VYTAUTO DIDŽIOJO UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS INSTITUTAS

Vaidas Giedrimas

KOMPONENTINIŲ PROGRAMŲ SISTEMŲ GENERAVIMO METODAS

Daktaro disertacija

Fiziniai mokslai (P 000)
Informatika (09 P)
Informatika, sistemų teorija (P 175)

Vilnius, 2010

Disertacija rengta 2002 – 2006 metais Matematikos ir informatikos institute.

Disertacija ginama eksternu.

Mokslinė konsultantė

doc. dr. Audronė Lupeikienė

(Matematikos ir informatikos institutas, fiziniai mokslai, informatika – 09 P)

Padėka

Už nuolatinį palaikymą, kantrybę ir supratimą norėčiau padėkoti savo šeimai ir visų pirma žmonai Linai.

Už vertingas mokslines konsultacijas ir diskusijas, skatinusias tobulėti, už pastabas ir patarimus, padėjusius pagerinti šio darbo kokybę nuoširdžiai dėkoju mokslinei vadovei Audronei Lupeikienei, profesoriui Albertui Čaplinskui ir visam Matematikos ir informatikos instituto Programų sistemų inžinerijos skyriui, gerbiamoms recenzentėms Aidai Pliuškevičienei ir Dalei Dzemydienei, kolegoms iš Vilniaus universiteto ir Šiaulių universiteto.

Už suteiktą vertingą informaciją dėkoju ir kolegoms iš užsienio: *J. Craig Cleaveland*, *Margus Veanes*, *Vahur Kotkas*, *Ando Saabas*, *Jaan Penjam*, *Enn Tyugu* bei Estijos Kibernetikos instituto darbuotojams.

Dėkoju Lietuvos valstybiniam mokslo ir studijų fondui bei Šiaulių universitetui už suteiktą finansinę paramą studijų metu.

Galiausiai norėčiau padėkoti visiems asmenis rengusiems respublikines ar tarptautines konferencijas ir kitaip prisidėjusiems prie šios disertacijos rezultatų skelbimo.

Turinys

Lentelių sąrašas

Iliustracijų sąrašas

Sutrumpinimų sąrašas

ADL	Architektūros aprašymo kalba.
ATP	Automatinio teoremų įrodymo programa (angl. *prover*).
BMTP	*R.S. Boyer* ir *J.S. Moore* sukurta teoremų įrodymo programa naudojanti induktyvinį metodą.
BPEL	Pasaulinio tinklo paslaugų sąveikos aprašymo kalba (angl. *Business Process Execution Language*).
CCM	CORBA pagrindu OMG konsorciumo sukurtas komponento modelis (angl. *CORBA Component Model*).
CIP	Programų sintezės sistema, realizuojanti metaprogramavimo metodą.
CoC	konstrukcijų skaičiavimas (angl. *Calculus of Constructions*), naudotas *Coq* sistemoje.
COM	Microsoft komponentinė technologija.
CORBA	Vienas iš OMG konsorciumo palaikomų komponento modelių (angl. *Calculus of Constructions*).
CSP	Formalioji specifikavimo kalba lygiagretiesiems procesams specifikuoti angl. (*Communicating sequential processes*) [40].
CTT	Skaičiuojamoji tipų teorija.

CWB	*Component WorkBeanch* programų sintezės sistema.
DB	Duomenų bazė.
DB	Duomenų bazių valdymo sistema.
DAML-S	Ontologijų aprašymo kalba.
FP	Funkcinių taškų skaičius.
EJB	Vienas iš Sun Microsotems komponento modelių (angl. *Enterprise Java Beans*)[157].
ESSP	Išplėstasis SSP metodas (angl. *Extended SSP*).
FM	Formalieji metodai.
HOPE	Programų sintezės sistema, realizavusi metaprogramavimo metodą.
ITS	Intuicionistinis teiginių skaičiavimas (angl. *intuitionistic propositional calculus*).
JPK	Jungiantysis programinis kodas (angl. *glue-code*).
KK	Komponentinis karkasas (angl. *framework*), vykdymo aplinka.
KPS	Komponentinė programų sistema.
KPSGC	Komponentinės programų sistemos gyvavimo ciklas.
KVK	Klasių veiksmų (angl. *Class actions*) aprašymo kalba, Vykdomosios UML dalis.
.NET	*Microsoft* komponentinė technologija.
LCF	Skaičiuojamųjų funkcijų logika.
LoC	Kodo eilučių skaičius (angl. *lines of code*).
LTT	Loginė tipų teorija.

MDA	Modelinė programų sistemų architektūra (angl. *Model-driven architecture*).
ML	Funkcinio programavimo kalba.
MOP	Metaobjektų protokolas (angl. *Metaobject protocol*).
NuPRL	Viena pirmųjų programų sintezės sistemų.
NUT	rus. *Novyj UTOPIST* – SSP metodą realizuojanti objektinių programų sintezės sistema.
OCL	Objektų ribojimų specifikavimo kalba (angl. *Object Constraint language*) apibrėžta UML standarte.
OWL-S	Ontologijų aprašymo kalba.
PECOS	2000–2002 metais vykdytas projektas *Pervasive Component Systems (IST-1999-20398)* arba projekte naudotas komponento modelis.
PIM	Abstraktusis programos modelis MDA architektūroje (angl. Platform-independent model).
PKM	Disertacijos autoriaus siūlomas Programinio komponento modelis, apibendrinantis kitus komponento modelius.
PRIZ	1975–1977 m. pradėta kurti programų sintezės sistema, realizuojanti SSP metodą.
PSM	Konkretusis programos modelis MDA architektūroje (angl. *platform-specific model*).
PTP	Pasaulinio tinklo paslaugos (angl. *web-services*).
QoS	angl. *Quality of service* – kokybės užtikrinimas.
RAPTS	Programų sintezės sistema, realizavusi išplėstojo kompiliavimo metodą [27].

SCA	angl. *Servise Component Architecture* – vienas iš komponento modelių realizuojantis paslauginę architektūrą.
SETL	Programų sintezės sistema, realizavusi išplėstojo kompiliavimo metodą.
SOFA	Architektūros aprašymo kalba.
SML	Bendros paskirties modulinė funkcinio programavimo kalba.
SNARK	Teoremų įrodymo programa [156], naudota *DEDALUS* ir *Amphion* sistemose.
SP	Stipriausia „prieš“ sąlyga (angl. *strongest precondition*).
SPP	Struktūrinės programų sintezės metodas.
TAMPR	Programų sintezės sistema, realizavusi išplėstojo kompiliavimo metodą [25].
UCM	Architektūriniai ruošiniai (angl. *Use Case Maps*) naudojami programinių produktų šeimoms projektuoti ir realizuoti.
UMM	angl. *Unified Metamodel* – projekto *UniFrame* metu sukurtas komponento modelis.
UML	angl. *Unified Modeling Langugage* – OMG konsorciumo palaikoma verslo ir programinės įrangos modeliavimo kalba.
UTOPIST	PRIZ sistemoje naudota specifikavimo kalba.
WP	Silpniausia „prieš“ sąlyga (angl. *Weakest precondition*).
VA	Komponentų ir jų sistemų vykdymo aplinka.

WSDL	Pasaulinio tinklo paslaugų aprašymo kalba (angl. *Web Services Description Language*).
WSDL	Pasaulinio tinklo paslaugų komponavimo aprašymo kalba (angl. *Web Services Flow Language*), pasiūlyta *IBM*.
XML	Universali duomenų struktūrų aprašymo kalba (angl. *Extensible Markup language*).
XSLT	angl. *XSL transformations* – XML formato duomenų transformacijų aprašymo kalba.
ZAP	Programų sintezės sistema, realizavusi metaprogramavimo metodą.
ŽB	Žinių bazė.

Sąvokų žodynėlis

Interfeisas	Komponento elgsenos abstrakcija, gaunama slepiant komponento operacijas nuo išorės. Skiriama paties komponento elgsena (teikiamas interfeisas) ir išorinės komponento aplinkos elgsena (reikalaujamas interfeisas).
Įrodomasis programavimas	Automatizuoto programų kūrimo metodų klasė (angl. *proofs-as-programs*), kuriai priklausantiems metodams būdingas formalus uždavinio aprašymas loginės teorijos terminais ir programos gavimas naudojant teoremų įrodymo įrankį.
Generavimas	Automatizuoto programų kūrimo būdas, dažniausiai susietas ir su apibendrintų ruošinių naudojimu.
Generatorius	Programa arba jos dalis, atliekanti konkrečius programų generavimo proceso uždavinius.
Kvalifikacija	Komponentą arba jo sudėtinius elementus charakterizuojanti savybė (angl. *credential*). Paprastai tokios savybės naudojamos komponentų patikimumui, veikimo spartai ir kt. nefunkciniams reikalavimams išreikšti.
Komponavimas	Komponentų jungimas tarpusavyje ir (arba) su kitomis komponento modelyje apibrėžtomis esybėmis (pvz.: komponentiniu karkasu) siekiant gauti komponentinę sistemą.
Komponentas	Komponavimo vienetas su kontraktu specifikuotais interfeisais, atitinkantis tam tikrą komponento modelį.

Komponento modelis	Sąvokų ir taisyklių, nusakančių komponentą, jo dalis ir jo naudojimui reikalingas paslaugas, visuma.
Kontraktas	Komponentų tarpusavio komunikavimo abstrakcija, apibrėžianti veiksmų su komponentu sekas, bei galimą šių veiksmų įtaką komponento būsenoms (jei tokios stebimos). Skiriamas komponento kontraktas, nusakantis komponento teikiamas ir reikalaujamas paslaugas, ir sąveikos kontraktas, nusakantis kokius vaidmenis komponentas atlieka sistemoje.
Prievadas	(angl. *port*) Komponento sąveikos su išore abstrakcija. Prievadu laikomas komponento interfeisas, įvykio šaltinis, įvykio jutiklis ir iš išorės pasiekiamas atributas.
Sintezė	Automatizuoto programų kūrimo būdas, dažniausiai susietas su sudedamųjų dalių jungimą į visumą. Šioje disertacijoje terminas *sintezė* naudojamas ir kaip termino *generavimas* sinonimas.

1 skyrius

Įvadas

1.1. Tyrimų sritis ir problemos aktualumas

Šiuolaikinis programinės įrangos kūrimo procesas yra kritinis tiek laiko, tiek kokybės požiūriu. Programų sistemos priklauso nuo dinamiškai besikeičiančio verslo, todėl turi būti paprastai modifikuojamos ir lengvai aptarnaujamos. Siekiant šio tikslo siūlomos vis naujos programų sistemų kūrimo paradigmos ir metodikos.

Programų sistemų inžinerijai praėjus struktūrinės ir objektinės paradigmos raidos etapus pastebėta, kad norint padidinti programų kūrimo metodų veiksmingumą, būtina kurti programas pakartotinai naudojant kiek įmanoma didesnės apimties (angl. *granularity*) modulius [115].

Didžiausios apimties „juodosios dėžės“ tipo modulius siūlo komponentinė paradigma. Komponentinių programų sistemų inžinerija įgalina sumažinti laiko ir kitų programų kūrimui bei testavimui būtinų išteklių poreikį. Griežtas turinių atskyrimo principo laikymasis palengvina tokių sistemų aptarnavimo procesą. Tačiau ir komponentinė paradigma negali užtikrinti reikiamos programinės įrangos kokybės, todėl būtini nauji komponentinių programų sistemų kūrimo metodai.

Praktinėms programų sistemų inžinerijos problemoms spręsti ne kartą siūlyti automatinio ir automatizuoto programavimo metodai. Automatinio programavimo terminas pradėtas vartoti ne vėliau kaip 1954 m., kai buvo norima nusakyti pirmuosius *Fortran* kalbos kompiliatorius [17]. Iš pradžių programų kūrimo automatizavimas buvo suprantamas kaip kompiliavimo problema, kai formali specifikacija kompiliuojama siekiant gauti vykdomą programą [27, 25, 53]. Vėlesniuose šaltiniuose šis terminas jau apima ir specifikacijų sudarymą, ir teisingumo užtikrinimą, ir projektinius sprendinius, aukšto abstrakcijos lygmens specifikacijas verčiant į žemesnio abstrakcijos lygmens specifikacijas, ir pan. Taigi, automatinis kompiliavimas ilgainiui išsivystė į automatizuoto pro-

gramavimo paradigmą ir automatizuotą programų sistemų inžineriją (angl. *automated software engineering*).

Per pastaruosius 20 metų daugėja programų gavimo generavimo metu idėjos realizacijų, kuriami ir plačiai naudojami generavimo metodai [143, 59, 43, 60, 69, 81]. Skiriamos kelios automatizuoto programų sistemų kūrimo kryptys priklausomai nuo naudojamos šio uždavinio sampratos ir kūrimo modelio. Vis dėl to dažniausiai akcentuojamas programų kūrimo proceso automatizuotu būdu greitis, o ne rezultatų patikimumas. Šis akcentas pastebimas ir egzistuojančiose komponentinių programų sistemų generavimo sistemose [150, 132].

Aukštesnį programų generavimo rezultatų patikimumą užtikrina įrodomojo programavimo (angl. *proofs-as-programs*) metodas [142, 90, 74]. Iš pradžių jį naudojant buvo kuriamos nesudėtingos programos, atliekančios skaičiavimus pagal automatizavimo įrankio išvestas formules [170, 175, 142]. Vėliau buvo analizuojamos įvairių logikų ir tipų teorijų savybės, bei jų taikymo šiam metodui galimybės.

Per pastaruosius penkerius metus vėl pastebimas padidėjęs mokslininkų susidomėjimas automatizuoto programų kūrimo procesais naudojančiais teoremų įrodymus konstruktyviosiose logikose [142, 152]. Ypač akcentuojama tokių metodų svarba kuriant didelės apimties (angl. *industrial-scale*) programinę įrangą.

Pažangių programų generavimo metodų savybių pritaikymas komponentei paradigmai galėtų ne tik sumažinti kūrimo proceso trukmę bet ir užtikrinti kuriamos programinės įrangos kokybę.

Šios disertacijos tyrimų sritis yra komponentinių programų sistemų surinkimo iš komponentų proceso automatizavimas.

1.2. Tyrimų objektas

Disertacijos tyrimų objektas – programų sistemų automatizuoto surinkimo iš komponentų uždavinys.

1.3. Tyrimų tikslas ir uždaviniai

Tyrimų tikslas – sudaryti deduktyviosios sintezės metodą komponentinėms programoms kurti atsižvelgiantį į struktūrines komponentų savybes ir nepriklausantį nuo konkretaus komponento modelio iš nagrinėjamos komponentų klasės. Tikslui pasiekti sprendžiami šie uždaviniai:

- Ištirti programinių komponentų savybes ir komponentinių programų kūrimo proceso ypatumus;

- Išanalizuoti programų sintezės ir generavimo metodus įvertinant šių metodų taikymo galimybes komponentinėms programoms kurti;
- Pasiūlyti komponentinių programų surinkimo iš gatavų komponentų proceso automatizavimo metodą;
- Siekiant eksperimentiškai patikrinti komponentinių programų surinkimo proceso automatizavimo metodą sukurti ir įvertinti jį realizuojančios sistemos prototipą.

1.4. Ginamieji disertacijos teiginiai

- Programų sintezės uždavinys ir jo sprendimo būdas gali būti taip suformuluotas apibendrinto komponento modelio terminais, kad gautas sprendinys gali būti pritaikomas kiekvienam iš apibendrintų komponentų modelių panaudojant konkretizavimo, patikslinimo ir papildymo operacijas.
- *Curry-Howard* protokolą galima taikyti automatizuotam komponentinių programų kūrimui ir tokiu būdu gauti specifikaciją atitinkančias programas.

1.5. Tyrimo metodai

Analizuojant mokslinius ir eksperimentinius pasiekimus komponentinės programinės įrangos kūrimo srityje naudoti *informacijos paieškos*, *sisteminimo*, *analizės*, *lyginamosios analizės* ir *apibendrinimo* metodai.

Programinio komponento modeliui sudaryti naudoti *klasterizavimo* ir *ekspertinio vertinimo* metodai. Klasterizavimo tikslas – sudaryti komponento modelių grupes, stebėti, pagal kuriuos požymius modeliai galėtų būti grupuojami. Klasterizuota komponento modelių savybių ir komponento abstrakcijos lygmenų lenteles interpretuojant kaip kategorinių kintamųjų aibę. Atstumams matuoti parinktas *Euklido atstumo kvadrato* matas. Klasterizavimas atliktas statistinės analizės programa *SPSS 16.0*. Kadangi klasterizavimo metodas negarantuoja, jog yra apibendrinamos tapačios sąvokos, rezultatai patikslinti ekspertinio vertinimo metodu.

