč. V orodju LISA smo v ta namen razširili prevajalnik. Proces tkanja je skrit v prevajalniško fazo LISA generatorja in sledi prvi fazi prevajanja LISA specifikacij, kjer se vrši razpoznava LISA izvornega koda in ustvarjanje potrebnih podatkovnih struktur, tudi za aspektne lastnosti, jezika. Zaradi dobre testiranosti LISA prevajalnika, le–tega nismo veliko spreminjali. Nadgradili smo ga z razpoznavo aspektnih lastnosti LISA jezika in z generatorjem podatkovnih struktur, ki jih potrebuje tkalec. Tkalec nato "popravi" podatkovne strukture neaspektnih lastnosti jezika tako, da doda semantiko zapisano v nasvetih semantiki ustreznih produkcij. Po končanem tkanju dobimo podatkovne strukture celovitih LISA specifikacij, katere pretvorimo v monolitne specifikacije atributnih gramatik. Iz teh LISA generator zgenerira prevajalnik na običajen način. LISA generatorja samega z razširitvijo LISA specifikacijskega jezika ni bilo potrebno razširjati.

Algoritem 6.1 Glavni algoritem tkanja v orodju LISA

```
1:  method tkanje(jezik)
2:     Vhod: jezik // končni jezik – najnižje v hierararhiji jezikov
3:     Izhod: spremenjene podatkovne strukture
4:
5:     Definirani_Apply_Stavki ← najdi_apply(jezik);
6:     for all Definirani_Apply_Stavki do
7:        apply_st ← naslednji(Definirani_Apply_Stavki);
8:        // najdi množico vseh parov (advice, množica pointcut-ov)
9:        nasvet_pointcut ← najdi_pare(apply_st);
10:       for all nasvet_pointcut do
11:          nasvet ← naslednji(nasvet_pointcut).nasvet;
12:          pointcuts ← naslednji(nasvet_pointcut).pointcuts;
13:          // semantiko nasveta združimo s semantiko produkcij,
14:          // ki jih označuje množica specifikatorjev stičišč
15:          združi(nasvet, pointcuti, apply_st);
16:       end for
17:    end for
18: end method
```

Celoten proces tkanja lahko na konceptualnem nivoju opišemo z algoritmom 6.1, ki opisuje proces tkanja z vključenim dedovanjem na dokaj visokem abstraktnem nivoju in je podoben algoritmu 5.1, ki smo ga opisali v poglavju o teoriji aspektno usmerjenih atributnih gramatik. Tkanje se prične v končnem (ciljnem) jeziku ne glede na to kje v hiearhiji jezikov smo definirali *apply* stavek. Enak algoritem je uporabljen za vsak *apply* stavek, ki definira unikaten par *nasvet–specifikator stičišč*. Prvi korak tkanja je iskanje vseh nasvetov in specifikatorjev stičišč, kamor želimo dodati semantiko nasveta (algoritem 6.1, vrstica 9). Ta korak je identičen, kot iskanje identifikatorjev v vseh programskih jezikih z asimetričnim nasledniškim dedovanjem. Pri tem koraku moramo biti zelo pozorni, saj je lahko zaradi večkratnega dedovanja, ki ga podpira LISA, vsaka definicija sestavljena iz ene ali več definicij. Pri specifikatorjih stičišč lahko ena sama definicija definira več množic produkcij, ki so lahko parametrizirane z različnimi parametri. Le–to razrešujemo v kasnejših fazah tkanja. Ko najdemo vse pare, se enak algoritem (klic algoritma 6.2; algoritem 6.1, vrstica 15) izvrši nad vsakim najdenim parom.

Semantiko, definirano v nasvetu, je potrebno združiti s semantiko vsake produkcije, ki jo definira specifikator stičišč. Prvi korak je zamenjava dejanskih in formalnih parametrov nasveta. Nato je potrebno poiskati vse produkcije, ki so definirane s specifikatorjem stičišč. Iskanje se vrši v vseh jezikih,

Algoritem 6.2 Algoritem za združevanje semantičnih funkcij

```
1: method združi(nasvet, pointcuti, apply_st)
2:    Vhod: nasvet // podatkovna struktura nasveta
3:          pointcuti // množica specifikatorjev stičišč
4:          apply_st // definicija "apply" stavka
5:    Izhod: spremenjene podatkovne strukture
6:
7:    // zamenjava formalnih in dejanskih parametrov
8:    nasvet_new = zamenjaj_parametre(nasvet, apply_st);
9:    for all pointcuti do
10:      pointcut_st ← naslednji(pointcuts);
11:      // najdi vse produkcije, ki se ujemanjo s specifikatorjem stičišč
12:      produkcije ← najdi_productions(pointcut_st);
13:      for all produkcije do
14:         produkcija_st = naslednji(produkcije);
15:         // generiranje semantike
16:         sem_funkcije = generiraj_semantične_funkcije(nasvet, produk-
            cija_st);
17:         // združitev semantike nasveta in semantike produkcije
18:         vstavi(sem_funkcije, produkcija_st);
19:      end for
20:   end for
21: end method
```

začenši z jezikom najnižje v hierarhiji. Poiskati je potrebno vse produkcije, ki se ujemajo z vzorcem specifikatorja stičišč oz. vseh specifikatorjev stičišč v primeru, da je specifikator razširjen. Pri tem moramo seveda upoštevati dosegljivost posameznih pravil v katerih so produkcije definirane. Naslednji korak, generiranje novih semantičnih funkcij za vse najdene produkcije (algoritem 6.2, vrstice 13–19).

Generiranje semantike je v primeru enostavnih funkcij dokaj preprosto. semantična funkcija nasveta preprosto prepiše ter vstavi v željeno produkcijo. V primerih, ko v nasvetih uporabimo generične konstrukte za specificiranje semantike, je potrebno pred samo aplikacijo semantike nasvetov le-to zgenerirati. Rezervirana beseda **LHS** se nadomesti z neterminalom na levi strani produkcije. V primeru, ko je rezervirana beseda **RHS** definirana na levi strani semantične funkcije, se semantična funkcija generira za vsak neterminalni simbol na desni strani produkcije. Rezervirana beseda RHS lahko nastopa tudi na desni strani semantične funkcije. V tem primeru lahko nastopi

samo v kombinaciji z eno izmed funkcij za obdelavo seznama neterminalnih simbolov (*pred*, *succ* itd.). V kolikor funkcija vrne napako (npr. funkcija pred(RHS), za prvi element seznama ni definirana), se semantična funkcija ne generira. Ko imamo zgenerirane vse semantične funkcije nasveta, je potrebno vse formalne parametre nasvetov zamenjati z dejanskimi parametri *apply* stavkov. Ko razrešimo parametre, lahko semantiki produkcij pridružimo semantiko nasveta. Pri tem moramo paziti na podvajanje semantičnih funkcij. Če ima produkcija že definirano semantično funkcijo, ki jo želimo prenesti iz nasveta, imamo dve možnosti. V primeru, da je *apply* stavek definiran z rezervirano besedo **overwrite**, semantično funkcijo produkcije prepišemo. V nasprotnem primeru semantično funkcijo nasveta ignoriramo in je ne pridružimo semantiki produkcije.

Primer 6.5 prikazuje LISA specifikacije, ki jih iz specifikacij na primeru 6.3 zgenerira LISA tkalec.

6.3 Vzporednice z ortogonalnimi koncepti aspektno usmerjenih jezikov

V poglavju 3.3.1 smo govorili o preučevanju modelov pri načrtovanju novega aspektnega jezika. S podobnimi problemi se soočimo tudi pri razvoju domensko specifičnega aspektnega jezika. V nadaljevanju odgovorimo na zastavljena vprašanja iz poglavja 3.3.1.

- *Ali programski jezik razlikuje med "osnovnim" in "aspektnim" delom jezika?*

  Podpora aspektov je omogočena na nivoju jezika. Aspektov ni potrebno definirati v ločenih komponentah, čeprav je takšen način razvoja jezika zaradi večje dekompozicije specifikacij jezika zaželen.

- *Na katerih točkah v programu lahko vključimo obnašanje, definirano z aspekti?*

  Stične točke v LISA specifikacijah so mesta v specifikacijah atributnih gramatik, kjer lahko dodamo semantične funkcije. To so produkcije. Model stičišč je statičen.

- *Kakšne jezikovne konstrukte ponuja jezik za opis načina in mesta aplikacije aspektov?*

  LISA ima v jezik vključen poseben stavek (rezervirana beseda *apply*), s katerim definiramo aplikacijo nasveta.

Primer 6.5 Specifikacije programskega jezika `RobotTime`, brez aspektnih lastnosti LISA jezika

```
language RobotTime extends Robot {
  attributes int *.time;
  rule extends start {
 START ::= begin COMMANDS end compute {
   START.time = COMMANDS.time;
  };
 }
  rule extends commands {
   COMMANDS ::= COMMAND COMMANDS compute {
     COMMANDS[0].time = COMMAND.time + COMMANDS[1].time;
   }
   | epsilon compute {
     COMMANDS.time = 0;
   };
 }
  rule extends command {
   COMMAND ::= left compute {
     COMMAND.time = 1;
   };
   COMMAND ::= right compute {
     COMMAND.time = 1;
   };
   COMMAND ::= up compute {
     COMMAND.time = 1;
   };
   COMMAND ::= down compute {
     COMMAND.time = 1;
   };
 }
}
```

- *Kakšne mehanizme ima na voljo razvijalec za sistematično aplikacijo aspektov?*

  S specifikatorji stičišč definiramo množico stičišč, na katere lahko apliciramo nasvete.

- *Kakšna je jezikovna podpora za omejevanje vidnosti in efektov aspektov in njihove interakcije z ostalim programom?*

  LISA ne omogoča jezikovne podpore skrivanju aspektnih lastnosti. Za omejitev aspektnih lastnosti so na voljo mehanizmi večkratnega dedovanja, s pomočjo katerih je mogoče določene lastnosti razširiti, omejiti ali celo izbrisati.

- *Kakšne mehanizme ima na voljo jezik za preverjanje kompatibilnosti komponent, ki jih združujemo?*

  Preverjanje kompatibilnosti komponent je omejeno na natančno javljanje napak ob tkanju. Sama razrešitev le–teh je prepuščena razvijalcu.

- *Ali mora biti programski kod, na katerega želimo aplicirati aspekte, zapisan v posebni obliki, da je primeren za aplikacijo aspektov?*

  Ne.

- *Ali jezik omogoča razvoj programov za poljubno domeno?*

  Jezik je domensko specifičen.

- *Kakšna je možnost ponovne uporabe aspektov v različnih kontekstih?*

  Za ponovno uporabo aspektov so na voljo naslednji mehanizmi: večkratno dedovanje, parametrizacija nasvetov (neodvisnost od semantike programskega jezika) in generični zapis semantičnih funkcij v nasvetih (neodvisnost od sintakse programskega jezika).

- *Ali jezik pozna mehanizme za opis in razreševanje konfliktov, ki lahko nastanejo pri aplikaciji aspektov?*

  Ne. Konflikti se zaznajo in se vrnejo v obliki napak ob tkanju.

- *Je programski jezik namenjen razširitvam obstoječih jezikov in okolij razvitih v teh jezikih?*

  Jezik je razširitev že obstoječega jezika in ohranja kompatibilnost s prejšnjimi različicami.

- *Ali jezik nudi podporo za spremembo aspektov v času izvajanja?*

  Ne.

- *Ali jezik ponuja mehanizme za statično preverjanje izvornega koda?*

  Ne.

- *Kakšen nivo razhroščevanja in orodja za razhroščevanje, nam jezik omogoča?*

  Razhroščevanje je podprto na nivoju končnih specifikacij (po tkanju).

- *Ali je možno aspekte testirati ločeno od ostalega programa?*

  Ne.

- *Kako je jezik implementiran? Kako je jezik mogoče implementirati?*

  Jezik je nadgradja že obstoječega jezika.

Iz odgovorov je mogoče razbrati, da LISA specifikacijski jezik pokriva široki spekter funkcionalnosti aspektnih jezikov. Do razlik v določenih lastnostih prihaja zaradi specifičnosti zapisa atributnih gramatik.

6.4 Primer

V poglavju je prikazan primer implementacije programskega jezika, ki smo ga poimenovali *Tiny*. Jezik je načrtovan po načelih aspektno usmerjenega programiranja, kar pomeni, da smo posamezne dolžnosti jezika zapisali ločeno kot komponente, ki jih je mogoče uporabiti pri razvoju drugih jezikov.

Primera 6.6 in 6.7 prikazujeta implementacijo jezika aritmetičnih izrazov za realna števila[3]. V jeziku s primera 6.6 smo podprli zgolj operaciji seštevanja in odštevanja. Jezik smo nadalje razširili z operacijama množenja in deljenja, dodali pa smo tudi operator `oklepaj`, ki omogoča združevanje določenega dela izraza. Kot je razvidno iz primera 6.7, smo v tem jeziku specificirali le dodatne lastnosti jezika, večino lastnosti pa smo ponovno uporabili. Pomen programa po izvajanju je vrednost izraza. Npr. vrednosti programa `5 + 6 * (7 - 2.45) / 2` je `Z.val = 18.65`.

[3]Primera prikazujeta implementacijo primera 4.4 (poglavje 4.3.1) v orodju LISA. Atributna gramatika z dedovanjem je za primera opisana v poglavju 4.5.2 (stran 69).

Primer 6.6 Specifikacije programskega jezika `PlusMinus`

```
language PlusMinusExpr {
  lexicon {
  Number      [0-9]*.?[0-9]+
  Operator    \+ | \-
  ignore      [\ \0x0D\0x0A\0x09]+
  }
  attributes  double *.val;
  rule Start {
   Z ::= E  compute {
     Z.val = E.val;
   };
  }
  rule Expression {
   E ::= E \+ T  compute {
     E.val = E[1].val + T.val;
   };
   E ::= E \- T  compute {
     E.val = E[1].val - T.val;
   };
   E ::= T  compute {
     E.val = T.val;
   };
  }
  rule Term {
   T ::= #Number  compute {
     T.val= Double.parseDouble(#Number.value());
   };
  }
}
```

Naravna razširitev jezika aritmetičnih izrazov je vključitev spremenljivk in prireditvenega stavka v sam jezik. V ta namen je potrebno v sam jezik uvesti okolje, kjer hranimo spremenljivke ter povežemo logična imena z dejanskimi vrednostmi. Atribute okolja je potrebno propagirati čez celotno sintaksno drevo. Razvijalec jezika mora torej vsem produkcijam dodati nove semantične funkcije. Ker LISA podpira dedovanje atributnih gramatik, originalnih semantičnih funkcij ni potrebno spreminjati, potrebno je le definirati nove. Kljub očitni izboljšavi je s strani razvijalca vseeno potrebno precej truda za dodajanje nove semantike. Za dodajanje novih semantičnih funkcij je namreč potrebno razširiti obstoječa pravila ter navesti produkcije, ki jim želimo dodati semantične fukcije. Na primeru 6.8 je prikazana implementacija razširitve jezika aritmetičnih izrazov brez uporabljenih aspektnih lastnosti[4].