Realizuojant *Curry-Howard* protokolą komponentinei paradigmai naudotas *matematinio modeliavimo* metodas. Atliekant komponentinių sistemų generavimo metodo eksperimentinį vertinimą naudoti *eksperimento* ir *apibendrinimo* metodai.

1.6. Mokslinis naujumas

Darbe gauti šie nauji moksliniai rezultatai:

- Pasiūlyta, kaip abstrahuojantis nuo konkrečių komponentų modelių ir jų savybių, programų generavimą vykdyti naudojant abstračiojo programinio komponento modelio terminus;
- Siekiant užtikrinti komponentinių programų atitikimą specifikacijai, pasiūlyta naudoti *Curry-Howard* protokolą ir struktūrinės programų sintezės algoritmus automatizuotam programų kūrimui naudojant *programinio komponento modelį*;
- Numatyta kelių programų generavimo metodų integravimo galimybė.

1.7. Praktinė darbo reikšmė

Praktiniu požiūriu disertacijos rezultatai reikšmingi dėl šių priežasčių:

- Siūlomas metodas įgalins pagerinti komponentinių programų sistemų kokybę bei sutrumpinti jų kūrimo ir atnaujinimo laiką;
- Metodą realizuojanti SoCoSyS sistema gali būti tiesiogiai naudojama sistemoms generuoti iš .NET komponentų. Taip pat dėl metodo universalumo, SoCoSyS sistema yra lengvai pritaikoma įvairiems komponento modeliams.

1.8. Rezultatų aprobavimas ir publikavimas

Tyrimų rezultatai buvo pristatyti ir aptarti šiose respublikinėje mokslinėse konferencijose:

1. 2003 m. rugpjūčio 29 d. Vienuoliktojoje mokslinėje kompiuterininkų konferencijoje skaitytas pranešimas „Komponento modelis struktūrinės programų sintezės kontekste“,
2. 2003 m. birželio 20 d. respublikinėje LMD konferencijoje skaitytas pranešimas tema „Programų sistemų automatizuoto surinkimo iš gatavų komponentų uždavinys“,
3. 2003 m. birželio 18 d. respublikinėje LMD konferencijoje skaitytas pranešimas tema „ Komponento specifikacijos formalizavimas“,
4. 2004 m. birželio 18 d. XLV-ojoje respublikinėje LMD konferencijoje skaitytas pranešimas tema „Komponento specifikacijos formalizavimas“,

5. 2005 m. sausio 26 d. respublikinėje konferencijoje „Informacinės technologijos 2005“ skaitytas pranešimas tema „.NET komponento specifikacija programų sintezės kontekste“,

6. 2005 m. birželio 20 d. XLVI-ojoje respublikinėje LMD konferencijoje skaitytas pranešimas tema „Komponentinių programų struktūrinės sintezės teorinės problemos“,

7. 2006 m. birželio 20 d. XLVII-ojoje respublikinėje LMD konferencijoje skaitytas pranešimas tema „Induktyvinis metodas komponentinių programų sintezėje“,

8. 2007 m. rugpjūčio 14 d. Tryliktojoje mokslinėje kompiuterininkų konferencijoje skaitytas pranešimas „Generavimo metodų panaudojimas kuriant .NET komponentines programų sistemas“,

9. 2009 m. rugsėjo 26 d. Keturioliktoje mokslinėje kompiuterininkų konferencijoje skaitytas pranešimas „Modelinės architektūros naudojimas kuriant komponentines programu sistemas“.

Tarptautinėse mokslinėse konferencijose ir kituose renginiuose:

1. 2004 m. rugpjūčio 10 d. Tarptautinėje vasaros mokykloje ESSCSS'04 (Estonian Summer School in Computer and System Science) skaitytas pranešimas tema „Formal specification of .NET component“,

2. 2004m. spalio 14 d. XLV-ojoje Rygos Technikos universiteto tarptautinėje mokslinėje konferencijoje skaitytas pranešimas „Component model and its formalisation for structural synthesis“,

3. 2005 m. rugpjūčio 9 d. tarptautinėje vasaros mokykloje ESSCSS'05 (Estonian Summer School in Computer and System Science) skaitytas pranešimas tema „An application of SSP method to component-based development process“,

4. 2005 m. rugsėjo 27 d. tarptautinėje mokslinėje konferencijoje GPCE'05 (Generative Programming and Component Engineering) skaitytas pranešimas tema „Component-based Software Generation: The Structural Synthesis Approach“,

5. 2006 m. liepos 3 d. tarptautinėje mokslinėje konferencijoje Baltic DB&IS'06 skaitytas pranešimas tema „Architectures of Component-Based Structural Synthesis Systems“,

6. 2006 m. rugpjūčio 7 d. tarptautinėje vasaros mokykloje ESSCSS'06 (Estonian Summer School in Computer and System Science) skaitytas

pranešimas tema „Component-based Software Synthesis: the union of two methods“,

7. 2008 m. lapkričio 29 d. XIV-jame Lietuvių mokslo ir kūrybos simpoziume skaitytas pranešimas tema „Grid tinklo panaudojimas programines įrangos sintezei“.

Tyrimo rezultatai publikuoti **7** straipsniuose recenzuojamuose Lietuvos žurnaluose, **3** straipsniuose užsienio žurnaluose ir tarptautinių konferencijų darbuose bei **1** straipsnyje kituose Lietuvos leidiniuose. Išsamus publikacijų sąrašas pateiktas A priede.

1.9. Disertacijos struktūra

Disertaciją sudaro **6** skyriai **Pirmasis skyrius** yra įvadinis. Jame glaustai aprašoma tyrimų sritis ir aktualumas, tyrimo objektas, tikslas ir uždaviniai. Taip pat pateikiamas ir skaitytų pranešimų disertacijos tema sąrašas.

Antrajame skyriuje pristatoma tyrimo problema. Analizuojama komponento modelių ir komponento abstrakcijos lygmenų įvairovė. Aprašoma komponentų ir komponentinių programų sistemų kūrimo specifika, analizuojamas tokių programų sistemų kūrimo uždavinys, identifikuojami probleminiai komponentinių programų sistemų gyvavimo ciklo etapai.

Trečiasis skyrius – analitinis. Jame analizuojamos deduktyviosios, induktyviosios, transformacinės programų sintezės ir kiti generavimo metodai.

Ketvirtajame skyriuje aprašoma komponentinių programų sistemų generavimo metodika. Pirmiausia formalizuojamas programinio komponento modelis. Komponentinei paradigmai realizuojamas *Curry-Howard* protokolas, atskleidžiamas struktūrinės sintezės ir kitų programų sintezės metodų pritaikomumas komponentinėms sistemoms kurti automatizuotu būdu.

Penktasis skyrius – eksperimentinis. Jame aprašoma SoCoSys sistema ir jos darbo rezultatai generuojant komponentines programas iš konkrečių komponentų aibės. Sistema palyginta su kitomis komponentinių programų kūrimui skirtomis sistemomis.

Šeštajame skyriuje pateikiamos išvados.

2 skyrius

Komponentinių programų sistemų kūrimo proceso samprata ir modeliai

Būtina sėkmingo programų sistemų kūrimo proceso prielaida yra aiškus probleminės srities identifikavimas. Šio skyriaus tikslas – ištirti programinių komponentų savybes ir komponentinių programų kūrimo proceso ypatumus.

Pirmiausia (2.1. poskyryje) supažindinama su komponento sąvoka, analizuojama komponento modelių įvairovė. 2.2. poskyryje aprašomi komponento abstrakcijos lygmenys.

Komponentinių programų sistemų inžinerijoje skiriamos dvi šakos: komponentų kūrimas ir komponentinių sistemų kūrimas. Kiekviena šių šakų yra pristatoma atskiruose poskyriuose (2.3. ir 2.4. atitinkamai), ypač daug dėmesio skiriant sistemų kūrimui. Aprašoma komponentų ir komponentinių programų sistemų kūrimo specifika, analizuojamas tokių programų sistemų kūrimo uždavinys, identifikuojami probleminiai komponentinių programų sistemų gyvavimo ciklo etapai.

2.1. Komponento modeliai

Komponento modelis apibrėžia komponento struktūrines ir dinamines savybes, aprašo komponento realizavimo, komponavimo, diegimo standartus [187]. Paprastai kiekviena koponentinė technologija remiasi vis kitu komponento modeliu. Komponento modelių įvairovė pastebima ir akademiniuose komponentinių programų sistemų kūrimo metoduose. Šio skyriaus poskyriuose nagrinėjami konkretūs komponento modeliai siekiant identifikuoti jų panašumus ir skirtumus. Dauguma nagrinėtų programinio komponento modelių poskyriuose įvardijami juos sukūrusių mokslininkų grupių vardais. Taip daroma dėl šių

priežasčių:

- dauguma komponento modelių neturi konkrečių pavadinimų, ar registruotų prekinių ženklų (angl. *trademark*),
- skirtingus komponento modelius charakterizuojančių savybių aibė yra pernelyg plati, kad ją būtų galima glaustai įvardinti poskyrio antraštėje.

2.1.1. *Szypersky* modelis

Plačiausiai naudojamos programinio komponento apibrėžties autorius *C. Szypersky* komponentą apibrėžia taip [159]:

„Programinis komponentas – komponavimo vienetas turintis kontraktais specifikuotus interfeisus bei aiškią priklausomybę nuo konteksto. Programinis komponentas gali būti paskirstytas nepriklausomai ir yra skirtas „trečiųjų šalių“ naudojimui.“

C. Szypersky apibendrindamas daugiausia komercinius komponento modelius (tokius kaip COM, CORBA, JavaBeans) išskiria šiuos komponento bruožus [158, 159]:

- Komponentai skirstomi į elementarius ir sudėtinius. Elementarus komponentas (pvz., Java JAR, .NET Assembly) apima dvi dalis: *modulį* ir *resursus*. Sudėtiniai komponentai sudaryti iš elementarių ir (arba) sudėtinių komponentų.
- Komponentai gali būti skirstomi ir atsižvelgiant į jų nepriklausomumą. *Lengvu komponentu* laikomas daug priklausomybių turintis komponentas. Tuo tarpu *sunkus komponentas* yra beveik arba visiškai nepriklausomas.
- Vykdymo metu iš tiesų naudojami objektai, o ne komponentai. Konstruojant sudėtinius komponentus, faktiškai sukuriamas vidinių komponentų sujungimo grafas, pagal kurį vykdymo metu (*run-time*) yra formuojamas sudėtinis komponentas. Tačiau kiekvieną vykdymo iteraciją gaunamas vis kitas konkretus sudėtinis komponentas.
- Priklausomybių tarp komponentų grafas privalo būti be ciklų.
- Komponentai neturi išsaugomos ilgalaikės būsenos.
- Komponentai gali komunikuoti dviem būdais: tiesiogiai per įvykių mechanizmą arba per tarpinę programinę įrangą (*CORBA ORB*, *.NET Framework* ir pan.).

Konkrečių komponento modelių abstrakcijos lygmenų *C. Szypersky* išreikštiniu būdu neišskiria, daugiausiai koncetruojamasi į komponentinio objekto lygmenį.

2.1.2. *UNU/IIST* modelis

Jungtinių Tautų Universiteto (*United Nations University*) Tarptautinis programinės įrangos technologijų institutas (*International Institute of Software Technology*) nuo 2000 metų vykdė projektą *II/1/1/5 Formal Methods for Object and Component Systems*, kurio tikslas – pasiūlyti formaliuosius metodus objektinėms ir komponentinėms sistemoms kūrti [174].

S. Meng ir B. K. Aicherning [119, 120] komponentus formaliai apibrėžia kaip koalgebras, nes koalgebrų teorija tinka dinaminėms sistemoms aprašyti, kai sistemos būsenų erdvė yra paslėpta ir gali būti stebima tik iš išorės. Todėl komponentas suprantamas kaip trejetas:

$$\langle U_p, \alpha : U_p \to T(U_p), u_O \in U_p \rangle \tag{2.1}$$

kur U_p yra galimų būsenų aibė, α – funkcija $U \to T(U)$ keičianti komponento būseną, u_0 – pradinė būsena, o

$$T_{I,O}{}^B = A \times B(Id \times O) \tag{2.2}$$

Čia I ir O yra aibės, atliekančios interfeisų vaidmenį ir savo ruožtu gali būti apibrėžtos kaip konkrečios koalgebros. A yra stebimos reikšmės (atributai), Id – identikiatorius, o B nusako konkretų komponento elgsenos šabloną.
Komponentai skirstomi į *aktyvius* (galinčius realizuojanti lygiagrečiąsias veiklas) ir *pasyvius* (realizuojančius nuoseklias veiklas).
2003 m. *J. He*, *Z. Liu* ir *X. Li* [78] aprašo komponento modelį ir *Komponentinio skaičiavimo (Component calculus)* specifikavimo kalbą. Nagrinėjamos komponento, kontrakto ir interfeiso sąvokos. Interfeisas nusakomas pora: $I = \langle FDec, MDec \rangle$, kur $FDec$ yra laukų, o $MDec$ – operacijų aprašų aibės. Nagrinėjami tik teikiami interfeisai, reikalaujamų interfeisų nėra.
kontraktas nusakomas ketvertu:

$$Ctr = \langle I, FVal, MSpec, Init \rangle , \tag{2.3}$$

kur I yra interfeisas, $FVal$ – funkcija visiems interfeiso laukams suteikianti pradines reikšmes, $MSpec$ – funkcija atvaizduojanti kiekvieną interfeiso operaciją į jos specifikaciją, $Init$ – inicializavimo funkcija. Norint parodyti, kad interfeise gali būti ir vidinių, „paslėptų“, operacijų, kontrakto sąvoka praplėsta:

$$GCtr = \langle Ctr, PriMDec, PriMSpec \rangle , \tag{2.4}$$

kur $PriMDec$ ir $PriMSpec$ yra atitinkamai vidinių operacijų aprašai ir funkcija atvaizduojanti juos į specifikaciją.
Aprašant komponentą daroma prielaida, kad jis realizuoja tik vieną interfeisą.

Jei komponentas realizuoja daugiau interfeisų, tai jie pagal darbe [78] nusakytas taisykles yra agreguojami ir toliau traktuojami, kaip vienas interfeisas. Komponentą sudaro trejetas:

$$\langle I, GCtr, ImportedMDec \rangle , \tag{2.5}$$

kur $ImportedMDec$ yra kitų komponentų operacijos reikalingos užtikrinti aprašomojo komponento funkcionalumą.
2005 m. ta pati mokslininkų grupė paskelbė darbą [79], kuriame daugiau kalbėta ne apie atskirus komponentus, bet apie jų sistemas. Parodyta būtinybė aprašyti ne tik komponentų funkcionalumą, bet ir nefunkcines savybes. Išskirtos tokios specifikuotinų komponentų savybių grupės:

- interfeisai (sintaksinis lygmuo);
- funkcinės savybės;
- dinamika (lygiagretumas, sinchronizavimas);
- komunikavimo protokolai;
- ribojimai (vykdymo laiko ar kitų resursų).

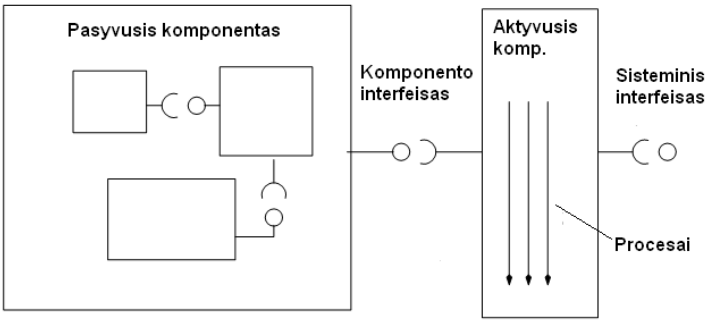
Pastebima, kad galimi keli požiūriai į komponentines sistemas:

- kaip į architektūrinę (statinę) sistemą;
- kaip į sąveikaujančią (angl. *behavior*) sistemą;
- kaip į pažingsniui vystomą (angl. *refinement*) sistemą.

Siekiant apjungti šiuos požiūrius siūloma apjungti keletą skirtingų teorijų:

- „horizontalia“ kryptimi: funkcionalumo tikslinimo (angl. *state-based functional refinement*), įvykiais grindžiamo sąveikos imitavimo (angl. *event-based interaction simulation*), realaus laiko, atsparumo (angl. *fault-tolerance*), apsaugos, mobilumo ir bendrojo kokybės užtikrinimo (angl. *Quality of Service – QoS*) teorijas;
- „vertikalia“ kryptimi: dalykinės srities analizės ir reikalavimų rinkimo teorijas, sistemų surinkimo iš komponentų, komponentų kūrimo ir komponentų išdėstymo (angl. *deployement*) teorijas.

Pristatomas *Bendrosios programavimo teorijos* [83] pagrindu sukurtas perrašymų skaičiavimas – rCOS, leidžiantis ne tik specifikuoti objektines bei komponentines sistemas, bet ir tas specifikacijas tikslinti. Siūlomame skaičiavime (rCOS) galima atlikti objektinių konstrukcijų (angl. *designs*) struktūrines ir



2.1 pav. UNU/IIST komponentinė sistema. Parengta pagal [163].

elgsenos transformacijas. Dar detaliau rCOS nagrinėjamas vėlesniame *J. He, X. Li* ir *Z. Liu* darbe [77].
Darbe [79] teigiama, kad komponentas neturi reikalaujamų interfeisų ir neturi būsenos (kaip ir interfeisas). Be to, komponentas turi būti nepriklausomas nuo savo aplinkos.

X. Chen grupė [35] papildo darbuose [79, 78] aprašytą komponento modelį proceso modeliu, aprašydami komponentų tarpusavio jungimosi savybes, ir įveda jungiančiojo kodo bei procesų sąvokas. Komponentinę programą, anot *X. Chen* [35], sudaro komponentų šeima (angl. *family*) ir keletas procesų. Dalis komponentų yra paimti iš saugyklos, o kita dalis gaunama jungiant procesus ir komponentų operacijas.

P. Thai ir *D.V. Hung* [163] apibendrina UNU/IIST darbus ir pastebi, kad rCOS turi trūkumų: nėra priemonių lygiagretumui ir paveldėjimui aprašyti. Jų siūlomą komponento modelį sudaro: konstrukcija (angl. *design*), kontraktas, interfeisai ir komponentai. Konstrukcija apibrėžiama kaip pora:

$$\langle \alpha, FP \rangle \tag{2.6}$$

kur α – operacijos naudojamų kintamųjų aibė, FP – *Bendrojoje programavimo teorijoje* [83] naudojama operacijos specifikacija:

$$FP : p \vdash R \equiv ok \vdash p \Rightarrow ok' \wedge R \tag{2.7}$$

kur p, R – atitinkamai operacijos prieš ir po sąlygos, ok ir ok' – loginiai kintamieji rodantys operacijos „gyvybingumą“: $ok = TRUE$, jei operacija startavo, o $ok' = TRUE$, jei ji baigėsi.

Darbe [163] įvedama reikalaujamo interfeiso sąvoka ir papildomos interfeiso, kontrakto ir komponento sampratos. Be to, tiek kontraktai, tiek interfeisai

skirstomi į bendruosius ir sisteminius. Taip skirstoma atsižvelgiant į tai, kad *P. H. Thai* ir *D. V. Hung* siūlomame modelyje komponentinę sistemą sudaro dviejų rūšių komponentai: *aktyvusis* ir *pasyvusis*. Aktyvųjį komponentą sudaro reaktyvūs procesai, kurie reaguoja į išorinius (sistemos atžvilgiu) įvykius naudodami *pasyviojo* komponento paslaugas. Pasyvusis komponentas šiek tiek primena *komponento konteinerius* (pvz.: EJB ar CORBA). Įvedama uždarojo komponento sąvoka [163]. Uždaruoju komponentu laikomas sudėtinis komponentas, kuris nebeturi reikalaujamų interfeisų, t.y. visi reikiami komponentai jau įjungti.

Pasyvusis komponentas aprašomas pora $\langle Ctr, Mcode\rangle$, kur Ctr – komponento kontraktas, $Mcode$ – operacijų realizacija.

Komponento kontraktas, lyginant su [79, 78], pakeistas atsisakant atribūtų reikšmių inicializavimo funkcijos $FVal$ ir įvedant invariantus (Inv_p, Inv_r) bei protokolą Pro, nusakantį leistinas operacijų kvietimo sekas:

$$Ctr = \langle I, Init, MSpec, Inv_p, Inv_r, Pro\rangle\,. \tag{2.8}$$

Protokolas Pro specifikuojamas naudojant CSP kalbą.
Sisteminis kontraktas sudarytas iš keturių elementų:

$$SysCtr = \langle SI, SMSpec, Inv, Behav\rangle\,, \tag{2.9}$$

kur Inv – invariantai (nebeskirstomi į teikiamų ir reikalaujamų paslaugų, nes sisteminis interfeisas yra tik teikiamų paslaugų interfeisas); $Behav$ – išorinės elgsenos aprašas, baigtinis aibės

$$e, m | e \in E, m \in Smd_p \tag{2.10}$$

poaibis (čia Smd_p – baigtinė operacijų aibė). Kiekvienas šio poaibio elementas – proceso specifikacija.

Komponento interfeisas I aprašomas, kaip teikiamų ir reikalaujamų interfeisų junginys $\langle I_p, I_r\rangle$, o *Sisteminis interfeisas* aprašomas visiškai kitaip, pabrėžiant, kad jis skirtas reaguoti į išorinius įvykius:

$$SI = \langle E, Fd, Smd_p\rangle \tag{2.11}$$

kur E – baigtinė įvykių, į kuriuos reaguojama, aibė, Fd ir Smd_p – savybių ir operacijų aibės atitinkamai.
Aktyvusis komponentas dar turi ir *Sisteminį kontraktą*

$$SysCtr:\ \langle Ctr, SysCtr, Mcode\rangle\,. \tag{2.12}$$

Be „prieš“ ir „po“ sąlygų [163] dar siūloma įvesti galimas paslaugų kvietimo sekas, tai nurodant sąveikos protokolą. Paslaugomis čia vadinama prieiga prie

duomenų arba operacijos. 2007 m. paskelbtas darbas [36], kuriame apibendrinami grupėje atlikti moksliniai tyrimai ir diskutuojama būtinybė naudoti modelinę paradigmą – *MDA*. Darbe kalbama ne tik apie programinį, bet ir apie aparatinį (angl. *hardware*) komponentą, kuriam aprašyti naudojamos tos pačios sąvokos. 2009 m. pateiktas metodą realizuojantis, *Eclipse* aplinkos pagrindu veikiantis komponentinių sistemų modeliavimo įrankis *rCOS Modeler* .

2.1.3. UMM modelis

Unified Metamodel (UMM) komponento modelis yra projekto *UniFrame* [173, 144, 73] rezultatas. Projekto tikslas – sukurti efektyvų karkasą patikimoms komponentinėms išskirstytoms sistemoms kurti. Vykdant projektą buvo siekiama:

- sudaryti komponentų, jų ryšių, kontraktų ir ribojimų metamodelį,
- siekiant užtikrinti tarpusavio operacinį suderinamumą (angl. *interoperability*), automatiškai generuoti *tarpininkus* (angl. *glue and wrappers*), atsižvelgiant į specifikacijas
- pateikti elementarių ir sudėtinių komponentų specifikavimo ir kokybės tikrinimo bendruosius reikalavimus.

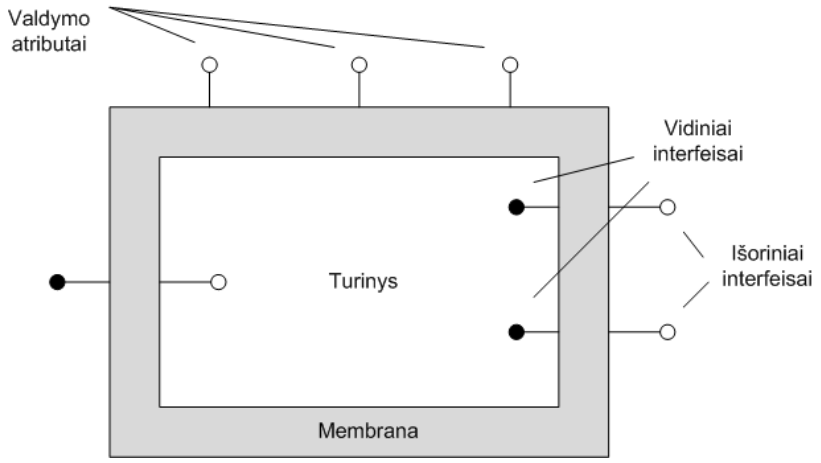
UMM modelyje komponentas aprašomas trimis konceptais: *objektas*, *paslauga* ir *bendradarbiavimas* (angl. *collaboration*).
Kiekvienas objektas nusakomas trimis aspektais:

- skaičiuojamuoju,
- sąveikos,
- papildomu (apima mobilumą, apsaugą, atsparumą klaidoms).

Kiekviena klasė turi *universalųjį interfeisą* (UI) aprašantį visų aspektų savybes bei pasižymintį *refleksijos* savybe. Vėlesniame darbe [72] UMM komponento modelis palygintas su .NET ir pasaulinio tinklo paslaugų komponentų modeliais, papildytas dar ir kontrakto sąvoka.

2.1.4. *Fractal* modelis

Mokslinio projekto *Fractal* tikslas – sukurti modulinį, plečiamą, nepriklausomą nuo programavimo kalbos komponento modelį, skirtą sistemų projektavimui, realizacijai ir konfigūracijos keitimui [56]. *Fractal* komponento modelis kuriamas laikantis turnių atskyrimo principo.



2.2 pav. FRACTAL komponento modelis [56].

Fractal komponento modelyje [24] yra nagrinėjamas vykdomasis komponentas (angl. *run-time entity*). Komponentą sudaro:

- **Interfeisai**. Kiekvienas interfeisas aprašomas šiomis charakteristikomis:
  - **tipas** (ar interfeisas yra teikiamų paslaugų, ar reikalaujamų),
  - **operacijų aibė**,
  - **darna** (angl. *consistency*) (nusako ar operacijos privalo būti atliktos (angl. *mandatory*), ar ne (angl. *optional*)).
  - **kardinalumas** (nusako kiek kreipinių į tą patį interfeisą komponentas gali aptarnauti).
- **Turinys** (angl. *content*) – baigtinė vidinių komponentų aibė.
- **Membrana** palaiko interfeisus komponentų peržiūrai ir vidinių savybių perkonfigūravimui. Turi *vidinius*, *išorinius* interfeisus ir valdymo elementus leidžiančius sustabdyti, patikrinti ir atlikti kitus veiksmus su komponento vidiniais komponentais (*sub-komponentais*).
- **Valdymo taškai**. *Fractal* komponento modelyje galimi trys kontrolės lygmenys:
  - 1 lygmuo – juodosios dėžės. Elementarūs komponentai (angl. *base components*) kuriuos „apklausti“ ar įtakoti nėra jokios galimybės.
  - 2 lygmuo – komponento interfeiso. Šio lygmens komponentą galima „apklausti“, tačiau negalima jo įtakoti.

– 3 lygmuo leidžia įtakoti viską. Skiriami keturi valdymo elementų tipai: *atributų valdymo*, *jungčių*, *turinio* (įgalina papildyti sudėtinį komponentą vidiniais komponentais arba juos pašalinti), *gyvavimo ciklo valdymo*.

Pagrindinės *Fractal* komponento modelio (2.2 pav.) savybės yra šios:

- **rekursyvumas** – komponentai gali būti kitų komponentų vidiniais komponentais,
- **refleksija** – komponentai gali stebėti kitų komponentų ir savo pačių galimybes.
- **bendro komponentų naudojimo galimybė** – komponento egzempliorius (angl. *instance*) gali būti naudojamas daugiau nei vieno komponento.
- **jungčių komponentai** angl. (*binding components*) inkapsuliuoja bet kokį (asinchroninį ir sinchroninį) komunikavimą.
- **nepriklausomybė nuo vykdymo modelio** (angl. *execution model independance*) – realizacijoje gali būti naudojamas tiek klasikinis gijų, tiek įvykiais grįstas, tiek kuris nors kitas modelis.
- **atvirumas** – nefunkcinės komponento savybės gali būti keičiamos per *membraną*.

Programos architektūra aprašoma sudėtinio komponento konstravimo taisyklėmis ir jungtimis (angl. *bindings*), kurios dar skirstomos į *elementariąsias* (jomis tiesiog sujungiami interfeisai) ir *sudėtines* (komunikavimui dar naudojama viena ar daugiau tarpinių jungčių (angl. *stubs*, *skeletons*, *adapters*)).
Apibrėžiami komponentai, galintys kurti naujus komponentus. Jie vadinami komponentų gamyklomis (angl. *component factory*). Skiriamos dvi komponentų gamyklos rūšys:

- bendros paskirties,
- konkretaus tipo.

2.1.5. PECOS modelis

2000–2002 metais vykdytas projektas *PECOS: Pervasive Component Systems (IST-1999-20398)*, kurio tikslas – pritaikyti komponentinę paradigmą įterptinėms (angl. *embedded*) sistemoms kurti, realizuoti aplinką leidžiančią specifikuoti, komponuoti, tikrinti konfigūraciją ir išskirstyti komponentines

įterptines sistemas [61].
PECOS projekte operuojama aparatiniais-programiniais komponentais (angl. *field device*), kurie aprašomi kaip *interfeiso* ir *savybių paketo* pora. Savybių pakete nurodomos nefunkcinės komponento savybės, tokios kaip laiko, atminties ar kt. resursų naudojimas. Aparatinio-programinio komponento interfeisą sudaro *prievadų* sąrašas. Čia prievadas suprantamas kaip bendrai naudojamas (angl. *shared*) kintamasis, leidžiantis komponentams keistis informacija. Prievadą charakterizuoja: *vardas*, *tipas*, *reikšmių sritis* ir *kryptis*. Prievado kryptis gali būti *in*, *out* arba *inout* ir parodo, kad kintamasis gali būti tik rašomas, tik skaitomas arba ir rašomas ir skaitomas atitinkamai [129]. Prievadai jungiami naudojant *jungtis*,kurios iš esmės realizuoja prievadų bendro naudojimo ryšį.
Komponentai skirstomi į tris grupes:

- **Pasyvieji komponentai** neturi valdymo elemento. Jie tik vykdo kitų komponentų komandas. Dažniausiai naudojami veiksmams, kurie vykdomi sinchroniškai ir greit baigiasi.
- **Aktyvieji komponentai** turi valdymo elementą. Naudojami ilgalaikėms veikloms realizuoti.
- **Įvykių komponentai** suveikia tik įvykus numatytam įvykiui. Daugiausia naudojami aparatiniams komponentams modeliuoti.

Sudėtinis komponentas aprašomas pora $\langle C, P\rangle$, kur C – jį sudarančių komponentų sąrašas, o P – prievadų sąrašas [61, 129]. Prievadai skirstomi į išorinius ir vidinius. Išoriniai prievadai gali būti naudojami komponentą jungiant į kitas sistemas, o vidiniai yra iš išorės neprieinami, naudojami tik vidiniams sudėtinio komponento komponentams komunikuoti. Tokia sudėtinio komponento architektūra rodo, kad PECOS projekte naudojamas surenkamasis programavimas, t.y. visa vykdymo logika inkapsuliuota į pačius komponentus, kokių nors atskirų jungiančiųjų dalių nėra.

2.1.6. UML komponento modelis

J. Cheesman ir *J. Daniels* daugiausiai dėmesio skiria didelės apimties verslo (angl. *enterprise-scale*) komponentų specifikavimui. Darbe [34] teigiama, kad egzistuoja keturios komponento formos:

1. komponento specifikacija (atskirai aptariamas dar ir *komponento interfeiso* vaidmuo šiame lygmenyje, 2.2 lentelė);
2. komponento realizacija;
3. įdiegtas komponentas;

2.1 lentelė Komponento ir jo interfeiso specifikacijų palyginimas [34].

komponento specifikacija	Interfeiso specifikacija
Interfeisų sąrašas.	Operacijų sąrašas.
Aprašo skirtingų interfeisų informacinių modelių priklausomybes.	Aprašo loginį informacinį modelį.
Nusako kontraktą su realizuotoju.	Nusako kontraktą su naudotoju.
Aprašo realizacijos ir vykdymo (angl. *run-time*) vienetą.	Aprašo, kaip operacijos įtakoja informacinį modelį arba nuo jo priklauso.
Aprašo operacijų realizaciją kitų interfeisų terminais.	Aprašo lokalų poveikį.