V primeru 6.10 smo koncepte AOP uporabili pri razširitvi semantike kakor tudi sintakse programskega jezika. V tem primeru ni potrebno redefinirati vseh produkcij, saj lahko semantiko produkcijam dodamo z nasveti in ustrezno definicijo stičnih točk. Prav tako je mogoče specifikacije zapi-

[4] V primeru zaradi obširnosti niso vključene vse produkcije

Primer 6.7 Specifikacije programskega jezika `MulDiv`

```
language MulDivExpr  extends PlusMinusExpr {
  lexicon {
   extends Operator   \* | \/
   Separator          \( | \)
  }
  rule  extends Start {
   Z ::= E  compute {
   };
  }
  rule  extends Expression {
   T ::= T \* F  compute {
     T.val = T[1].val * F.val;
   };
   T ::= T \/ F  compute {
     T.val = T[1].val / F.val;
   };
   T ::= F  compute {
     T.val = F.val;
   };
  }
  rule  overrides Term {
   F ::= #Number  compute {
     F.val= Double.parseDouble(#Number.value());
   };
   F ::= \( E \)  compute {
     F.val= E.val;
   };
  }
}
```

sati modularneje. V našem primeru smo del jezika, ki je neodvisen od vseh komponent, zapisali v ločeni komponenti (primer 6.9), ki jo lahko ponovno uporabimo pri razvoju drugih jezikov.

Na primeru 6.11 smo jeziku dodali koncept vhodno/izhodne naprave (I/O), ki jo simuliramo s seznamom vrednosti. Jeziku smo dodali dva stavka. Stavek `read` ovrednotimo z vrednostjo, ki na vrhu seznama `IN`, ki simulira vhodno napravo. Le–to definiramo kot podedovan atribut neterminalnega simbola STMTS (`inhIN`). Stavek `read` zapiše vrednost izraza na simulirano izhodno napravo, ki je predstavljena z atributom `synOUT`[5].

Nove lastnosti jezika so nedvomno dolžnosti, ki se prekrivajo z že obstoječimi, saj jezik ne podpira lastnosti kot so "propagacija vhodnih vrednosti" in "shranjevanje izhodnih vrednosti". Le–te morajo biti podprte na nivoju jezika, v kolikor želimo simulirati vhodno/izhodne naprave. Aspekta "propagacija vhodnih vrednosti" ni potrebno v celoti definirati, saj lahko ponovno uporabimo že definirane aspekte (primer 6.10). Ponovno definiramo zgolj

[5]Prikazani primer (primeri 6.9 – 6.11) smo povzeli po delu [105], kjer avtor navaja različne rešitve uporabe aspektno usmerjenega programiranja na deklarativnem nivoju. Med rešitvami navaja tudi možnost aspektno usmerjenega programiranja v atributnih gramatikah.

Primer 6.8 Specifikacije programskega jezika `Core` (implementacija brez aspektnih lastnosti LISA jezika)

```
import "Expressions.lisa";
language Core  extends MulDivExpr  {
  lexicon {
   extends Operator   \=
  Separator          \;
  Reserved          skip
  Identifier        [a-zA-Z_][a-zA-Z0-9_]*
  ignore            [\ \0x0D\0x0A\0x09]+
 }
 attributes java.util.Hashtable *.inhST;
           java.util.Hashtable *.synST;
            double *.val;
 rule Program {
  PROGRAM ::= STMTS  compute {
    STMTS.inhST =  new java.util.Hashtable();
    PROGRAM.synST = STMTS.synST;
  };
 }
 rule Statements {
  STMTS ::= STMT STMTS  compute {
    STMTS.synST = STMTS[1].synST;
    STMT.inhST = STMTS.inhST;
    STMTS[1].inhST = STMT.synST;
  } |  epsilon  compute {
    STMTS.synST = STMTS.inhST;
  };
 }
 rule Statement {
  STMT ::= #Identifier \= E \;  compute {
    STMT.synST = doAssign(STMT.inhST, #Identifier.value(), E.val);
    E.inhST = STMT.inhST;
  };
  STMT ::= skip \;  compute {
    STMT.synST = STMT.inhST;
  };
 }
 rule  extends Expression {
  E ::= E \+ T  compute {
    E.synST = T.synST;
    E[1].inhST = E.inhST;
    T.inhST = E[1].synST;
  };
  . . .
  T ::= F  compute {
    F.inhST = T.inhST;
    T.synST = F.synST;
  };
 }
 rule  extends Term {
  F ::= #Number  compute {
    F.synST = F.inhST;
  };
  F ::= \( E \)  compute {
    E.inhST = F.inhST;
    F.synST = E.synST;
  };
 }
  . . .
}
```

nove lastnosti jezika. Le–te dodamo v produkciji `Program` in `Term`. Za propagacijo atributov poskrbi že definiran nasvet `propagateST`. Razširiti je potrebno zgolj množico stičišč, kjer nasvet uporabimo. Aspekt "shranjevanje izhodnih vrednosti" je implementiran kot shranjevanje izhodnih vrednosti s pridobljenim atributom `synOUT`. Tudi za definicijo tega nasveta je potrebno definirati zgolj dodatno semantiko. Za propagiranje atributov poskrbi že definiran nasvet. Pri definiciji aspekta v podanem primeru se pokažejo prednosti ponovne uporabe aspektnih lastnosti in možnosti modularnega zapisa specifikacij, ki ga ponuja aspektno usmerjen pristop.

Primer 6.9 Specifikacije `abstraktnega` programskega jezika `Stmts`

```
language Stats {
 rule Statements {
  STMTS ::= STMT STMTS  compute {    } |  epsilon  compute { };
 }
}
```

Primer 6.10 Specifikacije programskega jezika `Core`

```
import "Expressions.lisa";
import "Statements.lisa";
language Core  extends MulDivExpr, Stats {
  lexicon {
   extends Operator   \=
   Separator         \;
   Reserved          skip
   Identifier        [a-zA-Z_][a-zA-Z0-9_]*
   ignore            [\ \0x0D\0x0A\0x09]+
  }
  attributes java.util.Hashtable *.inhST;
            java.util.Hashtable *.synST;
             double *.val;
  pointcut allProds *.Exp*  : * ::= .. ;
  pointcut  extends allProds  *.Stat* : * ::= .. ;
  pointcut  extends allProds  *.Te* : * ::= .. ;
  advice propagateST<inAtt, outAtt> {
   if (empty) {
      LHS.outAtt = LHS.inAtt;
  }
   else {
    first.inAtt = LHS.inAtt;
    RHS.inAtt = pred.outAtt;
    LHS.outAtt = last.outAtt;
   }
  }
  apply propagateST<inhST, synST>  on allProds;
  rule Program {
   PROGRAM ::= STMTS  compute {
     STMTS.inhST =  new java.util.Hashtable();
     PROGRAM.synST = STMTS.synST;
   };
  }
  rule Statement {
   STMT ::= #Identifier \= E \;  compute {
     STMT.synST = (java.util.Hashtable)doAssign(E.synST, #Identifier.value(), E.val).clone();
   };
   STMT ::= skip \;  compute { };
  }
  rule  extends Term {
   F ::= #Identifier  compute {
    F.val = ((Double)F.inhST.get(#Identifier.value())).doubleValue();
   };
  }
  method setVariable {
    public java.util.Hashtable doAssign(java.util.Hashtable ST, String  var,  double val) {
     ST.put( var,  new Double(val));
      return ST;
   }
  }
}
```

Primer 6.11 Specifikacije programskega jezika `CoreInOut`

```
import "AspectCore.lisa";
language CoreInOut  extends Core {
 lexicon {
   extends Reserved   read | write
   extends Separator  \( | \)
 }
 attributes java.util.Vector *.inhIN; java.util.Vector *.synIN;
 attributes java.util.Vector *.synOUT;
 pointcut pStmt *.Statement : * ::= .. ;
 advice initOUT<attr, val> { LHS.attr = val;  }
 apply initOUT<synOUT,  null>  on pStmt;
 apply propagateST<inhIN, synIN>  on allProds;

 rule  extends Program {
  PROGRAM ::= STMTS  compute {
    PROGRAM.synOUT = STMTS.synOUT;
    STMTS.inhIN = input();
  };
 }
 rule  extends Statements {
  STMTS ::= STMT STMTS  compute {
    STMTS.synOUT = doConcat(STMTS[1].synOUT, STMT.synOUT);
  } |  epsilon  compute {
    STMTS.synOUT =  new java.util.Vector();
  };
 }
 rule  extends Statement {
  STMT ::= write \( E \) \;  compute {
    STMT.synOUT = doWrite(E.val);
   };
 }
 rule  extends Term {
  F ::= read  compute {
   F.val = evalRead(F.inhIN);
  };
 }
 method creteInput {
  java.util.Vector input() {
    java.util.Vector in =  new java.util.Vector();
    in.add( new Double(7.666));
    in.add( new Double(10.666));
     return in;
  }
 }
 method EvalRead {
   double evalRead(java.util.Vector in) {
      return ((Double)in.remove(0)).doubleValue();
  }
 }
 method DoWrite {
  java.util.Vector doWrite( double val) {
     java.util.Vector in =  new java.util.Vector();
     in.add( new Double(val));
      return in;
   }
  java.util.Vector doConcat(java.util.Vector a, java.util.Vector b) {
      if (b !=  null)
       a.addAll(0, b);
      return a;
   }
 }
```

V nadaljevanju si poglejmo še dve razširitvi jezika, ki pa sta vsebinsko precej različni. V prvem primeru jezik nadgradimo s semantičnimi in sintaltičnimi konstrukti, ki jih zapišemo v povsem ločeno komponento v obliki aspekta. Razširitev se nujno ne prekriva z ostalimi koncepti jezika, a primer prikazuje smotrnost načrtovanja jezika z uporabo aspektov, saj so specifikacije na ta način precej manj obširne in precej bolj berljive. Kar pa je morda še pomembnejše, da imamo vsak koncept jezika zapisan v ločeni komponenti. V drugem primeru bomo poskušali jeziku vgraditi profilirnik, ki šteje število izvedenih stavkov programa. V našem primeru štejemo število zapisanih stavkov `write`. Na podoben način bi lahko implementirali tudi druga orodja, kot so razhroščevalnik, generator sporočil o napakah v programu, generator izvornega koda itd. V kolikor želimo v štetje vključiti še kakšen drugi stavek, preprosto razširimo specifikator stičišč ali celoten aspekt.

Primer 6.12 Specifikacije `abstraktnega` programskega jezika `StmtCounter`

```
language WriteCounter {
 attributes  int *.inCount;  int *.outCount;
 pointcut pCount *.* :STMT ::= write .. ;
 advice counter<in, out, val> {
  LHS.out = LHS.in + val ;
 }
 apply counter<inCount, outCount, 1>  on pCount;
 apply propagateST<inCount, outCount>  on allProds;
}
```

Implementacija profilirnika je prikazana na primeru 6.12. Za implementacijo profilirnika smo zapisali nov nasvet, ki v ustrezno produkcijo doda semantično funkcijo za povečevanje števca. Nasvet za propagacijo atributov ponovno uporabimo iz že definiranih jezikov.

Na podoben način kot neodvisno komponento, smo zapisali koncept vgnezdene bločne strukture. Jezik razširja jezik `CoreInOut` (primer 6.11) ter mu dodaja sintaksne konstrukte za zapis bločne strukture (rezervirani besedi `begin` in `end`) ter semantične akcije za dostop do spremenljivk. Znotraj bloka so dostopne vse spremenljivke, ki so bile definirane pred in v bloku. S to razliko, da se spremenljivke definirane izven bloka v blok prenesejo po vrednosti. Spremembe vrednosti spremenljivk se torej ne prenesejo izven bločne strukture. Implementacija aspekta za koncept bločne strukture je prikazana na primeru 6.13.

Celotne specifikacije združuje jezik `Tiny`, ki je prikazan na primeru 6.14. Jezik razširja jezika `Block` in `WriteCounter` ter dodaja sintaksni konstrukt za definicijo programa. Jezik ne vpeljuje novih semantičnih konstruktov.

Primer 6.15 prikazuje primer programa v programskem jeziku `Tiny`. V programu je v različnih blokih uporabljena spremenljivka `id`. V vgnezdenem bloku lahko beremo vrednost spremenljivke, spremeiti pa je ne moremo.

Primer 6.13 Specifikacije `abstraktnega` programskega jezika `Block`

```
import "AspectCoreInOut.lisa";
language Block  extends CoreInOut {
  lexicon {
   BReserved   begin | end
  }
  pointcut  extends pStmt *.BlockStatement : * ::= .. ;
  rule Block {
   BLOCK ::= begin STMTS end \;  compute {
     BLOCK.synOUT = STMTS.synOUT;
    };
   }
  rule BlockStatement {
  STMT ::= BLOCK  compute {
     STMT.synOUT = BLOCK.synOUT;
     STMT.synST = STMT.inhST;'
     BLOCK.inhST = (java.util.Hashtable)STMT.inhST.clone();
    };
   }
}
```

Slednje je razvidno iz seznama, ki predstavlja izhodno napravo, saj smo v vsakem bloku izpisali vrednost spremenljivke. Pomen programa po interpretaciji je naslednji: `Tiny.synST = {id = 10}`, `Tiny.synOUT = {10, 20, 33, 20}` in `Tiny.outCount = 4`.

Primer 6.14 Specifikacije programskega jezika `Tiny`

```
import "Block.lisa";
import "StmtCounter.lisa";
language Tiny  extends Block, StmtCounter {
 lexicon {
   extends Reserved   program
   extends Separator  \.
 }
 pointcut  extends allProds Block.* : * ::= .. ;
  rule Tiny {
   START ::= program #Identifier \; begin PROGRAM end \.  compute {
      START.outCount = PROGRAM.outCount;
      START.synOUT = PROGRAM.synOUT;
      START.synST = PROGRAM.synST;
   };
  }
 rule  extends Program {
  PROGRAM ::= STMTS  compute {
    PROGRAM.outCount = STMTS.outCount;
  };
 }
```

Primer 6.15 Program v programskem jeziku `Tiny`

```
program Test;
begin
 id = 10;
 write(id);
 begin
   id = id + 10;
   write(id);
   begin
     id = id + 13;
      write(id);
   end;
   write(id);
 end;
end.
```

Many of the things you can count, don't count. Many of the things you can't count, really count.

– Albert Einstein –

Poglavje 7

Rezultati in analiza

V tem poglavju bomo predstavili rezultate disertacije. Ker smo se ukvarjali s specifikacijami programskih jezikov, jih je potrebno analitično ovrednotiti in primerjati s primerljivimi že uveljavljenimi pristopi. Specifikacije (aspektno usmerjene specifikacije atributnih gramatik) smo ovrednotili z najpogostejšimi metrikami za gramatike, ki smo jih v ta namen priredili za LISA specifikacije. Orodje LISA smo v ta namen nadgradili z možnostjo izračuna določenih metrik na nivoju specifikacijskega jezika, kakor tudi monolitne oblike specifikacij atributne gramatike.

7.1 Uvod

V prejšnjih poglavjih smo se seznanili z načrtovanjem in implementacijo, predvsem domensko specifičnih, programskih jezikov. Spoznali smo formalne pristope, ki nam olajšajo razvoj novih jezikov. Prav tako pa formalni pristopi prinašajo pasti, na katere moramo biti še posebej pozorni. V delu smo se osredotočili predvsem na atributne gramatike, ki zraven očitnih prednosti in izboljšav, napram ostalim formalnim pristopom, prinašajo tudi določene

slabosti. Te slabosti smo poskušali v disertaciji odpraviti. Pri izboljšavah smo se osredotočili predvsem na modularni in inkrementalni razvoj programskih jezikov, saj se zavedamo, da predstavljata vzdrževanje in posodobitev programske opreme kar 60% do 80% celotnih stroškov razvoja [106]. Podobno bi lahko ugotovili tudi za razvoj programskih jezikov. Da bi zmanjšali stroške razvoja, je potrebno razvijalcem omogočiti mehanizme za modularen, inkrementalen razvoj s poudarkom na ponovni uporabi. Eden izmed ključnih pojmov za dosego tega cilja je kompleksnost pristopa (programskega jezika, specifikacij, orodij itd.). Eden izmed prostopov za merjenje kompleksnosti in upravljanje programov, predvsem skozi njegovo evolucijo, je uporaba metrik. Uporaba metrik je tako postala eden izmed standardov v dobri praksi razvoja programskih sistemov. Merjenje karakteristik programa je potrebno za ugotavljanje konsistentnosti zahtev, pravilnosti načrtovanja, kompleksnosti koda itd. Metrike za progamski kod so bile uspešno prenesene tudi na področje gramatik [107]. Nekatere od teh metrik, ki smo jih priredili našemu specifikacijskemu jeziku, smo uporabili tudi za meritve uspešnosti našega pristopa.

7.2 Uporabljene metode ocenitve rezultatov

Za meritev kompleksnosti specifikacij programskih jezikov v razvitem aspektno usmerjenem specifikacijskem jeziku smo uporabili 6 metrik, ki smo jih povzeli po avtorjih Power in Malloy [107]. Te metrike smo uporabili na specifikacijah atributnih gramatik kot tudi v našem jeziku. Za ta namen smo morali metrike malce dopolniti ter jih vključiti v orodje LISA. V že omenjenem članku so opisane zgolj metrike na sintaksnem nivoju gramatik, v našem delu pa se ukvarjamo predvsem s semantiko programskih jezikov. V ta namen in zaradi specifičnosti LISA specifikacijskega jezika smo tem metrikam dodali še metrike za semantično kompleksnost specifikacij. V grobem se metrike delijo na metrike velikosti in metrike strukturiranosti. S tem da sta ti dve lastnosti v gramatikah zelo podobni, saj lahko iz velikosti zapisa določenega koncepta programskega jezika zelo uspešno sklepamo o kompleksnosti specifikacij. Sledi opis posameznih metrik, ki smo jih uporabili pri naši analizi. Za vsako metriko opišemo definicijo za gramatike (v našem primeru monolitne specifikacije atributnih gramatik) in njeno uporabo na LISA specifikacijah. Cilj analize LISA specifikacij z metrikami je ocena kompleksnosti gramatike, ki jo iz teh specifikacij generiramo. Iz tega lahko sklepamo o naporu razvijalcev za zapis specifikacij programskega jezika. Zraven naštetih metrik smo za vsak primer analizirali tudi strukturo celotnih specifikacij (dedovanje) ter razčlenjenost lastnosti, ki so predmet ponovne uporabe. Teh ugotovitev ne

navajamo analitično (številčno), temveč jih skupaj s posameznimi metrikami analiziramo in opišemo za vsak podan primer posebej.

Efektivno število vrstic specifikacij (eLOC)

Najpreprostejša metrika za merjenje velikosti programov in tudi gramatik je merjenje števila vrstic (LOC – *Lines Of Code*), ki so potrebne za zapis določene funkcionalnosti. Ker so stili zapisa med posameznimi razvijalci različni (nekateri pišejo več stavkov v isto vrstico, drugi vsak stavek in oklepaje pišejo v svojo vrstico ipd.), zgolj štetje vrstic ne odraža realnega stanja. V ta namen se iz zapisa (programa, specifikacij) odstranijo vrstice, ki ne doprinesejo k samemu zapisu funkcionalnosti. Te vrstice so: komentarji, prazne vrstice, vrstice z oklepaji, ki označujejo bloke itd. To metriko potem označimo z eLOC (*Efective Lines Of Code*). Pri monolitnih specifikacijah atributnih gramatik merimo vsako zapisano vrstico (pri štetju izpustimo zgolj komentarje). V LISA specifikacijskem jeziku pa je mogoče kod strukturirati ter zapisati na različne načine, tako da pri izračunu eLOC metrike izpustimo vse prazne vrstice, vrstice s komentarji ter vrstice, v katerih se pojavlja samo leksikalni simbol za ločevanje blokov ('{' in '}') ali označevanje alternativ ('|').

Število terminalnih (TERM) in neterminalnih (VAR) simbolov

Ekvivalentno merjenju števila procedur (podprogramov) v programskih jezikih je merjenje števila neterminalnih simbolov v gramatikah. Število neterminalnih simbolov v gramatiki označimo z VAR. To metriko vrne velika večina sistemov za avtomatsko generiranje razpoznavalnikov/prevajalnikov ob generiranju razpoznavalnika/prevajalnika. Podobna metrika je število terminalnih simbolov, ki jih označimo s TERM. Čeprav sta metriki dokaj preprosti, veliko povesta o kompleksnosti same gramatike. Večje število neterminalnih in terminalnih simbolov pomeni kompleksnejšo gramatiko in posledično težje vzdrževanje. Prav tako lahko veliko število neterminalnih simbolov pomeni zapleteno sintakso jezika, veliko terminalnih pa veliko sintaktičnih dodatkov (*syntactic sugar*), ki lahko negativno vplivajo na učenje jezika s strani novih razvijalcev. Meritev te metrike se za LISA specifikacije in za monolitne specifikacije atributnih gramatik izvede enako.

Število produkcij kontekstno neodvisne gramatike (PROD)

Pri tej metriki štejemo število produkcij kontekstno neodvisne gramatike. Metrika skupaj s prej omenjenima metrikama TERM in VAR daje poglobljeno sliko kompleksnosti gramatike. Pri LISA specifikacijah je lahko to

število bistveno večje kot pri monolitnih specifikacijah atributnih gramatik, saj pri izračunu upoštevamo vse produkcije, izpeljane in prepisane; nedosegljive v končnih specifikacijah. Vse produkcije je potrebno upoštevati zato, ker je potrebno kompleksnost meriti za vse razvite specifikacije, tudi tiste, ki so se z inkrementalnim razvojem razširile, prepisale, spremenile itd. Pri monolitni obliki specifikacij atributne gramatike je to število enako številu neterminalov.

Povprečna velikost produkcije (AVS)

Metrika AVS predstavlja povprečno velikost desne strani produkcij kontekstno neodvisne gramatike. Izračunamo jo tako, da izračunamo vsoto števila vseh simbolov na desni strani produkcij (terminalni in neterminalni simboli) in jo delimo s številom neterminalov celotne gramatike. Na ta način lahko sklepamo o zahtevnosti razpoznavanja takšne gramatike ter kompleksnosti implementacije gramatike. Pri LISA specifikacijah smo metriko AVS malce priredili. Osnovna metrika, kot sta jo definirala avtorja v [107], je razvita za gramatike zapisane v notaciji EBNF, kjer je število produkcij enako številu neterminalnih simbolov. Za izračun AVS smo v LISA specifikacijskem jeziku število vseh simbolov na desni strani produkcij delili s številom vseh produkcij. Pri izračunu števila produkcij smo upoštevali vse produkcije, ki so vsebovane v specifikacijah.

McCabe ciklomatična kompleksnost (MCC)

McCabe ciklomatično kompleksnost v gramatikah merimo tako, da preštejemo število vseh alternativ v gramatikah. Kompleksnost gramatike z enakim številom terminalnih in neterminalnih simbolov in z različnim številom alternativ je lahko precej različna. V LISA specifikacijah pričakujemo povečanje tega števila napram monolitnim specifikacijam. Z uvedbo novih konceptov v LISA specifikacije (šablone, aspekti) in inkrementalnim načrtovanjem (dedovanje) programskega jezika pričakujemo zmanjševanje MCC števila. Z metriko v LISA specifikacijah ugotavljamo predvsem kompleksnost specifikacij v fazi testiranja, saj večje število alternativ pomeni bistveno več preverjanj.

Število semantičnih prireditev (SEMFUN)

Pri tej metriki merimo celotno število semantičnih funkcij v specifikaciji programskega jezika. V LISA specifikacijah je to število vseh semantičnih funkcij, tudi tistih, ki v končnih specifikacijah niso vključene. Metrika ponazarja kompleksnost pri načrtovanju in implementiranju semantike programskega

jezika. Z večanjem modularnosti in z uvedbo konceptov aspektno usmerjenega programiranja v LISA specifikacije pričakujemo bistveno zmanjšanje vrednosti te metrike. Z aspektnim pristopom bistveno zmanjšamo večkraten zapis istih semantičnih funkcij, ki je eden izmed vzrokov napak v specifikacijah in v kombinaciji z metrikami MCC, AVS in PROD predstavlja velik delež kompleksnosti razvoja novega programskega jezika.

Merjenje kompleksnosti aspektnih lastnosti LISA specifikacij

Čeprav z uvedbo novega koncepta pričakujemo zmanjševanje kompleksnosti specifikacij, sam zapis aspektov doprinese h kompleksnosti celotnih specifikacij. V kolikor želimo ustrezno primerjavo med posameznimi pristopi, teh lastnosti ne moremo ovrednotiti ločeno. Potrebno je najti vzporednice med neaspektnimi in aspektnimi lastnostmi LISA specifikacij. Pri implementaciji več prototipnih jezikov in pri meritvah zgoraj opisanih metrik smo izkustveno prišli do naslednjih ugotovitev:

- Kompleksnost definicije specifikatorja stičišč je enak kompleksnosti za definicijo produkcije.

  Pri definiranju specifikatorja stičišč je potrebno zelo dobro poznati strukturo produkcij, torej sintakso razvijajočega programskega jezika. Prav tako je potrebno zelo dobro poznati strukturo programskega jezika ob definiranju nove oz. redefiniciji produkcije.

- Kopleksnost definicije semantične funkcije v nasvetu je enak kompleksnosti za definicijo semantične funkcije v produkciji.

  Čeprav se v nasvetih definirajo generične semantične funkcije, njihov zapis ne poveča kompleksnosti celotnih specifikacij.

- Kompleksnost definicije aplikacije nasvetov je enak kompleksnosti definicije produkcije.

  Pri aplikaciji nasvetov je potrebno zelo dobro poznati strukturo produkcij, za katere želimo uporabiti določen nasvet. Prav tako je potrebno poznati semantiko, a ker se aplikacija nasveta močno navezuje na definicijo stičišč, menimo da je kompleksnost definicije obojega primerljiva.

Te metrike ne merimo ločeno od ostalih, ampak vrednosti prištejemo vrednostim ustreznih metrik. Da bi zagotovili verodostojnost že opisanih metrik, smo meritve izvajali ločeno, najprej samo za naštete metrike, potem pa smo tem vrednostim dodali še aspektni del.

7.3 Postavitev eksperimenta

Orodje LISA na vhodu prejme specifikacije programskega jezika zapisane v posebnem aspektno usmerjenem domensko specifičnem programskem jeziku, ki temelji na formalnem modelu atributnih gramatik. Specifikacijski jezik, kot smo natančneje opisali že v prejšnjem poglavju, omogoča več različnih načinov modularizacije ter konceptov za abstrahiranje sintakse in semantike programskega jezika. Osnovna modulacijska enota je jezik, ki abstrahira končne ali abstraktne (iz teh ni mogoče generirati prevajalnika za programski jezik) specifikacije določenega programskega jezika ali implementira zgolj določen koncept ipd. V posameznem jeziku lahko v logične enote združujemo eno ali več sintaksnih produkcij s semantiko, metode semantičnih domen, nasvete, šablone itd. Module lahko združujemo z mehanizmom večkratnega dedovanja, kjer so predmet dedovanja prej omenjene lastnosti jezika. Iz opisanega je razvidno, da se lahko razvijalec loti razvoja programskega jezika na precej načinov. Najenostavnejši od teh je prepis monolitnih specifikacij v LISA specifikacijski jezik. Takšen zapis ne omogoča visokega nivoja ponovne uporabe in ne izboljša bistveno berljivosti monolitnega zapisa atributnih gramatik. Seveda pa se lahko pri razvoju poslužimo vseh mehanizmov, kot so šablone v atributnih gramatikah, aspektov in seveda večkratnega dedovanja.

Ugotovljeno je bilo, da prepis monolitnih specifikacij v LISA specifikacije ne prinaša občutnejšega izboljšanja. Zato smo takšen zapis izpustili pri meritvah. Vsi prototipni jeziki, ki smo jih vključili v analizo, so razviti kar se da modularno (večkratno dedovanje aspektno usmerjenih atributnih gramatik). Primerjavo smo izvajali na več prototipnih jezikih, s tem da smo se meritev lotili postopoma. Meritve smo izvajali na vseh komponentah od osnovne do končne. Vse komponente seveda ne vsebujejo vseh konceptov, ki jih omogoča LISA, zato smo te meritve pri teh komponentah izpustili. Meritve smo izvedli na štirih prototipnih programskih jezikih, ki smo jih razvili z različnimi pristopi. Te pristope smo med seboj primerjali z upoštevanjem vseh opisanih metrik. Meritve smo izvajali na naslednjih pristopih:

- LISA.

  Programski jezik oz. komponento smo razvili z LISA specifikacijskim jezikom. V rešitev je vključeno modularno načrtovanje jezika z večkratnim dedovanjem. Analiza te rešitve nam služi kot referenca ostalim pristopom, zato ga označimo s 100%. Če so ostali pristopi označeni z višjim odstotkom, to pomeni, da je določena metrika višja in obratno.

- Šablone (šablone v atributnih gramatikah).

Primer smo implementirali z uporabo šablon v atributnih gramatikah.

- Aspekti.

  Primer smo implementirali z uporabo konceptov aspektno usmerjenega programiranja v atributnih gramatikah.

- Aspekti+.

  Analizirali smo enak primer kot v prejšnjem primeru, s tem da smo pri metrikah upoštevali še kompleksnost definicije aspektnih lastnosti LISA specifikacij.

Na koncu podajamo sumarno tabelo, v kateri je razvidna povprečna izboljšava pri vseh primerih za pristopa Aspekt in Aspekt+.

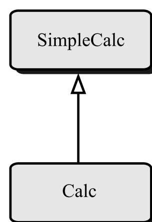
7.4 Meritve in analiza

V tem podpoglavju prikazujemo predstavitve primerov in rezultatov meritev metrik za štiri razvite prototipne jezike. Pri vseh primerih je uporabljena metodologija, ki smo jo predstavili že v prejšnjih sekcijah.

Primer 1: jezik Calc

Prvi primer prikazuje implementacijo preprostega jezika za izračun aritmetičnih izrazov (jezik `SimpleCalc`), ki ga v izpeljanem jeziku (jezik `Calc`) razširimo z uvedbo spremenljivk in definicijo prireditvenega stavka. Definirani spremenljivki lahko priredimo vrednosti aritmetičnega izraza, v katerem lahko uporabimo predhodno definirane spremenljivke. V prvem jeziku je pomen programa vrednost aritmetičnega izraza (npr. pomen programa `2 * (3 + 2) / 2` je `5`). V drugem jeziku pa je pomen jezika vrednosti spremenljivk po končanem izvajanju programa (npr. pomen programa `[ first = 10 ] [ second = first + 10]` je `first = 10, second = 20`). Hierarhija jezikov je predstavljena na sliki 7.1, rezultate analize z metrikami pa prikazuje tabela 7.1.

Specifikacije jezika `SimpleCalc` so zapisane z abstraktno sintakso, zato je število neterminalnih simbolov relativno nizko, prav tako pa število vrstic ni drastično večje od monolitnega zapisa atributnih gramatik. Enake vrednosti za metrike VAR, TERM, PROD pomenijo, da smo ohranili enako strukturo gramatike. Enake vrednosti metrik MCC in SEMFUN pa nakazujejo zgolj prepis osnovnih monolitnih specifikacij v LISA specifikacije. Specifikacije



Slika 7.1: Hieararhija dedovanja jezikov za jezik `Calc`.

tega jezika ne vsebujejo ostalih konceptov, ki jih podpira LISA, zato drugih meritev ni bilo mogoče izvesti.