4. komponento objektas.

komponentu nelaikomas klasikinis objektas (nes komponento objektas gali veikti tik komponentinės sistemos kontekste) ar paslauga. Skiriami du **kontrakto** tipai:

- **Naudojimo kontraktas**, kuris sudaromas tarp *komponento objektų* vykdymo metu. Jis aprašomas interfeiso specifikacijoje. Naudojimo kontraktą aprašo *operacijos* (sintaksė, prieš ir po sąlygos) ir *informacinis modelis* (abstraktus bet kokios informacijos tarp kliento ir komponento objekto aprašas, ribojimai tai informacijai).

- **Realizavimo kontraktas**, kuris sudaromas tarp *komponento specifikacijos* ir *komponento realizacijos*. Aprašomas komponento specifikacijoje ir yra jos dalis.

J. Cheesman ir *J. Daniels komponentinę architektūrą* aprašo kaip aibę taikomųjų programų lygmens (angl. *application-level*) komponentų, jų struktūrinių ryšių ir elgsenos priklausomybių (angl. *behavior dependences*). Elgsenos priklausomybės gali būti šios:

- interfeisas $\rightarrow$[1] interfeisas;

- komponentas $\rightarrow$ interfeisas;

- komponentas $\rightarrow$ komponentas;

- vidinis komponentas $\rightarrow$ komponentas.

[1] ženklu „$\rightarrow$“ žymimas sąryšis „priklauso nuo“.

Komponento realizacijos lygmens architektūroje gali būti komponentų, nepaminėtų komponento specifikacijoje, be to komponentai gali papildomai sąveikauti.

Darbe [34] teigiama, kad gali egzistuoti keletas alternatyvių komponento objekto lygmens architektūrų. Pvz., vieno komponento objekto funkcionalumą gali užtikrinti keletas skirtingų komponento objektų rinkinių.

2.1.7. *Poernomo* grupės modelis

Darbe [146] pagrindinis dėmesys skiriamas sudėtinių komponentų kūrimui, todėl tipinis komponento modelis plečiamas konstrukcijomis, egzistuojančiomis pramoninėje tarpinėje programinėje įrangoje.
komponento modelį [146] sudaro:

- **Kenai** (angl. *kens*) - architektūrinės esybės. Paprasčiausi kenai yra komponentai nagrinėjami kaip „juodosios dėžės“. Kenai gali būti ir sudėtiniai.

- **Vartai** (angl. *gates*) - saugo kenus. Paprasčiausių suderinamų vartų pora nusako jungtį. Vartus aprašo: *paslaugų signatūra* (angl. *signature*), *vartų protokolas*, *nefunkciniai paslaugų atributai.*

- **Jungtys** yra dviejų tipų: saistymai (angl. *bindings*) (naudojamos sujungti teikiamas ir reikalaujamas paslaugas) ir atitiktys (angl. *mapping*) – naudojamos sujungti tik teikiamas paslaugas.

- **Kontraktus įgyvendinančios esybės** (angl. *contractual supliers*): komponentai, objektai, operacijos. Pastebėsime, kad jų invariantai gali būti sąlyginiai: „po“ sąlyga priklauso nuo „prieš“ sąlygos ir atvirkščiai.

Komponetų grupės veikia tik tam tikrame kontekste [141]. Kiekvienas komponentas reikalauja paslaugos arba ją teikia. *I. Poernomo* grupės nagrinėjamiems komponentams būdingos būsenos, todėl *Component-FSM* kalba komponentai aprašomi taip:

$$Ctr = \langle E_c, A_c, Z_c, \delta_c, F_c, z_{0c} \rangle , \tag{2.13}$$

kur E_c – įvykių aibė, A_c – veiksmų aibė, Z_c – komponento būsenų aibė, δ_c – būsenos keitimo funkcija ($Z \times E \times AZ$), F_c – leistinų būsenų aibė, z_{0c} - pradinė komponento būsena. Kaip teigiama [146], toks modelis leidžia iš anksto įvertinti komponentinių architektūrų patikimumą.

2.1.8. *Cervantes* modelis

H. Cervantes darbe [28] pagrindinis dėmesys kreipiamas *dinaminiam pakeičiamumui*, t.y. daroma prielaida, kad ne visada paslaugą teikiantis komponentas yra pasiekiamas.
H. Cervantes apibrėžtą komponentą su išore sieja šie elementai:

- **Teikiamos ir reikalaujamos paslaugos**, kurias charakterizuoja dar keturios savybės:
  - *Kardinalumas*, nusakantis ar priklausomybė privaloma ir nuo kelių (vienos ar daugelio) kitų paslaugų priklauso nagrinėjamos paslaugos.
  - *Politika*, nusakanti sudėtinio komponento dinamiškumą. Esant statinei politikai, komponentinė programa visą gyvavimo laiką nesikeičia, t. y. nekeičiami susiejami komponentai. Jei politika dinaminė, susiejami komponentai gali keistis ir sistema gali būti perkonfigūruota).
  - *Filtras*, nusakantis taisykles, pagal kurias atrenkami komponentai programų sistemai.
  - Susiejimo (angl. *bind/unbind*) būdai.
- **Įgyvendinimo priklausomybės**, pvz., nuo vykdymo aplinkos.
- **Veiksnumas** (tinkamumas). Pradžioje visi komponentai startuoja su veiksnumo požymiu „*invalid*“ ir šis požymis pakeičiamas į „*valid*“ tik tuo atveju, jei visos priklausomybės patenkintos.

Komponento gyvavimo ciklui palaikyti naudojamas specialus konteineris (angl. *instance manager*).

2.1.9. *Salzmann* modelis

C. Salzmann [148] nagrinėja išskirstytąsias sistemas, kaip aibę paslaugas teikiančių ir jų reikalaujančių komponentų, kurie komunikuoja per jungtis (angl. *bindings*). Darbe aprašomas komponento modelis nusakomas šiais elementais:

- **Interfeisas** – dvikryptis *kanalas* apibrėžiantis pranešimų tipus.

- **Paslauga** – tai elgsenos vienetas. Kiekviena paslauga aprašoma trejetu $\langle if \in IF, beh, NS \subseteq S \rangle$, kur if – interfeisas iš interfeisų aibės IF; beh – „juodosios dėžės“ elgsenos aprašas; NS – reikalaujamų paslaugų aibė; $\forall ns \in NS$, ši paslauga turi savo interfeisą $if_{ns} \in IF$.

- **Saistymo jungtis** (angl. *binding*) – tai ryšys b tarp dviejų paslaugų: $b \in B = S \times S$. Jungtimi susiejama vieno komponento teikiama paslauga su kito komponento reikalaujama paslauga.

- **Tranzakcija** – baigtinė pranešimų tarp susietų paslaugų seka.

- **komponentas** aprašo tik struktūrą, kaip paslaugos yra realizuotos ir sugrupuotos, todėl jo sudėtinės dalys yra teikiamų ir reikalaujamų paslaugų aibės, bei interfeisų aibė. Komponentas gali turėti vidinių komponentų.

- **Vykdymo vieta** (angl. *sandbox*) – išreikštiniu būdu adresuojama komponentų buvimo vieta (konkretus kompiuteris arba tinklas).

- **Konfigūracija** (kitaip, techninė architektūra) – komponentų, susietų pagal jų teikiamas ir reikalaujamas paslaugas aibė. Tam pačiam tikslui pasiekti gali būti keletas skirtingų konfigūracijų. Konfigūracija keičiama arba pridedant/šalinant jungtį, arba perkeliant komponentą į kitą vykdymo vietą.

Komponentams specifikuoti *C. Salzmann* [148] naudoja XML kalbos pagrindu sukurtą SADL (*Service Architecture Description Language*) kalbą.

2.1.10. *Berger* grupės modelis

K. Berger grupės siūlomą komponento modelį sudaro šie elementai [19]:

- **komponentas**;

- **interfeisas**;

- **jungtys** tarp interfeisų. Išskiriama, kad jungčių kardinalumas gali būti nebūtinai vienetinis, t.y. galimos tipo „vienas-prie-daug“ ir kt. jungtys;

- **laiko srautas** naudojamas komponento elgsenai fiksuoti ir aprašyti;

- **konfigūracijos istorija** aprašo komponentinę sistemą ir parodo, kaip keičiasi sistema vykdymo metu: $Conf = \langle I, K, Cn \rangle$, kur I - interfeisų, K – komponentų, o Cn – jungčių aibės atitinkamai. Jei komponentas, interfeisas ar jungtis yra pašalinama iš sistemos, jie negali būti vėl įvedami. Pašalinus komponentą automatiškai pašalinami ir jo vidiniai elementai, bei jungtys, jungiančios tą komponentą su kitais;

- **pranešimų sekos**. Skirstomos į interfeiso ir komponento pranešimų sekas. Gaunamų komponento pranešimų seka:

$$IfIn = M^* \times (Cn \to M) \tag{2.14}$$

siunčiamų komponento pranešimų seka:

$$IfOut = (Cn \to M^*) \times M^* \tag{2.15}$$

Gaunamų interfeiso pranešimų seka:

$$CompIn = I \to M^* \tag{2.16}$$

siunčiamų interfeiso pranešimų seka:

$$CompOut = I \to M^* \tag{2.17}$$

K. Berger grupės modelyje daug dėmesio skiriama komponentų ir jų sistemų elgsenos kitimo laike tyrimams.

2.1.11. *Aguirre* ir *Maibaum* modelis

N. Aguirre ir T. Maibaum darbe [4] komponentai suprantami kaip sistemą sudarantys skaičiavimų vykdymo ar duomenų saugojimo struktūriniai vienetai. Komponentai skirstomi į bazinius ir sudėtinius. Baziniai komponentai yra nedalomi, turi atributais nusakomą būseną. Jie apibrėžiami trejetu:

$$\langle Rv, At, Ac \rangle , \tag{2.18}$$

kur Rv specialūs įeities atributai, kurių komponentas nekontroliuoja, bet naudoja informacijai apie aplinkos būseną gauti, At – atributai būsenai nusakyti (kitaip, lokalūs kintamieji), Ac – veiksmų aibė, specifikuojanti komponento elgseną.

Sudėtiniai komponentai sudaromi iš bazinių ir sudėtinių, juos susiejant jungtimis. Jie dar vadinami posistemėmis, jas suprantant kaip sudedamųjų komponentų konfigūracijas, kurios gali būti dinamiškai keičiamos. Taigi, posistemių veiksmai ne tik keičia komponentų atributų reikšmes, bet ir modifikuoja savo vidinę struktūrą (pvz., sukuriami ar naikinami komponentų egzemplioriai) bei komponentų sąveiką (pvz., šalinami jungčių egzemplioriai). Jei du komponentai yra sujungti, jų elgsenos yra susijusios jungties tipo sąlygojamu būdu.

Pagrindinis dėmesys [4] darbe skiriamas dinamiškai perkonfigūruojamų komponentų specifikavimo kalbai. Parodoma, kad komponentai gali būti specifikuojami naudojant šiuos elementus:

- bazinius duomenų tipus,
- bazinius komponentų tipus,
- jungčių tipus,
- sudėtinių komponentų tipus.

Specifikacijas sudaro keli hierarchiškai organizuoti sluoksniai - nuo duomenų tipų iki posistemių architektūrų specifikacijų.

2.1.12. *Nierstras, Lumpe* ir *Schneider* modelis

J. G. Schneider [149] aprašomą komponento modelį sudaro šie elementai:

1. komponentai,
2. scenarijai (angl. *scripts*), skirti vykdymo logikai realizuoti ir komponentinei sistemai konfigūruoti,
3. jungiantysis kodas, skirtas komponentų adaptavimui,
4. koordinuojantysis elementas (komponentų konkurencija valdyti).

Modelyje komponentinė programa apibrėžiama taip:

```
komponentinė programa = komponentai + scenarijai
```

Ypatingas dėmesys skiriamas antrajai dedamajai, t.y. jungiančiojo programinio kodo (JPK) sąvokai. Skiriamos šios JPK grupės:

- **pakikliai** (angl. *wrappers*). Inkapsuliuoja visas apgaubiamojo komponento komponento operacijas ir teikia jo teikiamas paslaugas. Gaubtai dar skirstomi į dvi grupes:
  1. adapteriai – susieja nesuderinamus interfeisus, pvz., keičia operacijos parametrų tvarką;
  2. trasformatoriai – naudodami protokolų nesuderinamumui minimizuoti;
- **tiltai** (angl. *bridges*) vieno komponento reikalaujamas prielaidas verčia į kito komponento teikiamas prielaidas; juos gali kviesti ir specialus išorinis *agentas*, kuris nebūtinai yra vienas iš sąveikaujančių komponentų;
- **įgaliotieji tarpininkai** (angl. *proxies*) – nutolusių komponentų surogatai (pvz., *CORBA stubs, skeletons* ir pan.);

- **mediatoriai** (angl. *mediators*). Apima adapterių bei transformatorių savybes, tačiau nuo *tiltų* skiriasi tuo, kad mediatorius turi planavimo funkciją, kuri nustato skirtumus. Pvz.,

$$(D_0 \to D_1 + D_1 \to D_2) \Rightarrow D_0 \to D_2.$$

Darbe [149] nurodomi reikalavimai, kuriuos turi tenkinti JPK:

1. skaidrumas (JPK egzistavimas turi būti išoriškai nepastebimas),
2. juodos dėžės principas,
3. komponuojamumas (projektavimo metu),
4. struktūros dvidališkumas (turi būti keičiamoji ir pastovioji dalys),

Ši mokslininkų grupė daugiau analizuoja komponentų ir juos sudarančių objektų, bet ne komponentinių sistemų specifikavimo galimybes [128]. Specifikavimui naudojamas algebrinis metodas ir πL bei PICOLLA kalbos.

2.1.13. *Yoshida ir Honiden* modelis

K. Yoshida ir S. Honiden [190] nagrinėja vizualiuosius grafinės vartotojo sąsajos komponentus (JavaBeans ir ActiveX/COM). Autorių teigimu komponentą nusako šios savybės:

$$\langle In, Rn, Ic, Rc, O, Exp, M \rangle \tag{2.19}$$

kur In – komponento įvestis (gaunami duomenys, komandos ir pan.), Rn – komponento atsakas įvestį, Ic – komponento išvestis nukreipta į dukterinius komponentus, o Rc - dukterinio komponento atsakas, O – komponento darbo rezultatas (angl. *Output*), Exp – pranešimai apie išimtis (angl. *exceptions*), M – komponento operacijų aibė. Kiekviena komponento operacija $m \in M$ aprašoma ketvertu: $m = \langle In_m, O_m, PreP, PostP \rangle$, kur In_m ir O_m - operacijos įvestis ir rezultatai atitinkamai. Taip pat naudojamos ir *priešproceso* $PreP$ bei *postproceso* $PostP$ sąvokos. Priešprocesas apdoroja duomenų srautą iš komponento įvesties Ic į dukterinio išvestį komponentui Ic. Postprocesas apdoroja duomenų srautą priešinga kryptimi, t.y. iš dukterinio komponento atsako Rc į paties komponento atsaką Rn.

Komponentai specifikuojami algebriniu metodu. Darbe [190] įvardijami šie reikalavimai specifikacijai:

1. komponentų komponavimas beciklio medžio struktūra turi būti aprašomas termu;

2. komponento rezultatas (angl. *output*) ir atsakas (angl. *return*) turi būti gaunami kartu, kaip termo perrašymo, atitinkančio vykdymą, rezultatas;

3. neapibrėžtumai turi būti aiškiai išreikšti;

4. turi būti nustatytos sąlygos, kada lygybės tampa teisingomis.

K. Yoshida ir S. Honiden [190] išskiria šiuos komponento konstravimo būdus:

- Agregavimas (angl. *putChild*). Komponentas inkapsuliuoja kitą komponentą.
- Komponavimas (angl. *compose*). Komponentai (tiksliau jų įvestys ir išvestys) jungiami laisvai, sudarant medžio, tinklo ar kt. struktūras.
- Nuoseklus jungimas (angl. *Sequental*). Komponentai jungiami nuosekliai, t.y. vieno komponento išvestis yra kito įvestis, medžio tipo ir kitos hierarchinės struktūros neleistinos.
- Pakeitimas (angl. *substitute*) Vienas komponentas keičiamas kitu.
- *Rekursinis kreipinys (angl. repeat).* Komponentas kviečia pats savo operaciją.
- Parametrizavimas (angl. *customize*). Vienas ar daugiau komponentų pateikiami kitam kaip parametrai, t.y. įstatomi į iš anksto numatytas vietas. Pvz., taip plečiamas taikomųjų programų funkcionalumas specialiais priedais (angl. *Add-ins*).