V izpeljanem jeziku (`Calc`) smo uvedli spremenljivke in prireditveni stavek. V ta namen je bilo potrebno podpreti koncept okolja (vsaka spremenljivka ima svojo vrednost). S standardnim pristopom je v ta namen potrebno definirati nove produkcije za uvedbo prireditvenega stavka in redefinirati (izpeljati) vse produkcije ter jim dodati semantične funkcije za propagacijo okolja. Kot je razvidno iz rezultatov, se metrika SEMFUN, napram jeziku `SimpleCalc`, drastično poveča. Pri implementaciji s šablonami se spremenita zgolj metriki eLOC (92.77%) in SEMFUN (68.75%). Šablone omogočajo ponovno uporabo generičnih semantičnih specifikacij, a je potrebno vsako šablono posebej instancirati, kar v praksi pomeni, da je potrebno izpeljati sintaksno pravilo, v kateri je klic šablone. Razvijalcu torej ne zmanjša truda pri razširjanju jezika, po našem mnenju pa bistvenu pripomore k ponovni uporabnosti in zmanjševanju napak v fazi razvoja novih jezikov. Z uvedbo aspektov se izognemo redefiniranju sintaksnih pravil, le–tem dodamo zgolj novo sintakso. Podobno kot pri šablonah, vzorec *bucketBrigadeLeft* implementiramo z aspektom (v prejšnjem primeru s šablono) ter uporabimo na vseh produkcijah v hierarhiji jezikov. S tem pristopom smo dosegli bistveno zmanjšanje kompleksnosti specifikacij, kljub temu da je metrika eLOC od prejšnjega primera padla le za 18.07 odstotne točke. Izboljšavo vidimo pri metrikah PROD, AVS, MCC in predvsem SEMFUN, ki je od osnovnih LISA specifikacij padla za 68.75% (31.25% osnovnih LISA specifikacij). Zadnja meritev `Aspekt+`, kjer smo pri izračunu metrik upoštevili tudi kompleksnost definicije aspektnih lastnosti, daje realnejšo sliko o metriki SEMFUN (zvišana za 12.5 odstotne točke). Kljub temu opažamo, da je kompleksnost razvoja nasvetov manjša od kompleksnosti ponovne redefinicije vseh sintaksnih pravil in produkcij.

Primer je dokaj preprost (število terminalnih, neterminalnih simbolov in produkcij je relativno majhno), a smo glede na majhnost primera dobili obetajoče rezultate, ki potrjujejo našo tezo o primernosti aspektno usmerjenih

Tabela 7.1: Rezultati meritev metrik za primer `Calc`.

SimpleCalc.lisa							
Spec.	eLOC	VAR	TERM	PROD	AVS	MCC	SEMFUN
LISA	24 (100%)	2 (100%)	7 (100%)	7 (100%)	2.42 (100%)	5 (100%)	7 (100%)

Calc.lisa							
Spec.	eLOC	VAR	TERM	PROD	AVS	MCC	SEMFUN
LISA	83 (100%)	4 (100%)	12 (100%)	18 (100%)	2.38 (100%)	14 (100%)	32 (100%)
Aspekti	52 (62.65%)	4 (100%)	12 (100%)	12 (66.66%)	2.25 (94.18%)	8 (57.14%)	10 (31.25 %)
Aspekti+	52 (62.65%)	4 (100%)	12 (100%)	14 (77.77%)	1.92 (80.73%)	8 (57.14%)	16 (50.00 %)

specifikacijah programskih jezikov.

Primer 2: jezik RobotCalculator

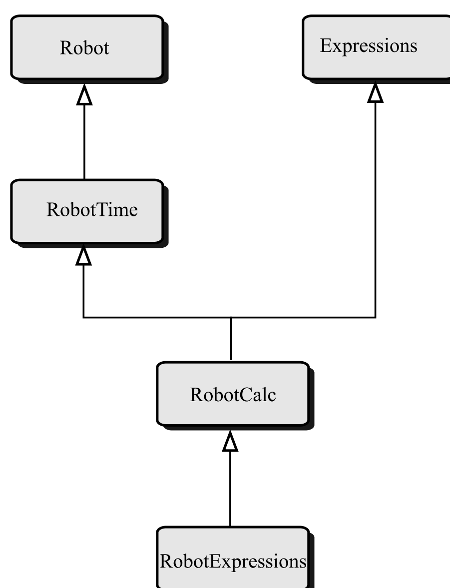
V tem primeru smo implementirali jezik za kontrolo robota, ki smo mu vgradili možnost pomnenja določenih operacij v obliki spremenljivk in merjenja časa do prispetja na cilj. Natančneje smo primer opisali že v poglavju 6.2, zato se v podrobno razlago primera na tem mestu ne bomo podajali. Pomen programa

```
begin
  left ID = 10*(5+2)
  right (1 + ID)
  up ID - 1
end
```

je končni položaj robota in čas, ki ga je potreboval do cilja (`time=3, x=1, y=69`). Seveda smo se razvoja celotnega jezika lotili na inkrementalni način, in kar se da modularno. Hieararhijo dedovanja jezikov je prikazana na sliki 7.2.

V prvem jeziku (Robot) smo implementirali osnovne ukaze robota (`up, down, left, right`). Pomen takšnega programa je končna lokacija robota. Jeziku `RobotTime` smo dodali koncept merjenja časa. Za vsak premik robot porabi enoto časa. V ta namen je potrebno redefinirati večino sintaksnih pravil ter produkcijam dodati enako semantično funkcijo. Le to smo definirali z aspektom, kar rezultira k bistvenemu zmanjšanju (23.53% oz. 17.65%) definiranih semantičnih funkcij. Primerno temu se zmanjšata tudi metriki MCC in PROD, kar dodatno potrjuje zmanjšanje kompleksnosti specifikacij.

Jezik `RobotCalc` je izpeljan iz jezikov `RobotTime` in `Expressions` (jezik aritmetičnih izrazov, podoben jeziku `SimpleCalc` iz podpoglavja 7.4). Jezik vključuje koncept aritmetičnih izrazov. Premik robota torej ni več konstanten, ampak se izračuna z aritmetičnim izrazom. V tem primeru je potrebno



Slika 7.2: Hieararhija dedovanja jezikov za jezik `RobotCalculator`.

ponovno redefinirati vsa sintaksna pravila, s katerim brišemo stare produkcije in jih nadomestimo z novimi (sintaksa jezika se spremeni). Ker smo v starševskem jeziku predvideli možnost razširitev, v katerih želimo ohraniti koncept merjenja časa, smo v jeziku `RobotTime` definirali specifikator stičišč, ki označuje tudi nove produkcije. Na ta način smo se izognili ponovnemu definiranju že definiranih sintaksnih pravil za ohranjanje konceptov ob spremembi sintakse jezika. Slednje z ostalimi koncepti ni mogoče.

V zadnjem jeziku v hierarhiji (`RobotExpressions`) smo jeziku dodali koncept spremenljivke in prireditvenega stavka. Na ta način si lahko robot zapomni prejšnjo lokacijo. Pri dodajanju tega koncepta, ki v osnovnih LISA specifikacijah zahteva izpeljavo vseh sintaksnih pravil in dopolnitev semantičnih funkcij v vseh produkcijah, se izkaže aspektni pristop za zelo primernega. To dejstvo prikazujejo tudi rezultati, ki so bistveno boljši od pristopa s šabloni (v povprečju za 49,64 odstotne točke za meritev Aspekt+) ter osnovnih LISA specifikacij. Predvsem je pomemben podatek, da so razlike največje pri metrikah PROD, AVS, MCC in SEMFUN, ki po našem mnenju povedo največ o kompleksnosti LISA specifikacij. Podrobnejša analiza meritev je prikazana v tabeli 7.2.

Tabela 7.2: Rezultati meritev metrik za primer `RobotCalculator`.

Expressions.lisa							
Spec.	eLOC	VAR	TERM	PROD	AVS	MCC	SEMFUN
LISA	23 (100%)	2 (100%)	7 (100%)	7 (100%)	2.42 (100%)	5 (100%)	7 (100%)
Robot.lisa							
Spec.	eLOC	VAR	TERM	PROD	AVS	MCC	SEMFUN
LISA	26 (100%)	3 (100%)	7 (100%)	7 (100%)	1.42 (100%)	4 (100%)	10 (100%)
RobotTime.lisa							
Spec.	eLOC	VAR	TERM	PROD	AVS	MCC	SEMFUN
LISA	45 (100%)	3 (100%)	7 (100%)	14 (100%)	1.42 (100%)	11 (100%)	17 (100%)
Aspekti	41 (91.11%)	3 (100%)	7 (100%)	10 (71.43%)	1.6 (112.0%)	7 (63.67%)	13 (76.47%)
Aspekti+	41 (91.11%)	3 (100%)	7 (100%)	13 (92.86%)	1.23 (86.15%)	7 (63.67%)	14 (82.35%)
RobotCalc.lisa							
Spec.	eLOC	VAR	TERM	PROD	AVS	MCC	SEMFUN
LISA	87 (100%)	5 (100%)	14 (100%)	26 (100%)	1.85 (100%)	21 (100%)	32 (100%)
Aspekti	79 (90.80%)	5 (100%)	14 (100%)	22 (84.62%)	2.0 (108.33%)	17 (80.95%)	24 (75%)
Aspekti+	79 (90.80%)	5 (100%)	14 (100%)	25 (96.15%)	1.76 (95.33%)	17 (80.955%)	25 (78.13%)
RobotExpressions.lisa							
Spec.	eLOC	VAR	TERM	PROD	AVS	MCC	SEMFUN
LISA	157 (100%)	6 (100%)	16 (100%)	43 (100%)	1.95 (100%)	37 (100%)	72 (100%)
Aspekti	65 (41.40%)	6 (100%)	16 (100%)	27 (62.79%)	1.96 (100.48%)	21 (56.75%)	27 (37.5%)
Aspekti+	65 (41.40%)	6 (100%)	16 (100%)	36 (83.72%)	1.47 (75.36%)	21 (56.76%)	33 (45.83%)

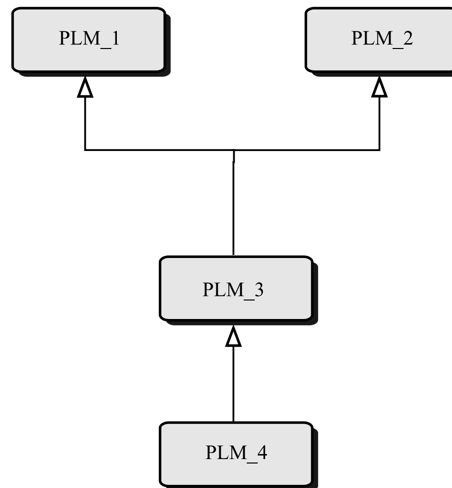
Primer 3: jezik PLM

V tej sekciji si poglejmo implementacijo jezika PLM [108]. Jezik PLM ima osnovne značilnosti jezika algol (z nekaterimi podobnostmi z jeziki pascal, modula-2, itd.). Lahko bi rekli, da je jezik *algolskega tipa*, kar pomeni, da omogoča osnovne značilnosti jezika algol, kot so podprogrami, vgnezdenje klicev podprogramov, podpora rekurziji itd. Za boljšo predstavitev si poglejmo preprost primer programa za izračun fakultete, ki ga definiramo s podprogramom.

```
program main;
var x;
 procedure fakulteta;
  var a,b;
  begin
    a := x; b := x;
    while a>1 do
     a := a-1;
     b := b * a;
    end;
   x := b;
  end fakulteta;
```

```
begin
 x := 4;
 call fakulteta;
end main.
```

Kot osnovo za primerjavo smo vzeli že implementirano različico specifikacij. Hierarhija jezikov za jezik PLM je prikazana na sliki 7.3. Sama zasnova implementacije, kljub inkrementalnemu razvoju jezika, ni dovolj modularna in ne sledi ločevanju dolžnosti na posamezne komponente jezika. Seveda je zasnova razumljiva, saj avtor ni načrtoval jezika z aspektnim pristopom. Same zasnove v aspektni implementaciji nismo spreminjali, temveč smo le identificirali vzorce semantičnih funkcij, ki bi jih bilo mogoče opisati z aspekti in na ta način zmanjšati kompleksnost razvoja.



Slika 7.3: Hieararhija dedovanja jezikov za jezik `PLM`.

V prvem jeziku PLM_1 so implementirani osnovni konstrukti jezika, kot so deklaracije in prireditve v glavnem programu. Specifikacije tega jezika so napram prejšnjemu primeru kar obsežne in v celoti zapisane v enem jeziku. To dejstvo botruje slabšim rezultatom ob uporabi aspektnih lastnosti. Bistveno smo z uporabo aspektov pridobili le na metriki eLOC in SEMFUN. Če pri metrikah ne upoštevamo aspektnih lastnosti, dobimo presenetljivo dobre rezultate, ki so za metriko SEMFUN zadovoljivi tudi ob upoštevanju le–teh (56.25% izboljšanje). Ob upoštevanju kompleksnosti definicije aspektov (eLOC=36, PROD=19, SEMFUN=6) ugotovimo, da nismo bistveno pridobili na zmanjševanju kompleksnosti (metrika PROD je celo 157.57%). Ugotavljamo, da je za neizkušenega razvijalca kompleksnost specifikacij pri ka-

snejši uvedbi aspektov prevelika. Ta ugotovitev samo še krepi dejstvo, da je aspektni pristop lahko zadovoljivo učinkovit samo ob primernem modularnem načrtovanju specifikacij jezika. Prav tako nam ta ugotovitev daje pritrdilen odgovor na vprašanje o smotrnosti podpore večkratnemu dedovanju aspektno usmerjenih atributnih gramatik. Ta mehanizem je namreč eden izmed ključnih za podporo modularnosti in inkrementalnemu razvoju v LISA specifikacijskem jeziku.

Tabela 7.3: Rezultati meritev metrik za jezik `PLM`.

PLM1.lisa							
Spec.	eLOC	VAR	TERM	PROD	AVS	MCC	SEMFUN
LISA	200 (100%)	16 (100%)	23 (100%)	33 (100%)	1.96 (100%)	16 (100%)	96 (100%)
Aspekti	176 (88%)	16 (100%)	23 (100%)	33 (100%)	1.96 (100%)	16 (100%)	36 (37.5%)
Aspekti+	176 (88%)	16 (100%)	23 (100.0%)	52 (157.57%)	1.23 (62.26%)	16 (100%)	42 (43.75%)
PLM2.lisa							
Spec.	eLOC	VAR	TERM	PROD	AVS	MCC	SEMFUN
LISA	235 (100%)	20 (100%)	24 (100%)	38 (100%)	2.02 (100%)	18 (100%)	109 (100%)
Aspekti	199 (84.68%)	20 (100%)	24 (100%)	38 (100%)	2.02 (100%)	18 (100%)	37 (33.9%)
Aspekti+	199 (84.68%)	20 (100%)	24 (100%)	61 (160.5%)	1.25 (62.29%)	18 (100%)	43 (39.4%)
PLM3.lisa							
Spec.	eLOC	VAR	TERM	PROD	AVS	MCC	SEMFUN
LISA	307 (100%)	20 (100%)	28 (100%)	46 (100%)	2.14 (100%)	26 (100%)	144 (100%)
Aspekti	256 (83.39%)	20 (100%)	28 (100%)	46 (100%)	2.14 (100%)	26 (100%)	54 (37.5%)
Aspekti+	256 (83.39%)	20 (100%)	28 (100%)	75 (163.04%)	1.31 (28.08%)	26 (100%)	60 (41.67%)
PLM4.lisa							
Spec.	eLOC	VAR	TERM	PROD	AVS	MCC	SEMFUN
LISA	440 (100%)	22 (100%)	30 (100%)	52 (100%)	2.38 (100%)	30 (100%)	174 (100%)
Aspekti	309 (70.23%)	22 (100%)	30 (100%)	52 (100%)	2.38 (100%)	30 (100%)	74 (42.53%)
Aspekti+	309 (70.23%)	22 (100%)	30 (100%)	85 (163.46%)	1.46 (61.34%)	30 (100%)	80 (45.98%)

Jezika PLM_2 in PLM_3 sta abstraktna, kar pomeni, da ni mogoče ustvariti instance teh jezikov[1].