2.1.14. *Lau* grupės modelis

Komponento modelio svarba ypač akcentuojama *K.-K. Lau* vadovaujamos grupės [98, 97] darbuose. Pastebima, kad komponento modelis sudaro semantinį karkasą ne tik komponentui apibrėžti, bet ir nusakyti, kaip jis turi būti sukonstruotas, sukomponuotas ir kaip samprotauti apie visas operacijas su komponentais.

Komponento modelis apibrėžia šiuos aspektus [100]:

- komponento sintaksę - jų konstravimo ir vaizdavimo taisykles;
- semantiką, nusakančią kas yra komponentais;
- komponavimą - kaip jungiami į sudėtinius.

Daugumoje dabartinių komponento modelių komponentams aprašyti naudojamos programavimo kalbos, taip pat IDL, ADL kalbos. Komponentais gali būti laikomi: klasės (JavaBeans, EJB), objektai (COM, CCM, FRACTAL),

architektūriniai vienetai (Koala, SOFA). Komponentą aprašo teikiamų ir reikalaujamų paslaugų interfeisai. Komponentai, taip pat ir jų kompozicijos neturi būsenos.

K.-K. Lau ir *M. Ornaghi* darbe [99] pabrėžia, kad kai komponentas suprantamas kaip „juoda dėžė“, jo interfeisas turi užtikrinti visą naudotojams reikiamą informaciją. Taigi, pasinaudoti komponentais leidžiama tik per interfeisus, kurie pateikia informaciją apie operacijas ir kontekstines priklausomybes. Komponentai apibrėžiami dalykinėse srityse, kitaip - kontekste. Kontekstas nusakomas sukuriant pirmos eilės loginę teoriją, kur kontekstas Ctx aprašomas pora $\langle \Sigma, X \rangle$. Signatūra Σ apima rūšių simbolius, funkcijų ir ryšių deklaracijas, o rekursyvi Σ aibė X nusako dalykinę sritį ir leidžia samprotauti apie ją.

Darbe [82] nagrinėjama komponentinio karkaso (angl. *framework*) sąvoka. Karkasu čia vadinama grupė sąveikaujančių objektų (*ne komponentų!*). Karkasą aprašo:

- **Klasių aibė** C. Klasės operacijos vadinamos parametrizuotais atributais (kaip *Catalysis* projekte). Kiekvienai klasei specifikuojamas jos savybių sąrašas $funs$ ir predikatų aibė *pred*.
- **Įvykių aibė** M. Kiekviena operacija $m \in M$ aprašoma trejetu $m = \langle C, T, L \rangle$, kur T – papildomi parametrai, o L - laikas.
- **Faktų (*ribojimų*) aibė** F. Šia surinkti ribojimai gai būti ir lokalieji ir globalieji, todėl tinka visiems invariantams, o *Prieš* i *Po* sąlygos aprašomos kartu su įvykiais.

Nagrinėjamos komponentinių karkasų funkcijų praplėtimo galimybės panaudojant kitą karkasą.

2.1.15. *Cox* ir *Song* modelis

P.T. Cox ir *B. Song* komponentines sistemas aprašo taikydami tiesinio vamzdyno (angl. *pipeline*) metaforą [41]. Komponentas apibrėžiamas kortežu X, kuris gali būti sudarytas iš skirtingų elementų. Programiniai komponentai klasifikuojami taip:

- **komponentas-siųstuvas** apibrėžiamas vieninteliu kortežo elementu – teikiamų atributų seka *outports(X)*.
- **komponentas-imtuvas** apibrėžiamas vieninteliu kortežo elementu - gaunamų atributų seka *inports(X)*.
- **Elementarus komponentas** kitaip dar vadinamas filtru. Jis aprašomas šiuo kortežu

$$X : \langle inports, outports, f, t \rangle , \tag{2.20}$$

kur *outports* – atributas, *inports* – atributų kortežas f – funkcijos, t – trigeriai. Funkcijos nusako komponento funkcionalumą ir turi tokį pavidalą $f : \tau(a_1) \times \tau(a_2) \times \ldots \times \tau(a_n) \rightarrow \tau(outports))$, kur $\tau(a_i)$ apibrėžia a_i tipą. Trigerių egzistavimas parodo, kad filtras yra aktyvusis – reaguoja į įvykius. Atributai nusako prievadus, per kuriuos komponentai sąveikauja tarpusavyje.

- **Sudėtinis komponentas** apibūdinamas šiuo kortežu

$$X : \langle K, In, Out, Con \rangle , \tag{2.21}$$

kur K – komponentų aibė, In ir Out – skirtingų atributų sekos, Rel – ryšių tarp vidinių komponentų atributų aibė. Con - jungčių aibė. Jungtis sudaro portų poros.

P.T. Cox ir *B. Song* komponento modelyje nesigilinama į komponentiniu sistemų realizacines datales. Komponento modelis nagrinėjamas aukštame abstrakcijos lygmenyje.

2.1.16. *Moschoyiannis* modelis

S. Moschoyiannis darbuose [123, 96] pagrindinis dėmesys skiriamas komponento ir jį sudarančių elementų semantikos aprašymui. Todėl pateikiamas formalus programinio komponento modelis, kuris dar yra naudojamas samprotavimui apie komponentus ir jų kompozicijų savybes.

Darbe [123] pastebima, kad daugelis komponentinių technologijų palaiko greitą programų sistemų surinkimą iš komponentų, tačiau rezultato kokybė paaiškėja tik sukūrus sistemą. Tai nėra priimtina, todėl autorius siekia rezultato kokybę užtikrinti kūrimo metu, t. y. reikiamą sudėtinio komponento elgseną gauti pasirenkant tinkamos elgsenos sudedamuosius komponentus. Kitaip tariant, teigiama, kad apjungus komponentus, kurie tenkina reikiamas savybes, sudėtinis komponentas taip pat tenkins reikiamas savybes. Analizuojant sudedamųjų dalių kombinacijų savybes ir jungimo pasekmes, remiamasi formalia sudėtinio komponento apibrėžtimi ir įvestomis sąlygomis, kurių turi būti prisilaikoma komponavimo procese. Be to, komponavimas semantiniame lygmenyje leidžia išspręsti komponento pakeitimo kitu tinkamu komponentu problemą – pakeičiantysis komponentas turi užtikrinti buvusio komponento funkcionalumą.

Komponentas apibrėžiamas nurodant jo rūšį ir elgseną nusakančių vektorių aibę, t. y. pora (Σ, B) [123, 96]. komponento rūšis Σ apibrėžiama trejetu $(P_\Sigma, R_\Sigma, \beta_\Sigma)$, kur:
$P_\Sigma \subseteq I$ – teikiamų interfeisų aibė, $R_\Sigma \subseteq I$ – reikalaujamų interfeisų aibė, o $\beta_\Sigma(i)$ – išvardintų interfeisuose, kvietimų sekų aibė, kurios poaibis, susijęs su

interfeisu i žymimas β_Σ Išnagrinėjus komponentų modeliavimą (CSP, CCS, SOFA ir pan. priemonėmis) darbe [123] daroma išvada, kad norint aprašyti komponentą, būtina aprašyti jo sąveikos (angl. *behavior*) protokolą, t. y. aprašyti galimas operacijų kvietimo sekas. Todėl komponento dinaminės savybės specifikuojamos apibrėžiant aibę V_Σ, kurios elementai $v(i)$ vadinami elgsenos vektoriais ir priklauso baigtinių sekų, kurias galima sudaryti iš aibės β_Σ elementų, aibei, t. y., $V_\Sigma : I_\Sigma \to Op$, tokia, kad kiekvienam $i \in I_\Sigma, v(i) \in \beta_\Sigma(i)^*$ (* žymi baigtinę seką), tada $B \subseteq V_{Sigma}$.

Komponentas turi tenkinti eilę savybių. Pirma, jo operacijų kvietimo sekų aibė turi būti diskreti, t. y. jei B – - diskreti, tai ir komponentas (Σ, B) laikomas diskrečiuoju. Komponentų diskretumas garantuoja, kad laiko atžvilgiu nėra begalinių didėjančių ar mažėjančių įvykių sekų; kad nėra trūkių (angl. *gaps*) laiko kontinuume; kad yra pradinis taškas, kuriame dar nėra įvykę nei vieno įvykio. Antra, siekiama, kad kiekvieno komponento elgsenoje būtų bent vienas egzempliorius bet kuriam jo interfeisų operacijų kvietimo atvejui. Tai užtikrinama, jei komponentas tenkina lokalaus uždarumo sąlygą. Jei komponentas yra diskretus ir lokaliai uždaras (angl. *locally left closed*) tada jis vadinamas normaliu.

Elgsenos aprašą (angl. *behavior representation*) sudaro keturių elementų junginys: (O, Π, E, λ), kur O – įvykių (angl. *occurrence*) aibė, $\Pi \subseteq \rho(O)$ – netuščia *taškų* (būsenų) aibė, E - įvykių (*events*) aibė, $\lambda : O \to E$ įvykių funkcija tenkinanti $\bigcup_{\pi \in \Pi}(\pi) = O$.
Jei komponentas yra diskretiškas ir lokaliai uždaras (angl. *locally left closed*) tada jis vadinamas *normaliu*.
Kad komponentas c yra lokaliai uždaras tada ir tik tada kai bet kokiems $u \in B$, $i \in I_\Sigma$ ir $x \in \beta_\Sigma(i)$, tenkinantiems sąlygą $\Lambda < x < u(i)$ egzistuoja toks $v \in B$, kad $v < u$ ir $v(i) = x$ komponento lokalus uždarumas reiškia, kad bet kokia operacijų kvietimo tvarka yra leistinų operacijų kvietimo sekos (pirminio elgsenos vektoriaus) poaibis. Komponentą galima keisti kitu tik tada, jei keičiamojo komponento elgsenos vektorius yra keičiančiojo komponento elgsenos vektoriaus poaibis, t.y. keičiantysis komponentas siūlo ne mažesnį funkcionalumą nei ankstesnysis.

2.1.17. *Whitehead* modelis

K. Whitehead [188] įvardija šias komponento interfeiso (plačiąja prasme) aprašymo savybes:

- interfeisus, kuriuos jis siūlo klientus, įskaitant ir galimas išimtis (angl. *exeptions*);
- reikalaujamus interfeisus, kuriuos turi tenkinti besikreipiantis paslaugos klientas

- reikalaujamus interfeisus, kuriuos turi turėti kiti (pagalbiniai) komponentai.

Pastebima, kad interfeisai gali būti ir sinchroniniai ir asinchroniniai (galima kalbėti ir apie *įvykius* įvairiuose komponento modeliuose, EJB žinučių komponentus (angl. *Message Driven Beans*)).
Taip pat pastebima, kad gali būti pateikiamos ir papildomos komponentų charakteristikos:

- būsenos, pakartotiniai kreipiniai (angl. *re-entrancy*), galimybė aptarnauti daugiau nei vieną klientą;
- ar komponentai gali naudoti tą pačią tranzakciją ar reikalauja naujos ir t.t.;
- ar komponentas rašo duomenis į DB, failus ir ar ne;
- parametrizavimo galimybės, apsaugos savybės;
- reikalavimai operacinei sistemai, darbinės atminties poreikiui ir t.t.

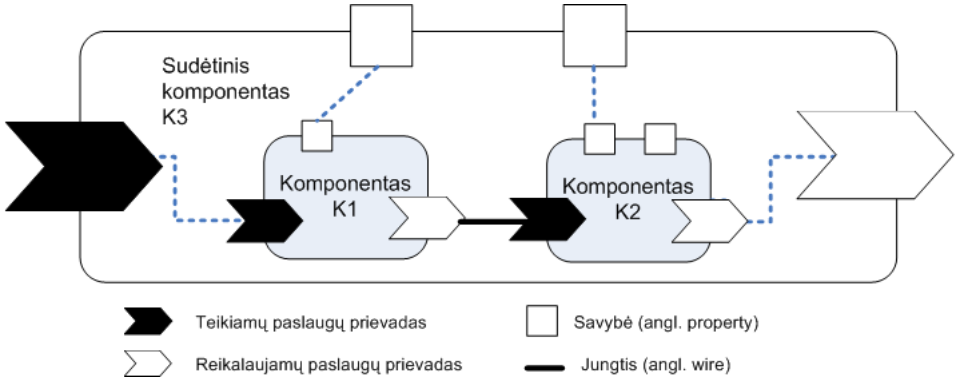
2.1.18. SCA modelis

Paslaugų komponento architektūros arba SCA (angl. *Service Component Architecture*) modelis kuriamas nuo 2005 m. siekiant supaprastinti paslaugų architektūros stiliaus programinės įrangos kūrimo procesą. Modelio kūrimą palaiko IBM, Oracle, BEA, SAP ir dar 14 įmonių.

Komponentams kurti gali būti naudojama bet kuri technologija. Pavyzdžiui, SCA stiliaus komponentai kuriami naudojant Java technologijas (*EJB*, *Java EE Connector Architecture* bei *Java Message Services*), jie realizuoja pasaulinio tinklo paslaugas [12]. Vienas iš SCA tikslų – apibrėžti komponentų jungimo į sudėtinius mechanizmą, taikomą visiems komponentams nepriklausomai nuo to, kokia technologija buvo sukurti sudedamieji komponentai. Tam SCA specifikuoja apibendrinto komponento modelį.

SCA komponentas yra sukonfigūruotas realizacijos egzempliorius. Komponento realizacija (angl. *implementation*) reiškia konkrečios technologijos elementą, tokį kaip Java ar C++ klasė, BPEL procesas, XSLT transformacija ir pan. komponentas gali funkcionuoti viename procese viename kompiuteryje ar būti išskirstytas po kelis procesus keliuose kompiuteriuose. Sudėtinis komponentas vadinamas kompozicija (angl. *construct*) ir yra loginė konstrukcija. Kompozicija turi šias savybes:

- naudojama kaip komponento realizacija – nustato komponento matomumo ribas (nuorodos iš išorės neleidžiamos);



2.3 pav. SCA komponento modelis [18].

- naudojama kaip išdėstymo vienetas – yra SCA domeno (organizacinio vieneto kontroliuojamų paslaugų visumos) elementas.

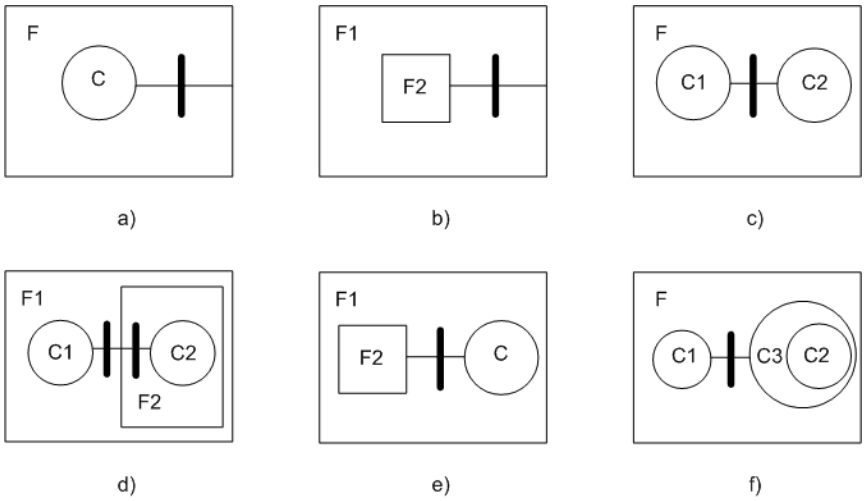
Sudėtinės konstrukcijos (angl. *assembly*) modelis specifikuoja ją sudarančius komponentus, jų interfeisus (paslaugas) ir tarpusavio ryšius, kai komponentai paslaugas teikia vienas kitam. Sąveikai su išore nusakyti naudojamos šios abstrakcijos: paslauga, nuoroda (angl. *reference*), savybė (angl. *property*) ir susiejimas (angl. *binding*) [12, 18]. Komponentų sąveika modeliuojama paslaugomis. Komponentas ne tik teikia paslaugas, bet ir jomis naudojasi. Reikalaujamos paslaugos specifikacija vadinama nuoroda. Jungtis (angl. *wire*) yra ryšio tarp reikalaujamos paslaugos specifikacijos ir ją tenkinančios paslaugos abstraktus pavaizdavimas. Kaip konkrečiai komponentas komunikuoja (kitais žodžiais tariant, kaip naudojamasi teikiamomis ir reikalaujamomis paslaugomis) nurodo susiejimai, kurių komponentas gali turėti ne vieną, t. y. jis gali sąveikauti pagal kelis skirtingus protokolus. Komponentas turi vieną ar kelias savybes, kurios naudojamos jo realizacijai konfigūruoti.