V jeziku PLM_2 so implementirane konstante (stavek za definiranje konstant). Jezik PLM_3 nato združi koncepte jezikov PLM_1 in PLM_2 (večkratno dedovanje) ter doda *if* in *while* stavek. Ker sta jezika abstraktna, ni bilo mogoče izvesti meritev za monolitne specifikacije atributnih gramatik. V jeziku PLM_2 smo ponovno uporabili aspektne lastnosti, definirane že v jeziku PLM_1, kar rezultira k zmanjšanju metrike SEMFUN. V bistvu smo celotno semantiko jezika PLM2 zapisali z aspekti, v produkcijah smo definirali zgolj eno semantično funkcijo. Zaradi ponovne uporabe aspektnih lastnosti se bistveno ne poveča tudi trud za definiranje aspektnih lastnosti (metrika PROD se poveča za 2,93 odstotne točke).

[1] Če ne moremo ustvariti instance jezika, iz teh specifikacij ne moremo generirati prevajalnika. V abstraktnih jezikih ponavadi implementiramo zgolj določen ponovno uporaben koncept programskega jezika.

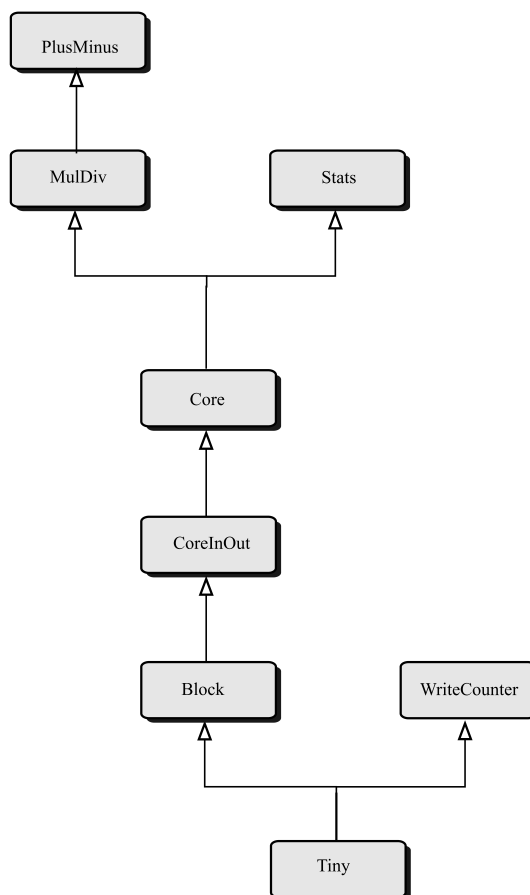
Jezik PLM_4 je specializacija jezika PLM_3, ki mu dodamo procedure in klice procedur. Za definiranje aspektnih lastnosti smo ponovno uporabili že definirane specifikatorje stičišč in nasvete, tako da so spremembe vrednosti metrik ob upoštevanju metrik za definicijo aspektov zanemarljive. Kljub temu da načrtovalec jezika ob implementaciji ni upošteval ločevanja dolžnosti in jezika ni načrtoval aspektno, smo z aspektnim pristopom prišli do zelo dobrih rezultatov. V poprečju smo kompleksnost aspektne implementacije zmanjšali ta 14.79%, ob neupoštevanju kompleksnosti definiranja aspektnih lastnosti pa kar za 43.62%. Pri tem podatku smo predpostavili, da vse metrike predstavljajo enako kompleksnost.

Primer 4: Jezik *Tiny*

V tej podsekciji prikažemo analizo primera, kateregа implementacijo smo predstavili v podpoglavju 6.4 (od strani 113 naprej). Hierarhija jezikov je prikazana na sliki 7.4, rezultati same analize pa v tabeli 7.4. Glede na to, da sta jezika `WriteCounter` in `Block` abstraktna, ju nismo vključili v analizo. Pri implementaciji jezika brez aspektnih lastnosti smo uporabili hierarhijo jezikov, kot je prikazana na sliki 7.4.

Tabela 7.4: Rezultati meritev metrik za jezik `Tiny`.

PlusMinus.lisa							
Spec.	eLOC	VAR	TERM	PROD	AVS	MCC	SEMFUN
LISA	19 (100%)	3 (100%)	3 (100%)	5 (100%)	1.8 (100%)	2 (100%)	5 (100%)
MulDiv.lisa							
Spec.	eLOC	VAR	TERM	PROD	AVS	MCC	SEMFUN
LISA	38 (100%)	4 (100%)	7 (100%)	11 (100%)	1.90 (100%)	7 (100%)	10 (100%)
Stats.lisa							
Spec.	eLOC	VAR	TERM	PROD	AVS	MCC	SEMFUN
LISA	4 (100%)	2 (100%)	0 (100%)	1 (100%)	2 (100%)	1 (100%)	0 (100%)
Core.lisa							
Spec.	eLOC	VAR	TERM	PROD	AVS	MCC	SEMFUN
LISA	106 (100%)	7 (100%)	12 (100%)	27 (100%)	1.96 (100%)	20 (100%)	40 (100%)
Aspekti	79 (74.53%)	7 (100%)	12 (100%)	17 (62.96%)	1.88 (95.89%)	10 (50.00%)	14 (35.00%)
Aspekti+	79 (74.53%)	7 (100%)	12 (100%)	21 (77.77%)	1.52 (77.63%)	10 (50.00%)	20 (50.00%)
CoreInOut.lisa							
Spec.	eLOC	VAR	TERM	PROD	AVS	MCC	SEMFUN
LISA	173 (100%)	7 (100%)	14 (100%)	43 (100%)	2.05 (100%)	36 (100%)	81 (100%)
Aspekti	105 (60.69%)	7 (100%)	14 (100%)	22 (51.16%)	1.90 (93.29%)	15 (41.66%)	20 (24.69%)
Aspekti+	105 (60.69%)	7 (100%)	14 (100%)	25 (58.14%)	1.68 (82.09%)	15 (41.66%)	21 (25.93%)
Tiny.lisa							
Spec.	eLOC	VAR	TERM	PROD	AVS	MCC	SEMFUN
LISA	282 (100%)	9 (100%)	18 (100%)	63 (100%)	2.15 (100%)	54 (100%)	130 (100%)
Aspekti	137 (48.58%)	9 (100.0%)	18 (100.0%)	26 (41.27%)	2.11 (97.99%)	17 (31.48%)	29 (22.37%)
Aspekti+	137 (48.58%)	9 (100.0%)	18 (100.0%)	38 (60.31%)	1.44 (67.05%)	17 (31.48%)	37 (28.46%)



Slika 7.4: Hieararhija dedovanja jezikov za jezik `Tiny`.

Iz tabele je razvidno, da s primernim načrtovanjem, aspektno usmerjen pristop daje bistveno boljše rezultate kot ostali mehanizmi za podporo inkrementalnemu načrtovanju. Kot je bilo pričakovano, so v tem primeru rezultati najboljši, saj smo jezik načrtovali aspektno in posamezne dolžnosti razvijali ločeno. Za aplikacijo semantičnih vzorcev pa smo uporabljali zgolj aspektne lastnosti LISA specifikacijskega jezika.

Skupni rezultati

V tabeli 7.5 so predstavljene povprečne vrednosti metrik za posamezno implementacijo končnih jezikov. Kot je razvidno iz tabele, se v povprečju rezultati za aspektno implementacijo jezikov precej boljši od ostalih implementacij. Predvsem opažamo trend zniževanja vrstic specifikacij ter kompleksnosti. Te

lastnosti se izkažejo v bistvenem zniževanju metrik eLOC, MCC, in SEMFUN, ki so glavni pokazatelji kompleksnosti specifikacij. Čeprav se modularen razvoj jezikov s koncepti aspektno usmerjenega programiranja pri majhnih jezikih ne izkaže najbolje, so prednosti pristopa pri večjih jezikih toliko bolj opazne. Enak vzorec padanja kompleksnosti specifikacij smo opazili pri vseh primerih implementiranih jezikov.

Tabela 7.5: Sumarna predstavitev rezultatov

Calc.lisa + RobotExpressions.lisa + PLM4.lisa + Tiny.lisa					
Spec.	eLOC	PROD	AVS	MCC	SEMFUN
LISA	229.25 (100%)	44.00 (100%)	2.26 (100%)	27.00 (100%)	102.00 (100%)
Aspekti	140.75 (61.39%)	29.75 (66.47%)	2.18 (96.23%)	19.67 (73%)	35.00 (34.31%)
Aspekti+	140.75 (61.39%)	55.75 (126.70%)	1.57 (67.86%)	19.67 (73%)	41.50 (40.69%)

Povzetek analize

Z željo po čimvečji ponovni uporabi poskušamo specifikacije zapisati čimbolj modularno in neodvisno. Pri tem nam na konceptualni ravni pomaga AOP. V analizi implementiranih jezikov smo potrdili naše hipoteze, da se sklopljenost komponent in kompleksnost specifikacij z uporabo aspektno usmerjenega pristopa niža. Posebej smo zadovoljni s pozitivnim trendom upadanja zahtevnosti specifikacij pri večanju jezika, ki ga razvijamo.

In the end, it's not going to matter how many breaths you took, but how many moments took your breath away.

– Shing Xiong –

Poglavje 8

Zaključek

V disertaciji zagovarjamo stališče, da so modularnost, ponovna uporabnost in kompleksnost bistveni koncepti za uspešen razvoj programskih sistemov, med katere zanesljivo sodi tudi razvoj programskih jezikov. Z modularnim pristopom k načrtovanju in implementaciji programskih jezikov lahko bistveno povečamo ponovno uporabnost specifikacij. Pogoj za ponovno uporabnost pa je rahla sklopljenost komponent, kjer je zapisan posamezen koncept programskega jezika.

Izmed novejših pristopov za ločevanje dolžnosti se vedno bolj uvelja aspektno usmerjeno programiranje. Za izboljšanje specifikacij programskih jezikov se je tako mogoče zanesti na že preverjene koncepte za ločevanje dolžnosti. Aspektno usmerjeno programiranje smo uspešno prenesli na področje atributnih gramatik in postavili formalni model aspektno usmerjenih atributnih gramatik, ki omogoča modularnejši razvoj programskih jezikov in s tem zmanjšuje kompleksnost specifikacij za implementacijo programskega jezika.

Po uvodnem poglavju se bralec uvede v svet programskih jezikov. Temu poglavju sledi poglavje o aspektno usmerjenem programiranju (poglavje 3), kjer prikažemo lastnosti in prednosti aspektno usmerjenega pristopa pri načrtovanju in implementaciji programskih sistemov ter spoznamo tudi koncepte

aspektno usmerjenih jezikov. Poglavje 4 opisuje teorijo programskih jezikov ter formalne metode za zapis specifikacij programskih jezikov. Formalne specifikacije programskih jezikov omogočajo večji nivo abstrakcije in modularnost zapisa ter možnost avtomatskega generiranja prevajalnika za razvit programski jezik. Slednje je tudi eden izmed ciljev disertacije. Poglavje je zaključeno s predstavitvijo formalnih modelov in sorodnih del.

Osrednji del doktorske disertacije predstavlja poglavje 5, kjer zapišemo formalen model aspektno usmerjenih atributnih gramatik in večkratnega dedovanja le–teh. Poglavje prikaže motivacijo za uvedbo aspektov v formalni model atributnih gramatik in postavljene teze na koncu poglavja tudi potrdi s praktičnim primerom. Razviti teoretični model smo implementirali v obliki aspektnega domensko specifičnega jezika za orodje LISA (poglavje 6). Iz takšnih specifikacij je mogoče avtomatsko generirati prevajalnik in druga orodja, kar orodje LISA omogoča. Razviti pristop se je izkazal za zelo praktično uporabnega, saj omogoča večjo modularnost, šibko sklopljenost komponent ter zmanjša kompleksnost specifikacij, kar posledično zmanjša trud pri razvoju novega programskega jezika. Slednje smo potrdili z implementacijo različnih prototipnih jezikov, ki smo jih ovrednotili z znanimi metrikami s področja gramatik, ki smo jim dodali tudi nekatere svoje. Rezultate prikazujemo v poglavju 7.

8.1 Cilji

V disertaciji zagovarjamo tezo o primernosti uporabe konceptov aspektno usmerjenega programiranja na področju načrtovanja in implementacije programskih jezikov. Celo več, s tem pristopom lahko pričakujemo celo boljše rezultate kot z dosedaj uporabljenimi pristopi. Ker smo se pri naši raziskavi omejili na atributne gramatike, je bil osnovni cilj disertacije nadgradnja atributnih gramatik s koncepti aspektno usmerjenega programiranja. V doktorski disertaciji smo zagovarjali naslednjo tezo:

Aspektno usmerjene atributne gramatike so primeren pristop za načrtovanje programskih jezikov, s katerim je mogoče doseči povečanje ponovne uporabnosti in zmanjšati obseg formalnih specifikacij, v primerjavi z osnovnimi atributnimi gramatikami.

Iz podane teze lahko izpeljemo več hipotez, ki jih zagovarjamo v nadaljevanju.

Formalni model aspektno usmerjenih atributnih gramatik smo implemen-

tirali v obliki domensko specifičnega aspektnega jezika za specifikacijo programskih jezikov, iz katerega je mogoče avtomatsko generirati pregledovalnik, razpoznavalnik, prevajalnik in druga orodja (orodje LISA). Z implementiranim jezikom smo uspešno načrtovali več prototipnih programskih jezikov, kar potrjuje naslednjo hipotezo:

Hipoteza 1: *Aspektno usmerjeno večkratno dedovanje atributnih gramatik je primeren pristop za specifikacijo programskih jezikov.*

Ponovna uporaba je ena izmed ključnih lastnosti za uspešnost modernih programskih jezikov. Eden izmed mehanizmov za ponovno uporabo v programskih jezikih je dedovanje. V gramatikah se je izkazalo, da večkratno dedovanje bistveno pripomore k učinkovitejši ponovni uporabi že razvitih komponent programskih jezikov, kar nas je vzpodbudilo k vpeljavi večkratnega dedovanja v model aspektno usmerjenih atributnih gramatik. V poglavju 7 prikazujemo implementacijo več prototipnih programskih jezikov, ki smo jih razvili z uporabo večkratnega dedovanja ter z uporabo aspektnih lastnosti specifikacijskega jezika in brez aspektov. Analize kažejo na zmanjšanje kompleksnosti in izboljšanje ponovne uporabnosti komponent ob uporabi aspektno usmerjenega pristopa, kar potrjuje naslednjo hipotezo:

Hipoteza 2: *Z razširitvijo večkratnega dedovanja atributnih gramatik z uporabo aspektno usmerjenega pristopa lahko dodatno povečamo ponovno uporabnost specifikacij.*

Ob uporabi aspektnih lastnosti atributnih gramatik se je pri implementaciji prototipnih programskih jezikov obseg specifikacij (metriki eLOC in SEMFUN) pri vseh jezikih precej zmanjšal, kar potrjuje naslednjo hipotezo:

Hipoteza 3: *Aspektno usmerjene atributne gramatike lahko zmanjšajo obseg specifikacij za definicijo programskega jezika.*

Cilje, ki smo si jih zadali, smo z raziskavo dosegli in ponekod celo presegli. Dobljeni rezultati so zadovoljivi in vzpodbujajo nadaljne raziskave na področju razvoja programskih jezikov. Disertacija uvaja nove koncepte na področju načrtovanja, formalnega zapisa specifikacij programskih jezikov ter avtomatskega generiranja interpreterjev/prevajalnikov za programski jezik. Koncepti so združitev in nadgradnja že znanih konceptov na področju načrtovanja in implementacije programskih jezikov, ob tem pa disertacija dodaja še naslednje novosti:

- definiranje formalnega modela aspektno usmerjenih atributnih gramatik,
- definicija in implementacija algoritma za tkanje v aspektno usmerjenih atributnih gramatikah,
- definicija in implementacija algoritma za večkratno dedovanje aspektno usmerjenih atributnih gramatik,
- načrtovanje in implementacija novega domensko specifičnega aspektnega specifikacijskega jezika v orodju LISA,
- parametrizacija nasvetov in definicija generičnih semantičnih funkcij znotraj nasveta in
- načrtovanje programskih jezikov z uporabo vzorcev za načrtovanje in avtomatsko generiranje interpreterja/prevajalnika iz podanih formalnih aspektno usmerjenih specifikacij programskega jezika.

8.2 Spoznanja in razprave

Predstavljeni pristop pri načrtovanju in implementaciji se je pri razvoju prototipnih jezikov izkazal za zelo uporabnega. Seveda pa pristop lahko prinaša tudi določene pasti in slabosti. Prednosti predstavljenega pristopa si poglejmo s samoevalvacijsko SWOT (***S****trengths,* ***W****eaknesses,* ***O****pportunities,* ***T****hreats*) analizo:

Prednosti (Strengths) Prednosti pristopa vidimo predvsem v modularizaciji in ponovni uporabi komponent, saj pristop omogoča ločevanje dolžnosti v fazi načrtovanja programskega jezika.

Pomanjkljivosti (Weaknesses) Glavna pomanjkljivost trenutnega dela je slabša podpora aspektom na nivoju orodja za razvoj programskih jezikov. Ta slabost se pokaže predvsem pri implementaciji večjih programskih jezikov, kjer je nivo ponovne uporabe visok, aplikacija aspektov pa lahko poruši strukturo semantičnih funkcij. S primernim orodjem, ki bi podpiralo reševanje konfliktov, se tej slabosti izognemo.

Priložnosti (Opportunities) Predlagana rešitev odpira mnoga področja za nadaljne raziskave.

Nevarnosti (Threats) Konkurenca je na področju programskih jezikov zelo močna. Zaradi majhnosti raziskovalne skupine zelo težko držimo korak s konkurenco, predvsem na področjih, ki zahtevajo veliko napora (npr., implementacija orodij, pomoči, spletnih strani itd.). V tem oziru nam ob rednem objavljanju naših dognanj v znanstvenih publikacijah grozi, da nas konkurenca prehiti z vključitvijo naših konceptov v svoja orodja.

Glede na široki spekter uporabe atributnih gramatih smo prepičani, da je možno enak pristop uporabiti tudi v drugih domenah, ne zgolj pri načrtovanju in implementaciji programskih jezikov. Slednje ni vključeno v raziskave v okviru doktorske disertacije in je ena izmed možnih nadaljnih smernic raziskav. Prav tako menimo, da ima predstavljen pristop velik potencial za nadaljne raziskave na področju načrtovanja in implementacije programskih jezikov. Ob povečani modularnosti in rahli sklopljenosti komponent programskega jezika ter parametrizaciji generičnih semantičnih funkcij v nasvetih se poraja misel o knjižnici aspektov, ki bi vsebovala abstraktno implementacijo posameznih konceptov programskega jezika. Seveda se pri tem poraja misel o neodvisnosti od sintakse načrtovanega programskega jezika. Le–to bi bilo mogoče v popolnosti zagotoviti s parametrizacijo jezika (podobno kot pri nasvetih). Na ta način bi na jezik (komponento z implementacijo koncepta) gledali kot na šablono, ki jo v končnih specifikacijah na ustreznem mestu instanciramo. Razmišljanje nakazuje le eno od možnih poti nadaljnih raziskav, ki bi po našem mnenju še olajšalo delo razvijalcem programskih jezikov.

Disertacijo zaključujemo z latinskim pregovorom “*Cum quad habere cupis, noli cessare petendo: arbor nonprimo, sed saepe cadit.*”[1], ki nas uči, da moramo biti za dosego ciljev vztrajni, in da nas do idealnih rešitev čaka še precej, precej dela.

[1] Če nekaj želiš imeti, ne prenehaj iskati; drevo ne pade po prvem, ampak po mnogih udarcih.

LITERATURA

[1] M. Mernik. *Ponovno uporabni semantični opis pri načrtovanju in implementaciji programskih jezikov.* Doktorska disertacija, Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko, 1998.

[2] *Sveto pismo stare in nove zaveze : slovenski standardni prevod.* Svetopisemska družba Slovenije, Ljubljana, 1997.

[3] D. Adams. *The hitchhiker's guide to the galaxy.* Pan Books, London, 1979.

[4] Esperanto. http://www.esperanto.si, 2008.

[5] List of programming languages.
http://en.wikipedia.org/wiki/list_of_programming_languages, 2008.

[6] Collection on Computer Programming Languages.
http://people.ku.edu/ nkinners/langlist/extras/langlist.htm, 2008.

[7] History of Programming Languages.
http://en.wikipedia.org/wiki/history_of_programming_languages, 2008.

[8] Najpogosteje uporabljeni programski jeziki.
http://www.tiobe.com/tpci.htm, 2008.

[9] K. Nygaard and O.-J. Dahl. Simula 67. In R. W. Wexelblat, editor, *History of Programming Languages.* acm press, 1981.

[10] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation.* Addison-Wesley, Boston, MA, USA, 1983.

[11] V. Žumer, N. Korbar. *Programiranje v jeziku C++.* Fakulteta za elektrotehniko, računalništvo in informatiko, 1997.

[12] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17:471–522, 1985.

[13] V. Žumer, and M. Mernik. *Principi programskih jezikov.* Fakulteta za elektrotehniko, računalništvo in informatiko, 2001.

[14] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools and Applications.* Addison-Wesley, 2000.

[15] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000.

[16] M. Mernik, J. Heering, and A. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

[17] J. W. Backus. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference. In *IFIP Congress*, pages 125–131, 1959.

[18] C. Jones. End-User Programming. *IEEE Computer*, 28(9):68–70, 1995.

[19] T. Kosar. *Razvoj domensko specifičnih jezikov iz aplikacijskih vmesnikov.* Doktorska disertacije, Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko, 2007.

[20] T. Kosar, P. E. M. López, P. A. Barrientos, , and M. Mernik. A preliminary study on various implementation approaches of domain-specific language. *Information and Software Technologies*, 50(5):390–405, 2008.

[21] E. Avdičaušević. *Aspektno usmerjeno programiranje.* Magistrsko delo, Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko, 2001.

[22] R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors. *Aspect-Oriented Software Development.* Addison-Wesley, 2005.

[23] M. G. Gouda and T. Herman. Adaptive programming. *IEEE Transactions on Software Engineering*, 17(9):911–921, 1991.

[24] http://www.ccs.neu.edu/research/demeter/DemeterJava. DemeterJ, 2007.

[25] L. Bergmans, M. Aksit, and K. Wakita. An object-oriented model for extensible concurrent systems: The composition-filters approach. *IEEE Transactions on Parallel and Distributed Systems*, 1993.

[26] H. Ossher and P. Tarr. Multi-dimensional separation of concerns using Hyperspaces. Technical Report 21452, IBM Research Report, April 1999.

[27] H. Ossher and P. L. Tarr. Hyper/J^{TM}: Multi-dimensional separation of concerns for java^{TM}. In *ICSE*, pages 821–822. IEEE Computer Society, 2001.

[28] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *ECOOP97 Object-Oriented Programming, Lecture Notes in Computer Science*, volume 1241, pages 220–242. Springer-Verlag, 1997.

[29] http://en.wikipedia.org/wiki/Aspect oriented_programming. Spletna stran *Wiki* za aspektno usmerjeno programiranje, 2008.

[30] http://www.aosd.net. Spletna stran aspektno usmerjenega razvoja programske opreme, 2008.

[31] 1st Domain-Specific Aspect Languages Workshop. http://dsal.dcc.uchile.cl/2006, Oktober 2006.

[32] 2nd Domain-Specific Aspect Languages Workshop. http://dsal.dcc.uchile.cl/2007, Marec 2007.

[33] 3rd Domain-Specific Aspect Languages Workshop. http://dsal.dcc.uchile.cl/2008, April 2008.

[34] Ogrodje Spring. http://www.springframework.org.

[35] Aplikacijski strežnik JBoss. http://www.jboss.org/.

[36] http://www.eclipse.org/aspectj. Spletna stran projekta `AspectJ`, 2008.

[37] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting started with AspectJ. *Communications of the ACM (Special issue on Aspect-Oriented Programming)*, 44(10):59–65, October 2001.

[38] C. Lopes. *D: A language framework for distributed programming.* PhD thesis, Northeastern University, 1997.

[39] http://ant.apache.org. Spletna stran projekta `ant`, 2008.

[40] C. Lopes. *Aspect-Oriented Programming: A Historical Perspective.* In Aspect-Oriented Software Development, R. Filman, T. Elrad, M. Aksit, S. Clarke (eds.), Addison-Wesley, 2004.

[41] C. Courbis and A. Finkelstein. Towards aspect weaving applications. In *Proceedings of International Conference on Software Engineering (ICSE)*, pages 69–77, 2005.

[42] M. Shonle, K. Lieberherr, and A. Shah. XAspects: An extensible system for domain-specific aspect languages. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 28–37, 2003.

[43] J. Stankovic, R. Zhu, R. Poornalingam, C. Lu, Z. Yu, M. Humphrey, and B. Ellis. VEST: An aspect-based composition tool for real-time systems. In *Real-Time Applications Symposium*, pages 58–69, 2003.

[44] J. Irwin, J. Loingtier, J. R. Gilbert, G. Kiczales, J. Lamping, A. Mendhekar, and T. Shpeisman. Aspect-oriented programming of sparse matrix code. In *Proceedings of International Scientific Computing in Object-Oriented Parallel Environments*, 1997.

[45] J. Gray, T. Bapty, S. Neema, and J. Tuck. Handling crosscutting constraints in domain-specific modeling. *Communications of the ACM, Special Issue on Aspect-Oriented Programming*, pages 87–93, October 2001.

[46] J. Hugunin. The next steps for aspect-oriented programming languages. In *NSF Workshop on Software Design and Productivity*, 2001.

[47] N. Chomsky. Three models for the description of language. *IRI Transactions on Information Theory*, 2(3):113–124, 1956.

[48] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[49] D. Scott and C. Strachey. Toward a mathematical semantics for computer languages. Programming Research Group Technical Monograph PRG-6, Oxford Univ. Computing Lab., 1971.

[50] A. Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33:346–366, 1932.

[51] F. Nielson and H. R. Neilson. *Semantics with Applications - A Formal Introduction*. John Wiley and Sons, New York, NY, 1992.

[52] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications ACM*, 26(1):53–56, 1983.

[53] M. Broy, M. Wirsing, and P. Pepper. On the algebraic definition of programming languages. *ACM Transactions on Programming Languages and Systems*, 9(1):54–99, 1987.

[54] H. Alblas and B. Melichar, editors. *Attribute Grammars, Applications and Systems, International Summer School SAGA, Prague, Czechoslovakia, June 4-13, 1991, Proceedings*, volume 545 of *Lecture Notes in Computer Science*. Springer, 1991.

[55] T. Kosar, M. Mernik, V. Žumer, P. R. Henriques, and M. J. Varanda. Software development with grammatical approach. *Informatica (Ljublj.)*, 28(4):393–404, 2004.

[56] G. Riedewald. Prototyping by using an attribute grammar as a logic program. In *Proceedings on Attribute Grammars, Applications and Systems*, pages 401–437, London, UK, 1991. Springer-Verlag.

[57] I. Guessarian. *Algebraic semantics*, volume 99 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.

[58] E. Visser. Polymorphic syntax definition. *TCS: Theoretical Computer Science*, 199, 1998.

[59] J. R. Levine, T. Mason, and D. Brown. *Lex and Yacc*. Nutshell Handbook. O'Reilly and Associates, Sebastopol, California, U.S.A., 2nd edition, 1992.

[60] P. Borras, D. Clement, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: the system. In P. Henderson, editor, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, number 2 in SIGPLAN, pages 14–24. ACM, February 1988.

[61] G. Riedewald. The LDL - language development laboratory. In *CC '92: Proceedings of the 4th International Conference on Compiler Construction*, pages 88–94, London, UK, 1992. Springer-Verlag.

[62] E. Astesiano and G. Reggio. An qutline of the SMoLCS approach. In Marisa Venturini Zilli, editor, *Mathematical Models for the Semantics of Parallelism*, volume 280 of *Lecture Notes in Computer Science*, pages 81–113. Springer, 1986.

[63] ANTLR – ANother Tool for Language Recognition. http://www.antlr.org, 2008.

[64] M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The ASF + SDF meta-environment: A component-based language development environment. *Lecture Notes in Computer Science*, 2027:365–370, 2001.

[65] A. Van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific, Singapore, 1996.

[66] P. Klint, T.van der Storm, and J.J. Vinju. Term rewriting meets aspect-oriented programming. Technical report, CWI, 2004.

[67] I. Attali, C. Courbis, P. Degenne, A. Fau, J. Fillon, D. Parigot, C. Pasquier, and C. Sacerdoti Coen. *SmartTools: a generator of interactive environments tools*. Springer Verlag, 2001.

[68] G. Hedin and E. Magnusson. JastAdd: an aspect-oriented compiler construction system. *Science of Computer Programming*, 47(1):37–58, 2003.

[69] LISA. http://labraj.uni-mb.si/lisa, 2008.

[70] M. Mernik, M. Lenič, E. Avdičaušević, and V. Žumer. LISA: An Interactive Environment for Programming Language Development. In Nigel Horspool, editor, *11th International Conference on Compiler Construction*, volume 2304, pages 1–4. Lecture Notes in Computer Science, Springer-Verlag, 2002.

[71] R. Lämmel and G. Wachsmuth. Transformation of SDF syntax definitions in the ASF+SDF Meta-Environment. In Mark van den Brand

and Didier Parigot, editors, *Proceedings of the First Workshop on Language Descriptions, Tools and Applications (LDTA'01), Genova, Italy, April 7, 2001, Satellite event of ETAPS'2001*, volume 44 of *ENTCS*. Elsevier Science, April 2001.

[72] J. Emilio L. Gayo, M. C. L. Díez, J. M. C. Lovelle, and A. Cernuda del Río. LPS: A language prototyping system using modular monadic semantics. *Electr. Notes Theor. Comput. Sci*, 44(2), 2001.

[73] R. W. Gray, V. P. Heuring, S. P. Levi, A. M. Sloane, and W. M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–131, 1992.