2.1.19. Komponavimo formos

Komponavimo formos priklauso nuo konkretaus komponento modelio [1] ir išreiškia galimų sudėtinių komponentų ir komponentinių sistemų kūrimo taisykles.

Darbe [10] daugiausia dėmesio skiriama būtent komponentų ir komponentinių karkasų komponavimo formoms (2.4 pav.). Autoriai pateikia tokią jų taksonomiją:

a) komponento įkomponavimas (angl. *deployement*) (2.4 pav., a) – komponentinio karkaso ir komponento sąveikos protokolas. Komponentas, negalintis su karkasu sąveikauti numatytuoju būdu, nepriskiriamas konkrečiam komponento modeliui, kurį palaiko karkasas.



2.4 pav. Komponavimo formos [10].

b) Karkaso įkomponavimas (2.4 pav., b) – dviejų ar daugiau karkasų suderinimas veikti kartu (angl. *interoperability*). Sudaro galimybes kurti heterogenines *sluoksnines* (angl. *tiered*) sistemas [159].

c) Elementari kompozicija (2.4 pav., c) – komponentų jungimas per kontraktu aprašytą interfeisą. Ši komponavimo forma sutinkama dažniausiai.

d) Heterogeninė kompozicija (2.4 pav., d) – sudėtingesnis heterogeninių sistemų kūrimo atvejis kai įkomponuojamas ne tik kitas karkasas, bet ir pastarojo palaikomas komponentas.

e) Karkaso praplėtimas (2.4 pav., e) – karkaso galimybių plėtimas, jungiant jį su naujais komponentais. Tokie komponentai dažniausia vadinami *papildiniais* (angl. *add-ins*, *plug-ins*).

f) komponentų apjungimas ir sistemos surinkimas. (2.4 pav., f). Skirtingai nei *elementarios kompozicijos* atveju vienas komponentas (pavyzdyje – C3) gali agreguoti kitą (C2). Pirklausomai nuo komponento modelio agreguotasis komponentas yra išoriškai matomas arba ne [10].

Egzistuoja ir kitos komponavimo formų taksonomijos [190, 147]. Disertacijoje neanalizuojamos heterogeninės komponentinių programų sistemos, todėl toliau bus taikomos tik *elementarios kompozicijos* bei *komponentų apjungimo ir sistemos surinkimo komponavimo* formos.

2.2 lentelė KM taksonomija pagal komponavimo kriterijų [100].

Kategorija	Komp. modelis	DR	RR	CS	DC	CP
1	JavaBeans	Taip	Ne	Ne	Ne	Taip
2	EJB, COM, .NET, CCM	Taip	Ne	Taip	Ne	Ne
3	Koala, SOFA, KobrA	Taip	Taip	Taip	Taip	Ne
4	Fractal, PECOS, UML	Ne	Ne	Taip	Ne	Ne

2.2. Komponento abstrakcijos lygmenys

Kaip minėta 2.1. skyriuje, komponentinė paradigma pasižymi pramoninių ir akademinių programinio komponento modelių gausa. Išsami visų šių modelių analizė yra neįmanoma, todėl paprastai analizuojami tik charakteringi komponento modelių pavyzdžiai. Komponento modelių savybių analizei atrinkti ir disertacijos 2.1.1.–2.1.18. poskyriuose apžvelgti tie komponento modeliai, kurie atstovauja tam tikroms komponento modelių grupėms, pagal šias taksonomijas:

- **taksonomija pagal komponavimo kriterijų** [100, 101]. Komponento modeliai suskirstyti į keturias grupes (2.2 lentelė). Skirstymas atliktas pagal šiuos požymius:

  DR: komponentų saugykla (angl. *repository*) gali būti papildyta kūrimo metu gautais naujais komponentais,

  RR: kūrimo metu komponentai gali būti paimti (angl. *retrieve*) iš komponentų saugyklos,

  CS: komponavimas leidžiamas kūrimo metu,

  DC: komponentų saugykla gali būti papildyta kūrimo metu gautais naujais sudėtiniais komponentais,

  CP: komponavimas leidžiamas išdėstymo (angl. *deployement*) metu.

- **objekto ir proceso taksonomija** [110]. Komponento modeliai klasifikuojama pagal tai ar jų komponentai isaugo savo vidinę būseną ar ne.

- **M.C. Goulao taksonomija** [70]. Komponento modeliai klasifikuojami pagal šiuos požymius:

  – pagal komponento modelio kilmę: pramoninis, akademinis ar mišrus;

  - pagal komponento sąvokos vaizdavimo būdą: komponentai-objektai, komponentai-klasės, komponentai-architektūriniai vienetai;
  - pagal komponento aprašymo kalbą: objektinė kalba, IL, ADL;
  - pagal komponentų saugyklos naudojimo būdą;
  - pagal naudojamų kontraktų tipus;
  - pagal tai ar modelio komponentai sertifikuojami;
  - pagal tai ar išskiriama sudėtinio komponento sąvoka ar ne;

- **taksonomija pagal abstrakcijos lygmenis** [34, 7]. Komponentai skirstomi į šias grupes:
  - *komponento specifikacijos* lygmenyje operuojama projektuojant informacines, arba programų sistemas. Taip pat šis lygmuo būdingas *architektūros aprašymo kalbų – ADL* [116] komponentiniams modeliams.
  - *komponento realizacijos* lygmenyje operuojama pagal specifikaciją sukurtais komponentais, tačiau nekalbama apie jų veikimą ir veikimo palaikymui reikalingas priemones.
  - *įdiegto komponento* lygmenyje nurodomos konkrečios komponento paskirties vietos, konfigūruojama jo vykdymo aplinka (pvz.: vardų skyrimo tarnyba).
  - *komponentinio objekto* lygmuo aktualus jau komponentinės sistemos veikimo etapui, kada pagal konkrečioje vietoje įdiegto komponento taisykles sukuriami objektai. Būtent objektai ir realizuoja aukštesniuose lygiuose deklaruotą funkcionalumą.

Visos minėtos taksonomijos naudotos komponento modeliams atrinkti, o viena jų – komponentų klasifikacija *pagal abstrakcijos lygmenis* – naudojama ir likusioje disertacijoje dalyje (2.3 lentelė).

Konkrečiai, komponentinių programų kūrimo procesas šioje disertacijoje nagrinėjamas tik *komponento specifikacijos* ir *komponento realizacija* lygmenyse, nes būtent šie lygmenys yra naudojami sistemų kūrimo etape. *Įdiegto komponento* ir ypač *komponentinio objekto* lygmenys glaudžiai susiję su sistemos veikimo etapu ir kūrimo procesui yra mažiau aktualūs.

2.3. Komponentų kūrimo procesas

Komponentinių sistemų kūrimo procese skiriamas komponentų ir sistemų kūrimas [37, 7]. Daugeliu aspektų komponentų kūrimas yra panašus į sistemų kūrimą. Jų gyvavimo ciklo modelis apima reikalavimų nustatymo, analizės ir specifikavimo, komponento projektavimo, realizavimo, testavimo, dokumentavimo ir išskirstymo[2] etapus. Kuriant komponentus, gali būti naudojami ar modifikuojami ir anksčiau sukurti komponentai. Tačiau komponentų kūrimas turi šiuos savus ypatumus [37]:

- **Komponentai kuriami tam, kad būtų pakartotinai panaudoti** įvairiose programų sistemose, o tai apsunkina komponentų kūrimo procesą. Komponentai privalo būti lengvai pritaikomi, konfigūruojami, dėl ko didėja jų sudėtingumas ir apimtis.Tuo pačiu metu komponentai privalo būti ir pakankamai paprasti naudoti. Kaip teigia *C. Szyperski* [159], tam, kad sukurti pakartotinai naudojamą komponentą, reikia 3-4 kartus daugiau resursų nei komponentą, kuris bus naudojamas konkrečioje programų sistemoje

- **Sunku nustatyti komponento reikalavimus**. Viena pagrindinių programų kūrimo problemų yra neaiškios, neišsamios ar net prieštaringos reikalavimų specifikacijos. Komponentai (pagal apibrėžtį) gali būti naudojami skirtingose dalykinėse sistemose, kurių išsamus sąrašas iš anksto nėra žinomas, ir todėl jų keliami reikalavimai negali būti net nuspėjami. Komponentų kūrimas yra ne toks sunkus tik tuo atveju, jei jų reikalavimai išlieka nekintantys palyginti ilgą laikotarpį. Augant komponento panaudojamumo laipsniui, didėja ir jo paklausa, plečiasi reikalavimų, kuriuos šis komponentas turi tenkinti, aibė. Siekiant patenkinti naujus reikalavimus, kuriamos naujos komponento versijos (tačiau būtinai išlaikančios visas ankstesnių versijų kontraktu specifikuotas savybes). Ypač sparti evoliucija būdinga pirmosioms komponento versijoms [37]. Iš pradžių komponentas yra ne toks abstraktus ir ne toks bendras, todėl kuriant naujas versijas reikėjo didesnių pakeitimų atnaujinimų. Tuo tarpu paskesnės komponento versijos pasižymi didesniu abstraktumu, todėl šiek tiek pasikeitus reikalavimas komponento keisti nereikia, arba pakanka pakeisti nežymiai. Taigi, adaptavimo mechanizmus turintys apibendrinti komponentai yra naudingiausi. Kai komponentas pradeda nebetenkinti kuriamų sistemų keliamų reikalavimų ir šios problemos nebegali išspręsti net naujos komponento versijos (pvz., pasikeitus technologijoms, paradigmoms), sakoma, kad baigėsi komponento gyvavimo ciklas ir jis toliau nebepalaikomas. Darbe [37] taip pat atkreipiamas

[2] angl. *deployement*.

dėmesys į komponento rišlumo užtikrinimo problemą, kai komponente darant daug pakeitimų rišlumas gali būti sumažinamas.

- **Komponentus būtina itin tiksliai specifikuoti**. Tai ypač svarbu, nes komponento naudotojas yra ne jo gamintojas, o komponentas paprastai parduodamas kaip juodoji dėžė, be pradinių tekstų. Be to, siekiant užtikrinti kuriamų programų sistemų saugą, visi komponentai turi būti *patikimi* (*angl. trustworthy*) [107].

komponentų kūrimo procesas šioje disertacijoje nėra išsamiai nagrinėjamas. Apie komponentų kūrimo procesą bus kalbama tik sudėtinio komponento kūrimo kontekste.

2.4. Komponentinių sistemų kūrimo procesas

Komponentinių programų sistemų gyvavimo ciklą sudaro šie etapai [37, 151, 188, 181]:

1. **Reikalavimų analizė** (*angl. Requirements analysis and definition*). Šiame etape vienareikšmiškai nurodomos sistemos ribos. Komponentinių sistemų kūrimo (KSK) procese šiame etape analizuojamos ir komponentų, kurių sąveika (angl. *collaboration*) užtikrintų kuriamosios sistemos funkcionalumą, specifikacijos. Tam yra būtina apibrėžti sistemos infrastruktūrą. KSK proceso analizės etape sprendžiami trys uždaviniai: sistemos reikalavimų identifikavimas ir sistemos ribų nustatymas, sistemos infrastruktūros, užtikrinančios komponentų sąveiką nusakymas, ir reikalavimų komponentams identifikavimas, pagal kuriuos būtų galima juos pasirinkti arba sukurti.

2. **Komponentų atranka ir įvertinimas** (angl. *selection and evaluation*). Komponentinių programų sistemų kūrimo procese pabrėžiamas ne suprojektuotos sistemos ir jos dalių realizavimas, bet jau sukurtų juodosios dėžės komponentų pakartotinis naudojimas. Šiame etape ieškoma tinkamų komponentų, jei reikia, jie yra parametrizuojami ir vertinami. Taigi šį etapą galima skaidyti į tris poetapius:

   - *komponentų paieška.* Ieškant komponentų reikia atsižvelgti ne tik į jų funkcines, bet ir nefunkcines savybes. Kuriant komponentines sistemas viena didžiausių problemų yra patenkinti kuriamos sistemos nefunkcinius reikalavimus, žinant (arba net nežinant) kokius reikalavimus tenkina sudedamieji komponentai. Egzistuoja įvairios nefunkcinių reikalavimų taksonomijos [38, 155, 88, 58]. Pagrindinės nefunkcinių reikalavimų grupės parodytos 2.3 lentelėje. Nefunkcinių

reikalavimų taksonomija yra glaudžiai susijusi su komponento abstrakcijos lygmenimis, nes kiekviename lygmenyje yra svarbūs tik kai kurie 2.4 lentelėje išvardinti reikalavimai.

- *komponentų konfigūravimas.*
- *komponentų testavimas.*

3. **Sistemos architektūros projektavimas** Pažymėtina, kad architektūros pasirinkimas iš dalies priklauso ir nuo parinktų naudoti komponentų [37]. Pasak [151], šiame etape taip pat atliekamas ir reikalavimų sistemai įvertinimas bei projektinių sprendimų pasirinkimas.

4. **Sistemos realizavimas** Kuriant nekomponentines programų sistemas šiame etape akcentuojamas konstravimas (angl. *construction*), t. y. programinis kodas kuriamas „nuo nulio“, arba pakartotinai panaudojama labai maža dalis anksčiau sukurto kodo. Kuriant komponentines programų sistemas akcentuojamas komponavimas. Komponavimui (angl. *composition*) būdinga tai, kad galutinis produktas gaunamas parenkant iš anksto paruoštus programinius komponentus ir juos sujungiant į vieną sistemą. Bendruoju atveju gali prireikti realizuoti ir jungiantįjį kodą (angl. *glue code*), tačiau visa tokio jungiančiojo kodo apimtis, nustatoma pagal pasirinktą vertinimo metodą (LoC, FP ir pan.), turi neviršyti bendros naudojamų komponentų apimties [188, 50]. Jungiančiojo programinio kodo kūrimas paprastai reikalauja mažiau nei 50% visai sistemai kurti skirtų pastangų, tačiau viena jungiančiojo kodo eilutė sukuriama tris kartus greičiau nei kodo eilutė, realizuojanti komponento funkcionalumą [16].
Komponentų komponavimas – vienas iš būdų surinkti sistemą iš juodosios dėžės komponentų. Pagal komponavimo laiką nagrinėjami du šio proceso atvejai:

- *Komponavimas kūrimo metu* – dažniausiai pasitaikantis atvejis. Komponentams jungti reikiami artefaktai kuriami arba specialiu įrankiu, arba naudojant specialią komponavimo kalbą, arba tiesiog programavimo kalbą, pvz. *C#* ar *JScript*.
- *Komponavimas vykdymo (veikimo) metu.* Šis komponavimo atvejis dar vadinamas dinaminiu. Dinaminis komponavimas naudojimas: (a) kai reikia sukurti programą, tenkinančią trumpalaikius naudotojų poreikius; (b) kai reikia pakeisti sistemos, turinčios veikti nenutrūkstamai (pvz., elektroninės prekybos, klientų aptarnavimo), dalį.

5. **Sistemos integravimas** (*angl. system integration*). Iš anksčiau pasirinktų gatavų komponentų, atsižvelgiant pasirinktą į sistemos architektūrą, sudaroma pati sistema. Darbe [151] pastebima, kad šiame etape galimas ir vienų komponentų keitimas kitais, jei paaiškėja, kad parinktieji komponentai netenkina sistemos reikalavimų. Tokiu būdu gali būti grįžtama į komponentų atrankos ir įvertinimo etapą.

6. **Sistemos tikrinimas ir vertinimas**. Etapo tikslas - nustatyti, ar sukurta programinė įranga atitinka specifikaciją (tikrinimas) ir ar programinė įranga yra tinkama, t. y. tenkina naudotojų poreikius (vertinimas) [37]. Tikrinimo etapas yra svarbus tuo, kad jame gali būti aptinkami atskirų komponentų nesuderinamumo atvejai, kurių negalėjo parodyti komponentų testavimas, atliekamas ankstesniuose etapuose.

7. **Sistemos palaikymas**. Etapą galima skaidyti į poetapius [151]: palaikymo strategijos rengimas, problemų valdymas ir ilgalaikis sistemos palaikymas.

Pastebėtina, kad į komponentų atrankos ir įvertinimo etapą gali būti grįžtama tiek iš sistemos tikrinimo, tiek iš palaikymo etapų. Pirmuoju atveju - paaiškėjus, kad sistema netenkina nefunkcinių reikalavimų, antruoju - pasirodžius naujoms komponentų, karkasų ar operacinių sistemų versijoms.