[74] W. Waite, U. Kastens, and A. M. Sloane. *Generating Software from Specifications*. Jones and Bartlett Publishers, Inc., USA, 2007.

[75] Seznam generatorjev razpoznavalnikov. http://en.wikipedia.org/wiki/list_of_parser_generators, 2008.

[76] K. Nakamura and T. Ishiwata. Synthesizing context free grammars from sample strings based on inductive CYK algorithm. In *Grammatical Inference: Algorithms and Applications, 5th International Colloquium, ICGI 2000, Lisbon, Portugal, September 11 - 13, 2000; Proceedings*, volume 1891 of *Lecture Notes in Artificial Intelligence*, pages 186–195. Springer, 2000.

[77] M. Črepinšek. *Hevristični in evolucijski algoritmi pri sklepanju o kontekstno neodvisnih gramatikah*. Doktorska disertacija, Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko, 2007.

[78] M. Črepinšek, M. Mernik, F. Javed, B. R. Bryant, and A. Sprague. Extracting grammar from programs: evolutionary approach. *SIGPLAN Not.*, 40(4):39–46, 2005.

[79] M. Črepinšek, M. Mernik, and V. Žumer. Extracting grammar from programs: brute force approach. *SIGPLAN Not.*, 40(4):29–38, 2005.

[80] R. A. Heinlein. *The Moon Is a Harsh Mistress*. G. P. Putnam's Sons, 1966.

[81] U. Kastens and W.M. Waite. Modularity and reusability in attribute grammars. *Acta Informatica*, 31:601 – 627, 1994.

[82] G.D.P. Dueck, G.V. Cormack. Modular attribute grammars. *Computer Journal*, 33(2):164 – 172, 1990.

[83] J. Paakki. Attribute grammar paradigms — A high-level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–255, June 1995.

[84] G. Hedin. An overview of door attribute grammars. *Proceedings of the International Conference on Compiler Construction, Springer - Verlag*, 1994.

[85] G. Hedin. An object-oriented notation for attribute grammars. In S. Cook, editor, *Proceedings of the ECOOP '89 European Conference on Object-oriented Programming*, pages 329–345, Nottingham, July 1989. Cambridge University Press.

[86] G. Hedin. Reference attributed grammars. *Informatica*, 24:301–318, 2000.

[87] M. Akist, R. Mostert, B. Haverkort. Grammar inheritance. *Technical Report, Department of Computer Science, University of Twente*, 1991.

[88] M. Mernik, M. Lenič, E. Avdičaušević, and V. Žumer. Multiple Attribute Grammar Inheritance. *Informatica*, 24(3):319–328, 2000.

[89] M. Lenič. *Načrtovanje in implementacija programskih jezikov z uporabo večkratnega dedovanja atributnih gramatik.* Magistrsko delo, Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko, 2001.

[90] A. Taivalsaari. On the Notion of Inheritance. *ACM Computing Surveys*, 28(3):438 – 479, 1996.

[91] J. T. Boyland. *Descriptional Composition of Compiler Components.* PhD thesis, University of California, Berkeley, September 1996. Available as technical report UCB//CSD-96-916.

[92] M. Anlauff, P. W. Kutter, and A. Pierantonio. Montages/Gem-Mex: A Meta Visual Programming Generator. In *VL '98: Proceedings of the IEEE Symposium on Visual Languages*, page 304, Washington, DC, USA, 1998.

[93] C. Denzler. *Modular Language Specification and Composition.* Swiss Federal Institute of Technology, 2001.

[94] J. Paakki. Attribute Grammar Paradigms - A High-Level Methodology in Language Implementation. *ACM Computing Surveys*, 27(2):196 – 255, 1995.

[95] W. M. Waite. Use of attribute grammars in compiler construction. In *INRIA: Attribute Grammars and their Applications, International Conference (WAGA)*, pages 255–265, Berlin - Heidelberg - New York, September 1990. Springer.

[96] P. Deransart and M. Jourdan, editors. *Attribute Grammars and their Applications. Proceedings of 1st WAGA*, volume 461. Lecture Notes in Computer Science, Springer-Verlag, 1990.

[97] D. Parigot and M. Mernik, editors. *Attribute Grammars and their Applications. Proceedings of 2nd WAGA*. INRIA, 1999.

[98] D. Parigot and M. Mernik, editors. *Attribute Grammars and their Applications. Proceedings of 3rd WAGA*. INRIA, 2000.

[99] P. Henriques, M. Varanda Pereira, M. Mernik, M. Lenič, J. Gray, and H. Wu. Automatic generation of language-based tools using LISA. *IEE Proceedings - Software Engineering*, 152(2):54–69, April 2005.

[100] R. Grimm. Better extensibility through modular syntax. In *Conference on Programming Language Design and Implementation*, pages 38–51, 2006.

[101] D. Rebernak, M. Mernik, P. R. Henriques, and M. J. V. Pereira. AspectLISA: an aspect-oriented compiler construction system based on attribute grammars. *Electronic notes in theoretical computer science*, (2):37–53, 2006.

[102] E. Avdičaušević, M. Lenič, M. Mernik, and V. Žumer. AspectCOOL: An experiment in design and implementation of aspect-oriented language. *ACM SIGPLAN Notices*, 36(12):84–94, December 2001.

[103] D. Rebernak, M. Črepinšek, and M. Mernik. Web service for designing and implementing formal languages. In *ITI 2006 : Proceedings of the 28th International Conference on Information Technology Interfaces*, pages 463–468. SRCE University Computing Centre, University of Zagreb, 2006.

[104] W. Lohman, G. Riedewald, and T. Zühlke. A lightweight infrastructure to support experimenting with heterogeneous transformations. In *.NET Technologies 2006*, pages 35–42, 2006.

[105] R. Lämmel. Declarative aspect-oriented programming. In *PEPM*, pages 131–146, 1999.

[106] J. Martin and C. L. McClure. *Software Maintenance: The Problems and Its Solutions.* Prentice Hall Professional Technical Reference, 1983.

[107] J. F. Power and B. A. Malloy. A metrics suite for grammar-based software. *Journal of Software Maintenance*, 16(6):405–426, 2004.

[108] B. Vilfan. *Prevajanje programskih jezikov 1. del.* Univerza v Ljubljani, Fakulteta za elektrotehniko in računalništvo, Ljubljana, 1991.

Življenjepis

Ime in priimek:	Damijan Rebernak	
Rojen:	22. februar 1978, Maribor, Slovenija	
Šolanje:		
	1985 – 1992	Osnova šola Šmartno na Pohorju, Šmartno na Pohorju
	1992 – 1996	Srednja elektro-računalniška šola Maribor, Maribor
	1996 – 2000	Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko (VS študij)
	2001 – 2003	Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko (UNI študij)
	2003 – 2008	Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko (doktorski študij)
Zaposlitev:		
	1999 – 2003	tehniški sodelavec, Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko
	2003 – 2008	mladi raziskovalec, Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko
	2008 –	Adacta, programska oprema d. o. o.

Bibliografija

DAMIJAN REBERNAK [20401]
Osebna bibliografija za obdobje 1999-2007

1.01 Izvirni znanstveni članek

1. MERNIK, Marjan, ČREPINŠEK, Matej, KOSAR, Tomaž, REBERNAK, Damijan, ŽUMER, Viljem. Grammar-based systems: definition and examples. *Informatica (Ljublj.)*, 2004, let. 28, št. 3, str. 245-255. [COBISS.SI-ID 9286166]

2. GREINER, Sašo, REBERNAK, Damijan, BREST, Janez, ŽUMER, Viljem. Z0 - a tiny experimental language. *SIGPLAN not.*, Avg. 2005, vol. 40, no 8, str. 19-28. [COBISS.SI-ID 9879574]
JCR IF: 0.175, SE (76/79), computer science, software engineering, x: 0.937

3. REBERNAK, Damijan, MERNIK, Marjan, HENRIQUES, Pedro Rangel, DA CRUZ, Daniela, VARANDO PEREIRA, Maria João. Specifying Languages Using Aspect-oriented Approach: AspectLISA. *CIT. J. Comput. Inf. Technol.*, 2006, vol. 14, no. 4, str. 343-350. [COBISS.SI-ID 11014678]

1.08 Objavljeni znanstveni prispevek na konferenci

4. BOŠKOVIĆ, Borko, BREST, Janez, REBERNAK, Damijan, ŽUMER, Viljem. Načrtovanje in gradnja informacijskega sistema s trinivojsko arhitekturo. V: ZAJC, Baldomir (ur.). *Zbornik devete Elektrotehniške in računalniške konference ERK 2000, 21. - 23. september 2000, Portorož, Slovenija.* Ljubljana: IEEE Region 8, Slovenska sekcija IEEE, 2000, zv. B, str. 47-50. [COBISS.SI-ID 5802262]

5. REBERNAK, Damijan, LENIČ, Mitja, MERNIK, Marjan, ŽUMER, Viljem. Načrtovanje in implementacija pomoči in uporabniškega priročnika z uporabo standarda XML. V: ZAJC, Baldomir (ur.). *Zbornik devete Elektrotehniške in računalniške konference ERK 2000, 21. - 23. september 2000, Portorož, Slovenija.* Ljubljana: IEEE Region 8, Slovenska sekcija IEEE, 2000, zv. B, str. 51-54. [COBISS.SI-ID 5802774]

6. BOŠKOVIĆ, Borko, REBERNAK, Damijan, BREST, Janez, MERNIK, Marjan, ŽUMER, Viljem. Orodje za izdelavo aplikacij v spletnem informacijskem sistemu. V: ZAJC, Baldomir (ur.). *Zbornik desete Elektrotehniške in računalniške konference ERK 2001, 24. - 26. september 2001, Portorož, Slovenija.* Ljubljana: IEEE Region 8, Slovenska sekcija IEEE, 2001, zv. B, str. 115-118. [COBISS.SI-ID 6542870]

7. BREST, Janez, ROŠKAR, Silvo, BOŠKOVIĆ, Borko, REBERNAK, Damijan, ŽUMER, Viljem. Algoritma za izračun sumarnih kosovnic v strojegradnji. V: ZAJC, Baldomir (ur.). *Zbornik desete Elektrotehniške in računalniške konference ERK 2001, 24. - 26. september 2001, Portorož, Slovenija.* Ljubljana: IEEE Region 8, Slovenska sekcija IEEE, 2001, zv. B, str. 127-130. [COBISS.SI-ID 6543638]

8. BOŠKOVIĆ, Borko, REBERNAK, Damijan, BREST, Janez, ŽUMER, Viljem. Preslikava E-R modela v objektno orientiran model. V: ZAJC, Baldomir (ur.). *Zbornik enajste mednarodne Elektrotehniške in računalniške konference ERK 2002, 23.-25. september 2002, Portorož, Slovenija.* Ljubljana: IEEE Region 8, Slovenska sekcija IEEE, [2002], zv. B, str. 63-66. [COBISS.SI-ID 7416854]

9. REBERNAK, Damijan, BREST, Janez, BOŠKOVIĆ, Borko, ŽUMER, Viljem. Načrtovanje in implementacija spletnega informacijskega sistema. V: ZAJC, Baldomir (ur.). *Zbornik enajste mednarodne Elektrotehniške in računalniške konference ERK 2002, 23.-25. september 2002, Portorož, Slovenija.* Ljubljana: IEEE Region 8, Slovenska sekcija IEEE, [2002], zv. B, str. 159-162. [COBISS.SI-ID 7426326]

10. BREST, Janez, ROŠKAR, Silvo, BOŠKOVIĆ, Borko, REBERNAK, Damijan, ŽUMER, Viljem. O pripadnosti materialov in polizdelkov v kosovni proizvodnji. V: ZAJC, Baldomir (ur.). *Zbornik enajste mednarodne Elektrotehniške in računalniške konference ERK 2002, 23.-25. september 2002, Portorož, Slovenija.* Ljubljana: IEEE Region 8, Slovenska sekcija IEEE, [2002], zv. B, str. 187-190. [COBISS.SI-ID 7427606]

11. GREINER, Sašo, REBERNAK, Damijan, BOŠKOVIĆ, Borko, BREST, Janez, ŽUMER, Viljem. Web information system based on open source technologies. V: BUDIN, Leo (ur.), LUŽAR - STIFFLER, Vesna (ur.), BEKIĆ, Zoran (ur.), HLJUZ DOBRIĆ, Vesna (ur.). 25th International Conference on Information Technology Interfaces, June 16-19, 2003, Cavtat, Croatia. *ITI 2003 : proceedings of the 25th International Conference on Information Technology Interfaces, June 16-19, 2003, Cavtat, Croatia.* Zagreb: University of Zagreb, SRCE University Computing Centre, 2003, str. 137-142. [COBISS.SI-ID 7991318]

12. REBERNAK, Damijan, LENIČ, Mitja, KOKOL, Peter, ŽUMER, Viljem. Finding boundary subjects for medical decision support with support vector machines. V: KROL, Marina (ur.), MITRA, Sunanda (ur.), LEE, D. J. (ur.). Sixteenth IEEE symposium on computer-based medical systems [also] CBMS 2003, 26-27 June 2003, New York, New York. *Proceedings.* Los Alamitos: IEEE Computer society, 2003, str. 385-390. [COBISS.SI-ID 8110870]

13. BREST, Janez, ROŠKAR, Silvo, BOŠKOVIĆ, Borko, REBERNAK, Damijan, GREINER, Sašo, KRUŠEC, Robert, ŽUMER, Viljem. Informacijski sistem v proizvodnem procesu. V: ZAJC, Baldomir (ur.). *Zbornik dvanajste mednarodne Elektrotehniške in računalniške konference ERK 2003, 25. - 26. september 2003, Ljubljana, Slovenija*, (Zbornik ... Elektrotehniške in

računalniške konference ERK ..., 1581-4572). Ljubljana: IEEE Region 8, Slovenska sekcija IEEE, 2003, str. 365-368. [COBISS.SI-ID 8229910]

14. BOŠKOVIĆ, Borko, REBERNAK, Damijan, GREINER, Sašo, BREST, Janez, ŽUMER, Viljem. Nadzorovanje virov računalniškega sistema v operacijskem sistemu Linux. V: ZAJC, Baldomir (ur.). *Zbornik dvanajste mednarodne Elektrotehniške in računalniške konference ERK 2003, 25. - 26. september 2003, Ljubljana, Slovenija*, (Zbornik ... Elektrotehniške in računalniške konference ERK ..., 1581-4572). Ljubljana: IEEE Region 8, Slovenska sekcija IEEE, 2003, str. 397-400. [COBISS.SI-ID 8233494]

15. KOSAR, Tomaž, REBERNAK, Damijan, MERNIK, Marjan, ŽUMER, Viljem. Visual language framework for LISA. V: LUŽAR - STIFFLER, Vesna (ur.), HLJUZ DOBRIĆ, Vesna (ur.). *ITI 2004 : proceedings of the 26th International Conference on Information Technology Interfaces, June 7-10, 2004, Cavtat, (Dubrovnik), Croatia*, (IEEE Catalog, No. 04EX794). Zagreb: University of Zagreb, SRCE University Computing Centre, 2004, str. 395-400. [COBISS.SI-ID 8808470]

16. REBERNAK, Damijan, ČREPINŠEK, Matej, MERNIK, Marjan. Spletna storitev za generiranje prevajalnika. V: ZAJC, Baldomir (ur.), TROST, Andrej (ur.). *Zbornik štirinajste mednarodne Elektrotehniške in računalniške konference ERK 2005, 26. - 28. september 2005, Portorož, Slovenija*, (Zbornik ... Elektrotehniške in računalniške konference ERK ...). Ljubljana: IEEE Region 8, Slovenska sekcija IEEE, 2005, zv. B, str. 15-18. [COBISS.SI-ID 9876758]

17. REBERNAK, Damijan, ZORMAN, Milan, LENIČ, Mitja. Določanje mejne odločitvene poti z uporabo metode podpornih vektorjev. V: ZAJC, Baldomir (ur.), TROST, Andrej (ur.). *Zbornik štirinajste mednarodne Elektrotehniške in računalniške konference ERK 2005, 26. - 28. september 2005, Portorož, Slovenija*, (Zbornik ... Elektrotehniške in računalniške konference ERK ...). Ljubljana: IEEE Region 8, Slovenska sekcija IEEE, 2005, zv. B, str. 146-149. [COBISS.SI-ID 9871126]

18. REBERNAK, Damijan, ČREPINŠEK, Matej, MERNIK, Marjan. Web service for designing and implementing formal languages. V: LUŽAR - STIFFLER, Vesna (ur.), HLJUZ DOBRIĆ, Vesna (ur.). 28th International Conference on Information Technology Interfaces, June 19-22, 2006, Cavtat/Dubrovnik, Croatia. *ITI 2006 : proceedings of the 28th International Conference on Information Technology Interfaces, June 19-22, 2006, Cavtat/Dubrovnik, Croatia*, (IEEE Catalog, No. 06EX1244). Zagreb: University of Zagreb, SRCE University Computing Centre, cop. 2006, str. 463-468. [COBISS.SI-ID 10547222]

19. REBERNAK, Damijan, MERNIK, Marjan, HENRIQUES, Pedro Rangel, DA CRUZ, Daniela, VARANDO PEREIRA, Maria João. Specifying languages using aspect-oriented approach: AspectLISA. V: LUŽAR - STIFFLER, Vesna (ur.), HLJUZ DOBRIĆ, Vesna (ur.). 28th International Conference on Information Technology Interfaces, June 19-22, 2006, Cavtat/Dubrovnik, Croatia. *ITI 2006 : proceedings of the 28th International Conference on Information Technology Interfaces, June 19-22, 2006, Cavtat/Dubrovnik, Croatia*, (IEEE Catalog, No. 06EX1244). Zagreb: University of Zagreb, SRCE University Computing Centre, cop. 2006, str. 695-700. [COBISS.SI-ID 10547478]

20. REBERNAK, Damijan, LENIČ, Mitja, MERNIK, Marjan. Teaching compiler construction using LISA tool. V: AUER, Michael E. (ur.). *Lifelong and blended learning*. Wien: International Association of Online Engineering, 2006, [4] f. [COBISS.SI-ID 10710806]

21. REBERNAK, Damijan, MERNIK, Marjan, HENRIQUES, Pedro Rangel, VARANDO PEREIRA, Maria João. AspectLISA: an aspect-oriented compiler construction system based on attribute grammars. V: BOYLAND, John (ur.). *Proceedings of the Sixth Workshop on language descriptions, tools and applications* [*also*] *LDTA 2006*, (Electronics notes in theoretical computer science, Vol. 164, no. 2). [S. l.: s. n.], 2006, 2006, vol. 164, issue 2, str. 37-53. http://dx.doi.org/10.1016/j.entcs.2006.10.003. [COBISS.SI-ID 10823446]

22. REBERNAK, Damijan, MERNIK, Marjan. A tool for compiler construction based on aspect-oriented specifications. V: *COMPSAC 2007 : pro-*

ceedings of the thirty-first annual international computer software and applications conference, 23-27 July, Beijing, China. Piscataway: IEEE, 2007, str. 11-16. [COBISS.SI-ID 11536406]

1.13 Objavljeni povzetek strokovnega prispevka na konferenci

23. BREST, Janez, REBERNAK, Damijan, BOŠKOVIĆ, Borko, ŽUMER, Viljem. Uporaba Linuxa pri gradnji informacijskega sistema v podjetju = Enterprise information system with the use of Linux. V: 1. mednarodni festival KIBLIX 2002 - IT Linux festival, 7., 8., in 9. november 2002, Maribor, Slovenija. *Katalog.* Maribor: Multimedia center Kibla, 2002, str. 34-36. [COBISS.SI-ID 7513110]

MONOGRAFIJE IN DRUGA ZAKLJUČENA DELA

2.05 Drugo učno gradivo

24. ŽUMER, Viljem, MERNIK, Marjan, BREST, Janez, KRUŠEC, Robert, LENIČ, Mitja, AVDIČAUŠEVIĆ, Enis, REBERNAK, Damijan, VREŽE, Aleksander. *Programski jezik C : gradivo za tečaj: začetni.* Maribor: Fakulteta za elektrotehniko, računalništvo in informatiko, Laboratorij za računalniške arhitekture in jezike, 1999. 1 zv. (loč. pag.). [COBISS.SI-ID 4988182]

25. ŽUMER, Viljem, MERNIK, Marjan, BREST, Janez, KRUŠEC, Robert, AVDIČAUŠEVIĆ, Enis, REBERNAK, Damijan, BOŠKOVIĆ, Borko. *Programski jezik C : gradivo za tečaj: nadgradnja osnov C-N-11-01.* Maribor: Fakulteta za elektrotehniko, računalništvo in informatiko, Laboratorij za računalniške arhitekture in jezike, 2001. 1 zv. (loč. pag.). [COBISS.SI-ID 6794006]

26. ŽUMER, Viljem, MERNIK, Marjan, BREST, Janez, KRUŠEC, Robert, LENIČ, Mitja, AVDIČAUŠEVIĆ, Enis, REBERNAK, Damijan, BOŠKOVIĆ, Borko. *Programski jezik C++ : gradivo za tečaj.* Maribor: Fakulteta za elektrotehniko, računalništvo in informatiko, Laboratorij za računalniške

arhitekture in jezike, 2001. 218 f. [COBISS.SI-ID 6054934]

27. ŽUMER, Viljem, MERNIK, Marjan, BREST, Janez, KRUŠEC, Robert, ČREPINŠEK, Matej, KOSAR, Tomaž, GREINER, Sašo, REBERNAK, Damijan, BOŠKOVIĆ, Borko, TUTEK, Simon. *Operacijski sistem LINUX/UNIX : gradivo za nadaljevalni tečaj*, (Računalniško opismenjevanje MŠZŠ). Maribor: Laboratorij za računalniške arhitekture in jezike in Center za programske jezike, 2003. 48 f. [COBISS.SI-ID 8160790]

28. ŽUMER, Viljem, MERNIK, Marjan, BREST, Janez, KRUŠEC, Robert, ČREPINŠEK, Matej, KOSAR, Tomaž, GREINER, Sašo, REBERNAK, Damijan, BOŠKOVIĆ, Borko. *Operacijski sistem LINUX/UNIX : gradivo za začetni tečaj*, (Računalniško opismenjevanje MŠZŠ). Maribor: Laboratorij za računalniške arhitekture in jezike in Center za programske jezike, 2003. 27 f. [COBISS.SI-ID 8160534]

2.11 Diplomsko delo

29. REBERNAK, Damijan. *Izdelava programskega paketa za vodenje proizvodnega procesa : diplomsko delo visokošolskega strokovnega študija*, (Fakulteta za elektrotehniko, računalništvo in informatiko, Diplomska dela visokošolskega strokovnega študija). Maribor: [D. Rebernak], 2001. XI, 72 f., ilustr. [COBISS.SI-ID 6092310]

30. REBERNAK, Damijan. *Tehnologije za hiter razvoj strežniških javanskih zrn : diplomsko delo univerzitetnega študijskega programa*, (Fakulteta za elektrotehniko, računalništvo in informatiko, Diplomska dela univerzitetnega študija). [Maribor]: [D. Rebernak], [2003]. XII, 82 f., ilustr. [COBISS.SI-ID 8093206]

2.12 Končno poročilo o rezultatih raziskav

31. ŽUMER, Viljem, KOKOL, Peter, GUID, Nikola, KUTNJAK, Milan, ŽALIK, Borut, OJSTERŠEK, Milan, ZAZULA, Damjan, MERNIK, Marjan, KORŽE, Danilo, KVAS, Aleksander, KOROŠEC, Dean, PODGORELEC,

David, KOLMANIČ, Simon, KRUŠEC, Robert, NOVAK, Uroš, CIGALE, Boris, REBERNAK, Damijan, GORENJAK, Borut, HLEB BABIČ, Špela, POTOČNIK, Božidar, BREST, Janez, PODGORELEC, Vili, ZORMAN, Milan, LENIČ, Mitja, STRNAD, Damjan, ŠPROGAR, Matej, GOMBOŠI, Matej. *Računalniški sistemi, metodologije in kibernetika : letno poročilo o rezultatih raziskovalnega programa za leto 1999* [*za MZT Slovenije*]. Maribor: Fakulteta za elektrotehniko, računalništvo in informatiko, 2000. 1 zv. (loč. pag.). [COBISS.SI-ID 5974806]

32. ŽUMER, Viljem, KOKOL, Peter, OJSTERŠEK, Milan, MERNIK, Marjan, BREST, Janez, KVAS, Aleksander, KRUŠEC, Robert, NOVAK, Uroš, ZORMAN, Milan, PODGORELEC, Vili, REBERNAK, Damijan. *Avtomatska implementacija programskih jezikov na heterogenih računalniških sistemih : letno poročilo o rezultatih raziskovalnega projekta za leto 2000.* Maribor: Fakulteta za elektrotehniko, računalništvo in informatiko, 2001. 1 zv. (loč. pag.). [COBISS.SI-ID 6159638]

33. ŽUMER, Viljem, KOKOL, Peter, GUID, Nikola, KUTNJAK, Milan, ŽALIK, Borut, OJSTERŠEK, Milan, ZAZULA, Damjan, MERNIK, Marjan, KORŽE, Danilo, KVAS, Aleksander, KOROŠEC, Dean, PODGORELEC, David, BREST, Janez, KOLMANIČ, Simon, KRUŠEC, Robert, NOVAK, Uroš, CIGALE, Boris, REBERNAK, Damijan, GORENJAK, Borut, HLEB BABIČ, Špela, POTOČNIK, Božidar, PODGORELEC, Vili, LENIČ, Mitja, ŠPROGAR, Matej, ZORMAN, Milan, STRNAD, Damjan, GOMBOŠI, Matej. *Računalniški sistemi, metodologije in kibernetika : letno poročilo o rezultatih raziskovalnega programa za leto 2000.* Maribor: Fakulteta za elektrotehniko, računalništvo in informatiko, 2001. 1 zv. (loč. pag.). [COBISS.SI-ID 6192662]

34. ŽUMER, Viljem, BREST, Janez, REBERNAK, Damijan, BOŠKOVIĆ, Borko. *Programska oprema za nadzor in uporavljanje procesa proizvodnje : poročilo za leto 2001 : projekt za podjetje Lestro Ledinek Engineering d.o.o.*. Maribor: Fakulteta za elektrotehniko, računalništvo in informatiko, 2002. 33 f. [COBISS.SI-ID 6909718]

35. ŽUMER, Viljem, KOKOL, Peter, GUID, Nikola, KUTNJAK, Milan, ŽALIK, Borut, OJSTERŠEK, Milan, ZAZULA, Damjan, MERNIK, Mar-

jan, KVAS, Aleksander, KOROŠEC, Dean, HLEB BABIČ, Špela, PODGORELEC, David, KOLMANIČ, Simon, PODGORELEC, Vili, ZORMAN, Milan, LENIČ, Mitja, KRUŠEC, Robert, NOVAK, Uroš, ŠPROGAR, Matej, CIGALE, Boris, GOMBOŠI, Matej, REBERNAK, Damijan, GORENJAK, Borut, BREST, Janez, LIPUŠ, Bogdan, KRIVOGRAD, Sebastian, HORVAT, Branko, HOLOBAR, Aleš, DIVJAK, Matjaž, POTOČNIK, Božidar. *Računalniški sistemi, metodologije in kibernetika : letno poročilo o rezultatih raziskovalnega programa za leto 2001* [*za MŠZŠ*]. Maribor: Fakulteta za elektrotehniko, računalništvo in informatiko, 2000. 1 zv. (loč. pag.). [COBISS.SI-ID 7008790]

36. BREST, Janez, GREINER, Sašo, REBERNAK, Damijan, BOŠKOVIĆ, Borko, TUTEK, Simon, ŽUMER, Viljem. *Informacijski sistem za projektno vodenje proizvodnje v strojegradnji : končno poročilo*. Maribor: Fakulteta za elektrotehniko, računalništvo in informatiko, 2005. 8 f., [25] f. [COBISS.SI-ID 9460502]

37. BREST, Janez, BOŠKOVIĆ, Borko, GREINER, Sašo, REBERNAK, Damijan, ŽUMER, Viljem. *Izdelava programske opreme za nadzor in upravljanje dela procesa proizvodnje : končno poročilo*. Maribor: Fakulteta za elektrotehniko, računalništvo in informatiko, 2005. 5 f. [COBISS.SI-ID 10210838]

2.25 Druge monografije in druga zaključena dela

38. ŽUMER, Viljem, MERNIK, Marjan, BREST, Janez, KRUŠEC, Robert, LENIČ, Mitja, AVDIČAUŠEVIĆ, Enis, REBERNAK, Damijan, BOŠKOVIĆ, Borko, VREŽE, Aleksander. *Operacijski sistem LINUX-UNIX : nadaljevalni*, (Računalniško opismenjevanje). Ljubljana: Ministrstvo za šolstvo in šport: Zavod Republike Slovenije za šolstvo, 2000. 72 f. [COBISS.SI-ID 5400342]

39. ŽUMER, Viljem, MERNIK, Marjan, BREST, Janez, KRUŠEC, Robert, AVDIČAUŠEVIĆ, Enis, REBERNAK, Damijan, BOŠKOVIĆ, Borko. *Operacijski sistem LINUX/UNIX : gradivo za nadaljevalni tečaj*, (Računalniško opismenjevanje). Maribor: Ministrstvo za šolstvo in šport: Zavod Republike

Slovenije za šolstvo, 2001. 81 f. [COBISS.SI-ID 6716694]

40. ŽUMER, Viljem, BREST, Janez, MERNIK, Marjan, ČREPINŠEK, Matej, KRUŠEC, Robert, KOSAR, Tomaž, REBERNAK, Damijan, BOŠKOVIĆ, Borko, TUTEK, Simon, GERLIČ, Goran, GREINER, Sašo. *LINUX in Qt : tečaj za AMES, Brezovica pri Ljubljani*. Maribor: Fakulteta za elektrotehniko, računalništvo in informatiko, Laboratorij za računalniške arhitekture in jezike, 2003. 1 mapa (loč. pag.). [COBISS.SI-ID 7801110]

IZVEDENA DELA (DOGODKI)

3.14 Predavanje na tuji univerzi

41. REBERNAK, Damijan. *AspectLISA: an aspect-oriented compiler construction system based on attribute grammars : invited talk on Department of Computing at Macquarie University, Sydney, Australia from July 10 to July 14, 2006*. New South Wales, 2006. [COBISS.SI-ID 10602262]

SEKUNDARNO AVTORSTVO

Pisec recenzij

42. *IET software*. Rebernak, Damijan (pisec recenzij 2007). Stevenage: IET. ISSN 1751-8814. [COBISS.SI-ID 11616790]

43. *Science of computer programming*. Rebernak, Damijan (pisec recenzij 2006). [Print ed.]. Amsterdam: North-Holland. ISSN 0167-6423. [COBISS.SI-ID 26367488]