Komponentinių programų sistemų gyvavimo ciklas (KPSGC) yra ypatingas tuo, kad palyginti su klasikiniu programų sistemų gyvavimo ciklu, keičiasi etapų svarba ir trukmė:

- komponentų atrankos etapo svarba padidėja. Tiek programų sistemos kokybė, tiek kūrimo išlaidos tiesiogiai priklauso nuo šiame etape pasirinktų komponentų savybių. Kaip jau minėta, kuriant sistemą nauji komponentai nekuriami. Tiesa, darbe [151] pateiktame gyvavimo ciklo modelyje numatytas ir komponentų kūrimo etapas, tačiau toks požiūris prieštarauja komponentinės paradigmos procesų atskyrimo principui ir šioje disertacijoje nebus nagrinėjamas.

- Realizavimo etapo svarba ir trukmė sumažėja. Idealiu atveju sistemai sukurti pakanka vien pakartotinai naudojamų komponentų. Jei komponentai nevisiškai atitinka reikalavimus, jie konfigūruojami arba kuriamas nedidelės apimties jungiantysis programinis kodas.

Pastebėtina, kad komponentų atrankos, sistemos projektavimo, realizavimo bei sistemos integravimo etapai iteratyviai kartojami (2.7 pav.) tol, kol programų sistema maksimaliai tenkins ne tik funkcinius bet ir nefunkcinius

reikalavimus. Šis gyvavimo ciklas dar vadinamas *Išrink-Pritaikyk-Bandyk* (angl. *Select-Adapt-Test*) ciklu [37].
komponentinių programų sistemų kūrimo procese pabrėžiamas ne suprojektuotos sistemos ir jos dalių realizavimas, bet jau sukurtų komponentų pakartotinis naudojimas [37]. Šis ypatumas turi šiuos trūkumus:

- pirmiausia būtina rasti komponentus (pakartotinai naudojamus elementus), atitinkančius sistemos reikalavimų specifikaciją ir sistemos projektinius sprendimus;
- reikalingos papildomos pastangos siekiant adaptuoti iš dalies tinkantį komponentą, patikrinti jo patikimumą ir pan.

2.5. Skyriaus išvados

Apibendrinant skyrių galima daryti šias išvadas:

1. Komponento modelių įvairovė yra veiksnys, apsunkinantis automatizuotą komponentinių programų sistemų kūrimo procesą:

   1.1. Apžvelgus komponento modelius, reprezentuojančius žinomas modelių klases, nustatyta, kad jiems aprašyti iš viso naudojamos 28 sąvokos. Kiekvienas modelis aprašomas naudojant nuo 3 iki 12 sąvokų.

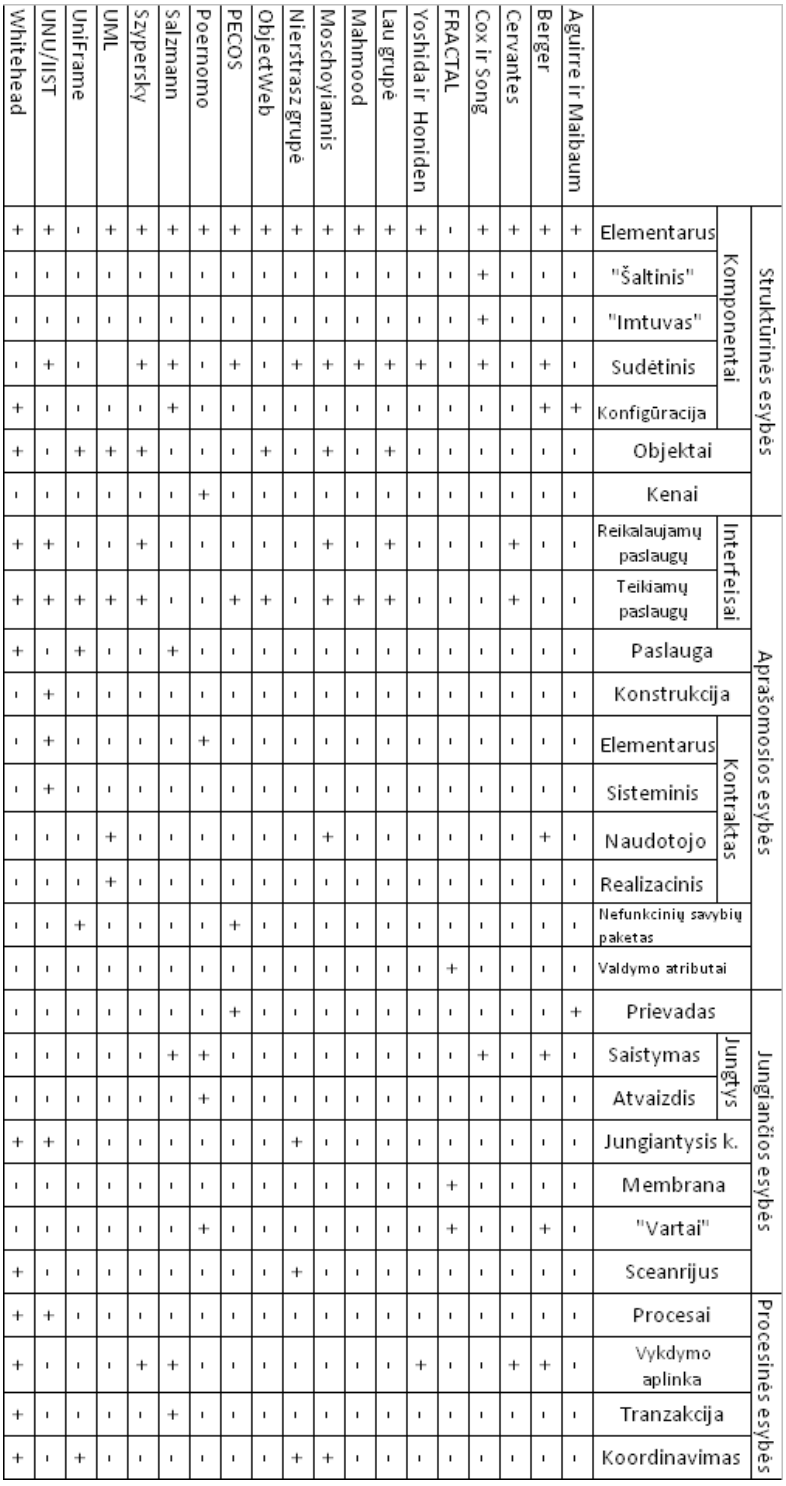
   1.2. Komponentinės programos gaunamos naudojant skirtingas komponavimo formas. Kurios būtent formos yra naudojamos, priklauso nuo komponento modelio ir jo abstrakcijos lygmens.

   1.3. Skiriami keturi komponento abstrakcijos lygmenys. Disertacijoje nagrinėjami *specifikacijos* ir *realizacijos* lygmenų komponentai. Įdiegto komponento ir komponentinio objekto lygmenys šioje disertacijoje nenagrinėjami, nes *įdiegto komponento* ir *komponentinio objekto* lygmenys glaudžiai susiję su sistemos veikimo etapu ir kūrimo procesui yra mažiau aktualūs.

2. Išanalizavus programinių komponentų savybes ir komponentinių programų kūrimo proceso ypatumus nustatyta:

   2.1. Pagrindiniai komponentų kūrimo proceso ypatumai yra šie: komponentai kūriami tam, kad būtų panaudoti daugiau nei vieną kartą; jie kuriami nežinant, kokie reikalavimai jiems bus keliami konkrečiose sistemose; komponentų specifikacija ir dokumentacija turi būti tiksli ir išsami.

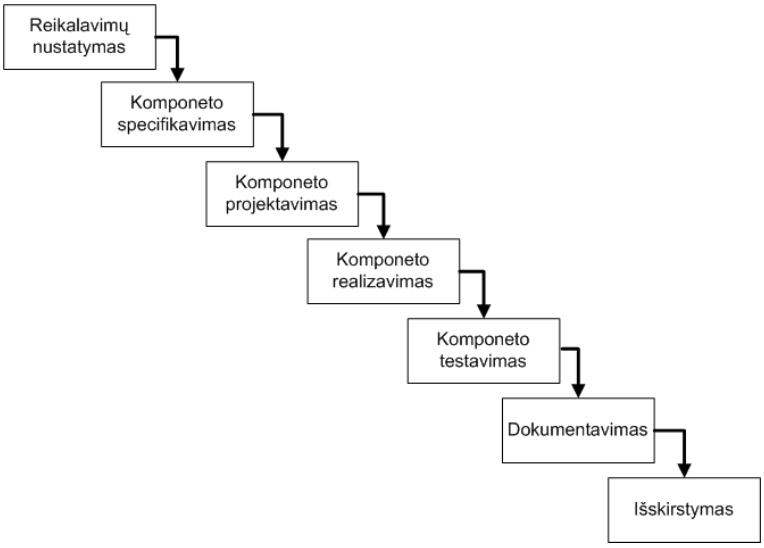
   2.2. Komponentinių programų sistemų kūrimo procesas, kuriame akcentuojamas ne suprojektuotos sistemos ir jos dalių realizavimas, bet jau sukurtų komponentų pakartotinis panaudojimas turi trūkumų: neišspręstas detalios komponentų paieškos uždavinys; komponentų adaptavimui reikalingos papildomos sąnaudos.



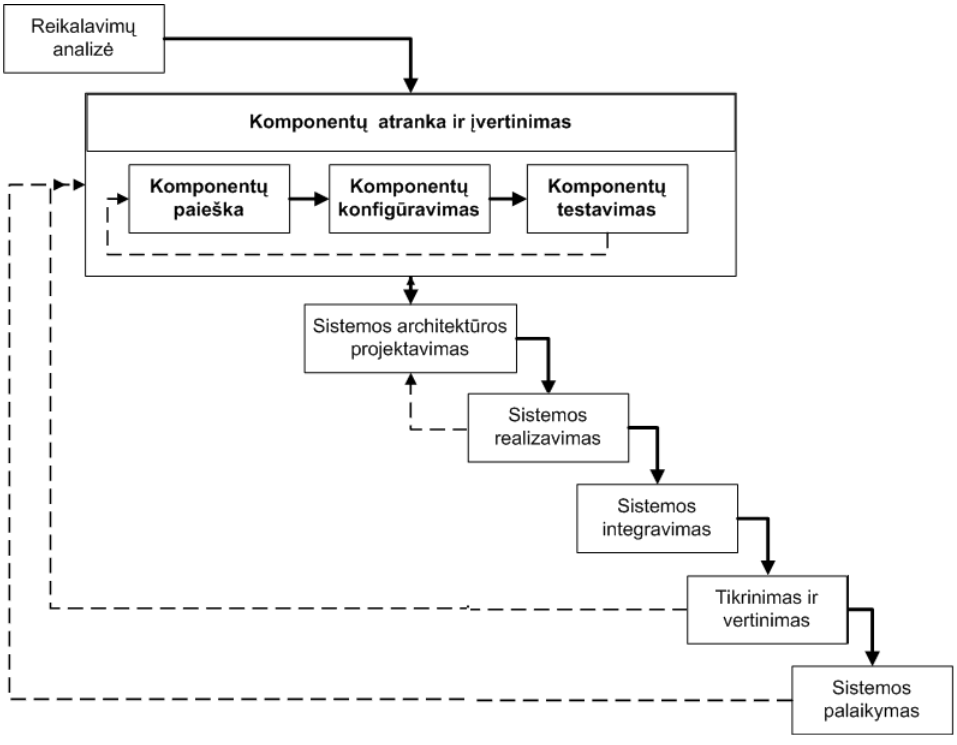
2.5 pav. Komponento modelių elementai.

2.3 lentelė Komponento abstrakcijos lygmenys.

Komponento modeliai	Komponento specifikacija	Komponento realizacija	Įdiegtas komponentas	Komponentinis objektas
Apperly	+	+	+	-
Szypersky	+	-	+	+
UNU/IIST	+	-	-	-
UMM (Uniframe)	+	-	-	+
FRACTAL	+	+	-	-
PECOS	+	-	-	-
UML	+	+	-	+
Poernomo	+	-	-	-
Cervantes	+	-	+	-
Salzmann	+	+	-	-
Berger	+	-	+	+
Aguirre ir Maibaum	+	-	+	-
ObjectWeb	+	-	-	+
Nierstrasz, Lumpe, Schneider	-	-	-	+
Yoshida, Honiden	+	-	+	-
Lau grupė	-	-	-	+
Cox ir Song	+	-	-	-
Whitehead	+	+	+	-
Moschoyiannis	+	-	-	-



2.6 pav. Komponento kūrimo etapai.



2.7 pav. Komponentinių programų sistemų kūrimo etapai.

2.4 lentelė Nefunkciniai reikalavimai. Parengta pagal [165, 38].

Aktualūs *planavimo etapui*	Aktualūs *projektavimo etapui*	Aktualūs *veikimo etapui*
• Atsparumas, robastiškumas – Konceptualus integralumas – Išbaigtumas – Rišlumas/darna • Prieinamumas – Pardavimo kaina. – Palaikymo kaštai. – Integravimo kaštai	• Mobilumas • Pakartotinis panaudojamumas • Testuojamumas • Išplečiamumas • Konfigūruojamumas • Mastelio keičiamumas • Suderinamumas su kitomis sistemomis	• Sparta – Našumas – Atsako laikas – Reikalaujamų resursų kiekis • Patikimumas – Pasiekiamumas – Saugumo lygis – Atsparumas gedimams • Panaudojamumas

3 skyrius

Programų sistemų generavimo metodų lyginamoji analizė

Automatinio programavimo terminas pradėtas vartoti ne vėliau kaip 1954 m., kai buvo norima nusakyti pirmuosius *Fortran* kalbos kompiliatorius [17]. Aštuntame XX a. dešimtmetyje generavimo metodais laikyti išplėstasis kompiliavimas (naudojamas SETL, RAPTS ir TAMPR [27, 25] sistemose), metaprogramavimas (naudojamas ZAP, HOPE, CIP [53] sistemose). Todėl programų kūrimo automatizavimas tradiciškai buvo suprantamas kaip kompiliavimo problema, kai formali specifikacija kompiliuojama siekiant gauti vykdomą programą. Nepaisant naujų kalbų, kūrimo metodų, įrankių atsiradimo, nuolatinis dėmesys automatizavimui reiškė, kad programų kūrimas ir šio proceso našumas yra pagrindinė kompiuterinių sistemų kūrimo problema. Kita vertus, programų kūrimas yra daugiau nei kompiliavimas. Tai apima ir specifikacijų sudarymą, ir jų teisingumo užtikrinimą, ir projektinius sprendinius, aukšto abstrakcijos lygmens specifikacijas verčiant į žemesnio abstrakcijos lygmens specifikacijas, ir pan. Taigi, automatinis kompiliavimas ilgainiui išsivystė į automatizuoto programavimo paradigmą ir automatizuotą programų sistemų inžineriją (angl. *automated software engineering*).

Skiriamos kelios automatizuoto programų sistemų kūrimo kryptys priklausomai nuo naudojamos šio uždavinio sampratos ir kūrimo modelio. Pavyzdžiui, yra žinomos, deduktyvi paradigma [153], žiniomis grindžiama paradigma [93], modeliais grindžiama paradigma [138], o programų sistemų kūrimo procesas įvardijamas kaip generavimas, sintezė, transformacinė realizacija, modeliais grindžiamas kūrimas.

Nesiekdami įvesti išsamios taksonomijos pastebėsime, kad vartojami terminai dažnai yra susiję su konkrečios mokyklos tradicijomis. Tačiau, aišku, pirmiausia jais siekiama pabrėžti automatizuoto kūrimo proceso ypatumus. Pavyzdžiui, terminas *generavimas* akcentuoja: a) automatizuotą rezultato gav-

imą, b) apibendrintų ruošinių naudojimą, o terminas *sintezė* - sudedamųjų dalių jungimą į vieną visumą. Disertacijoje vartojamas terminas *generavimas*. Tuo norima ne pabrėžti konkrečių procesų ir metodų ypatumus, o apibendrintai įvardinti skirtingus automatizuoto kūrimo metodus. Terminas *sintezė* vartojamas kaip termino *generavimas* sinonimas, kai siekiama prisilaikyti konkrečioje srityje vartojamos terminijos.

Šio skyriaus tikslas – išanalizuoti programų sistemų automatizuoto kūrimo metodus įvertinant jų taikymo galimybes komponentinėms programų sistemoms kurti. Metodų analizė pradedama programų sistemų inžinerijoje taikytų klasikinių formaliųjų metodų (3.1. poskyris) nagrinėjimu. Toliau analizuojami deduktyviosios (3.2. poskyris), struktūrinės (3.3. poskyris), induktyviosios (3.5. poskyris) ir transformacinės sintezės (3.6. poskyris) metodai. Ypatingas dėmesys skiriamas 3.4. poskyryje aprašomam įrodomąjį programavimą apibendrinančiam *Curry-Howard* protokolui.

3.1. Klasikiniai formalieji metodai

Klasikiniais formaliaisiais metodais šioje disertacijoje vadinami *E. Dijkstra* [48] ir *C.A.R. Hoare* [83] darbai, bei jais besiremiantys išvestiniai *M.Charpentier* [30], *R.Backhouse* [11] ir *J.Schumann* [152] metodai.

Fundamentaliame darbe [48] *E. Dijkstra* programavimą aprašo naudodamas determinuotos mašinos abstrakciją. Tokiai mašinai, kaip programos vykdymo *posalyga* (angl. *post-condition*) nusakomas predikatas, reiškiantis programos galutinį tikslą (t.y. kam suskaičiuoti programa yra skirta), kokioje būsenoje determinuotos mašina turi atsidurti. Turint galutinę programos būseną svarbu žinoti iš kokių pradinių būsenų ji gali būti pasiekiama. Jei mašiną galima pervesti į vieną iš tokių būsenų, tai aišku, kad skaičiavimų rezultatas bus pasiektas. Visų tokių pradinių būsenų, garantuojančių, kad iš jų bus pasiekta galutinė būsena ir mašina galutinėje būsenoje pasiliks, aibė nusakoma silpniausia sąlyga (angl. *weakest precondition*):

$$wp(S, R), \tag{3.1}$$

kur S – deterministinė mašina, o R – „po“ sąlyga (norimas rezultatas). Jei $wp(S, R)$ sąlyga netenkinama, tai mašina gali sustoti kitoje nei R būsenoje arba gali iš viso nebaigti darbo. Dažnai sprendžiant skaičiuojamuosius uždavinius visos galimų pradinių būsenų aibės rasti nereikia, todėl naudojama *pakankama sąlyga* P, pasižyminti savybe: $P \Rightarrow wp(S, R)$.
Silpniausia sąlyga pasižymi savybėmis:

$$wp(S, F) \quad = \quad F$$

$$\begin{aligned} \forall Q = R : wp(S,Q) &= wp(S,R) \\ (wp(S,Q) \wedge wp(S,R)) &= wp(S, Q \wedge R) \\ (wp(S,Q) \vee wp(S,R)) &\Rightarrow (wp(S, Q \vee R)) \end{aligned}$$

Sakoma, kad programos konstrukcijos S semantika žinoma pakankamai gerai, jeigu žinomas jos *predikatų transformatorius* (angl. *predicate transformer*), t.y. funkcija, iš bet kurios „po“ sąlygos R gaunanti silpniausią sąlygą $wp(S,R)$. *E.W. Dijkstra* pabrėžia, kad bet kuri programavimo kalba turi dvi paskirtis: mašinos vykdymui ir pačių programos kūrėjų naudai, kad iš programos būtų galima nustatyti predikatų transformatorių. *E.W. Dijkstra* darbe [48] atskleidžia, kaip konstruojama programavimo kalba, kurios kiekviena konstrukcija aprašoma per predikatų transformatorių wp:

$$\begin{aligned} \text{Priskyrimo sakinys: } wp(\text{“}x := E\text{“}, R) &= R_{E \to x} \\ \text{Funkcinė kompozicija: } wp(\text{“}S1; S2\text{“}, R) &= wp(S1, wp(S2, R)) \\ \text{Sąlyginis sakinys: } wp(IF, R) &= (B_1 \vee B_2 \vee \ldots \vee B_n) \wedge \\ &\quad (B_1 \Rightarrow wp(SL_1, R)) \wedge \\ &\quad (B_2 \Rightarrow wp(SL_2, R)) \wedge \ldots \wedge \\ &\quad (B_n \Rightarrow wp(SL_n, R)) \\ \text{Ciklas: } wp(DO, R) &= (Ek : k \geq 0 : H_k(R)) \end{aligned}$$

kur $H_k(R)$ – silpniausia „prieš“ sąlyga, tam kad konstrukcija darbą baigtų ne daugiau kaip po k komandų išrinkimo.

R. Backhouse darbe [11] aprašo formalų, sintaksinį programų bei jų savybių gavimo (angl. *calculation*) būdą. Teigiama, kad jis užtikrina aukščiausią įmanomą programinės įrangos patikimumą. Kaip vienas motyvacijos pavyzdžių pateikiamas dvejetainės paieškos (angl. *binary search*) algoritmas, bei jo realizacija Java kalba. Siekiama atskleisti programinės įrangos kūrimo, naudojant formaliuosius metodus svarbą.

R. Backhouse [11] aprašo skaičiuojamąją logiką (angl. *calculational logic*) su ekvivalentumo, neigimo, disjunkcijos, priskyrimo, praleidimo (angl. *skip*), silpninimo (angl. *weakening*), sąlygos aksiomomis. Konstrukcijų aprašymui naudojama *C.A.R. Hoare* notacija: $P\mathbf{S}R$[1] [83, 80]. Taip pat aprašomi kvantoriai (3.1 lentelė). Čia simetrinio dvejetainio operatoriaus *vienetu* vadinama

[1] čia S - operacija.

reikšmė kurią tuo operatoriumi sukomponavus su bet kuriuo x gauname x : $1_{\oplus} \oplus x = x$.

3.1 lentelė Konstrukcijų kalbos kvantoriai. Parengta pagal [11].

Operatorius	**Vienetas**	**Kvantorius**	**Reikšmė**
$\wedge$	true	$\forall$	konjunkcija
$\vee$	false	$\exists$	disjunkcija
$+$	0	$\sum$	sudėtis
$\times$	1	$\prod$	daugyba
$\downarrow$	∞	$\Downarrow$	minimumas
$\uparrow$	$-\infty$	$\Uparrow$	maksimumas
$\equiv$	true	$\equiv$	ekvivalencija
$\not\equiv$	false	$\not\equiv$	neekvivalencija
$\cup$	$\oslash$	$\bigcup$	sąjunga
$\cap$	U	$\bigcap$	sankirta

R. Backhouse [11] parodo, kaip aprašyti pagrindines programavimo kalbos konstrukcijas: nuoseklų sakinių jungimą (angl. *sequent composition*), sąlygos sakinį, ciklą. Pabrėžiama, kad konstruojant programas labai svarbus konstruktoriaus kūrybiškumas (angl. *creativity element*). Kiekviena formaliai aprašyta kalbos konstrukcija formaliai aprašo ir tą kūrybiškumo elementą. Pavyzdžiui sąlygos sakiniu toks elementas yra sprendimas, kaip padalinti uždavinį į dalis; nuosekliame jungime kūrybingumo elementas yra sprendimas, kokia tarpinė sąlyga turi būti patenkinta, kai baigiamas vykdyti pirmasis sakinys; ciklo konstrukcijose kūrybingumo elementas – tinkamos invarianto savybės projektas.

Darbe [30] vystoma komponuojamųjų sistemų teorija. Daugiausia dėmesio skiriama komponavimo taisyklėms, įvedamos tokios savybės kaip asociatyvumas, simetrija ir idempotentumas (angl. *idempotency*). Darbe laikoma, kad specifikacija (tiek sistemos, tiek komponento) turima tada, kai yra išvardijamos visos jos (jo) savybės. Jungiant komponentus „jungiamos“ ir jų specifikacijos bei tokiu būdu gaunama ką tik sukurtos sistemos specifikacija. Taigi, specifikacijos yra transformuojamos. Jos taip pat gali būti transformuojamos ir siekiant gauti „geresnes“ kompozicines specifikacijas. Todėl siūlomasis metodas [30] remiasi predikatų transformatoriais (angl. *predicate transformers*) – funkcijomis transformuojančiomis predikatus į kitus predikatus. Kaip pripažįsta [30] autorius, šios funkcijos primena *E.W. Dijkstra* [48] silpniausią salygą (angl. *weakest precondition – WP*) ir stipriausią sąlygą (angl. *strongest precondition – SP*) (3.2 lentelė).

Daugiausia nagrinėjamos sistemos sudarytos iš elementariųjų komponentų, tačiau neatmetama ir sudėtinių komponentų panaudojimo galimybė. Sistemos,

kaip ir komponentai žymimi didžiosiomis raidėmis, o komponavimo taisyklės žymimos dvinariais operatoriais: $\star$, $\bigodot$.

Kadangi siūloma tik pati koncepcija, komponavimo operatorius naudojamas tik tam, kad parodytų jog kompozicija iš viso įmanoma, neatsižvelgiant į jos detales. Ta pati komponavimo taisyklė gali būti susijusi su skirtingais *sudėrinamumo operatoriais* [32], kurie neaptariami darbe [30]. Sistemų savybių predikatai arba atskiriami tašku (.), arba pateikiamos tokių predikatų aibės. Pvz., jei F ir G - sistemos, turinčios savybę X tai $X.F \bigodot G$ bus žymima sudėtinė sistema iš F ir G, turinti tą pačią savybę.

M. Charpentier pabrėžia būtinybę, nustatyti sistemos savybes pagal komponentų savybes ir atvirkščiai (jei komponentų neturime, ir jų reikia ieškoti) [30].

Savybių sąrašas laikomas *komponuojamu*, jei paskutinė sąrašo savybė išlaikoma bet kurioje sistemoje, sukurtoje iš komponentų išlaikančių likusias savybes. Pavyzdžiui, jei nagrinėsime sistemą $(F \bigodot G) \star H$, tai savybės (X, Y, Z, T) būtų laikomos komponuojamomis tik jeigu

$$\left\langle \forall F, G, H :: X.F \wedge Y.G \wedge Z.H \Longrightarrow T.(F \bigodot G) \star H \right\rangle$$

Jei T būtų nežinomas, gautume lygtį. Šios lygties sprendinių konjunkcija taip pat būtų sprendinys, tačiau egzistuoja ir stipriausias sprendinys (angl. *strongest solution*), kuris žymimas $(X \bigodot Y) \star Z$.

Komponavimo savybės apibrėžiamos dvejomis teoremomis:

1 teorema. $(X \bigodot Y) \star Z$ charakterizuoja tas sistemas, kurios gali būti gautos komponuojant X, Y ir Z sistemas per $\star$, $\bigodot$. Bet kuri taip sukomponuota sistema tenkina $(X \bigodot Y) \star Z$ ir bet kuri sistema, kuri tenkina $(X \bigodot Y) \star Z$, gali būti dekomponuota:

$$(X \bigodot Y) \star Z.K \equiv \left\langle \exists F, G, H : X.F \wedge Y.G \wedge Z.H : (F \bigodot G) \star H = K \right\rangle$$

Pasinaudojus šia teorema galima perrašyti kompozicinių savybių sąrašo apibrėžimą:

$$\left\langle \forall K :: (X \bigodot Y) \star Z.K \Rightarrow T.K \right\rangle$$

o tai leidžia nebekalbėti apie sistemas, o kalbėti tik apie jų savybes. Jei kuri nors posistemė yra ypatinga ir reikia pabrėžti, kad ji yra sistema, galima naudoti savybę „būti sistema F“: $\langle \forall G :: F_{=}.G \equiv (F = G) \rangle$.

Jei $(X \bigodot Y) \star Z$ yra ekvivalenti X (t.y. išlaikomos visos X savybės), [30] sakoma, kad X gerai komponuojamas (angl. *good compositional behavior*). Gali būti ir kitaip:

- Sistema $(X \bigodot Y) \star Z$ turi daugiau savybių nei X. Dažnai būtent to ir siekiama.
- Sistema $(X \bigodot Y) \star Z$ turi mažiau savybių nei X
- Sistema $(X \bigodot Y) \star Z$ redukuojasi į konstantą *true* t.y. visos X savybės prarandamos.

Lygtis $S : [(S \bigodot Y) \star Z \Rightarrow T]$ turi silpniausią sprendinį (sprendinių disjunkcija taip yra sprendinys). Egzistuoja funkcija atvaizduojanti (angl. *maps*) T į silpniausią sprendinį (kai Y ir Z - fiksuotos savybės): $((? \bigodot Y) \star Z).T$ Ši funkcija yra predikatų transformatorius. Jis charakterizuoja, kokia turi būti sistema (sudėtinis komponentas), kurią sukomponavus su Y ir Z, gautume sistemą su savybėmis T. Įrodžius, kad konkreti sistema (komponentas) yra lygties sprendinys, kurį galima jungti į sistemą (komponuoti).

2 teorema.

$$((? \bigodot Y) \star Z).T.K \equiv \left\langle \forall F, G : Y.F \wedge Z.G : T.(K \bigodot F) \star G \right\rangle$$

Jei ieškomasis komponentas pasižymi savybe T ir ta savybė yra *gerai komponuojama*, tai ir sistema išlaikys savybę T. Tačiau, jei T, nėra gerai komponuojama, gali tekti įrodyti, kokią *stipresnę nei* T savybę turi turėti ieškomasis komponentas. Blogiausiu atveju, jei $(? \bigodot Y) \star Z).T$ redukuojasi į *false* konstantą, komponentas, kuris tiktų sudėtinei sistemai su T savybe neegzistuoja.

Darbe [30] parodomas *M. Charpentier* metodo ryšys su *E. W. Dijkstra* darbais [49] (3.2 lentelė).

Kiekvienas predikatų transformatorius τ turi unikalų τ^* *konjugatą* (angl. *conjugate*): $\tau^*.X \equiv -\tau.(-X)$. Transformatorius $(? \bigodot Y) \star Z$ taip pat turi konjugatą $((? \bigodot Y) \star Z)^*$, tokį, kuris charakterizuoja sistemos (formuojamos iš Y ir Z sistemų) likutį. Nors pastarasis transformatorius ir padeda rasti trūkstamo komponento savybes, jis negarantuoja, kad sistema tenkins savybes T. Todėl ieškomasis komponentas dar turi tenkinti ir $((? \bigodot Y) \star Z).T$

M. Charpentier [30] pabrėžia, kad šos idėjos ypač būtų naudingos kuriant lygiagrečiąsias konkurencines sistemas, modeliuojant jas laiko logikomis. Jis komponavimo kontekste atsiribojama tiek nuo skaičiavimų, tiek nuo būsenų. Šios savybės laikomos neturinčiomis komponavimui įtakos. Komponavimo operatoriai $\bigodot, \star$, taip pat abstraktūs. Nenagrinėjama konkreti jų prasmė ir netgi savybės (simetrija, asociatyvumas ir pan.).

3.2 lentelė *E. W. Dijkstra* ir *M. Charpentier* teorijų atitikmenys.

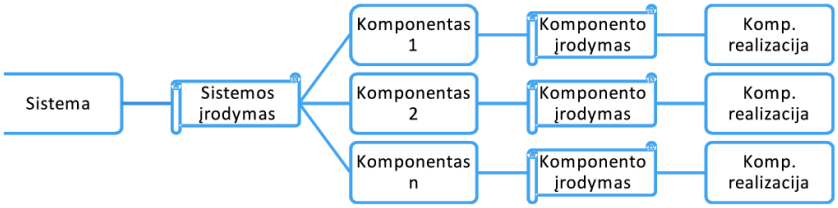
Dijkstra/Hoare	*Charpentier*
"programų semantika"	"komponuojamumas"
būsena/predikatas	sistema/savybė
vykdyti programą	sukonstruoti sistemą
priskirti kintamąjį	prijungti posistemę
wlp (weakest liberal precondition)	predikatų transformatorius „?", nes $wlp.s.p$ reiškia, kad jei eilutė (angl. *statment*) s įvykdyta ir s baigiasi, tai gaunama būsena q. [30] siūloma $(? \bigodot F_{=}).T$ reiškia, kad jei prie sistemos F pridėsime/komponuosime F gausime sistemą su T savybėmis.
sp (strongest precondition)	predikatų transformatorius $(\lambda X.(X \bigodot Y))$, nes $sp.s.p$ reiškia, būsena, kuri yra gaunama po to, kai būsenoje, tenkinančioje p vykdoma eilutė s. $X \bigodot Y$: stipriausia savybė, kuri išlaikoma po to, kai prie X prijungiama Y.
$[p \Rightarrow wlp.s.q] \equiv [sp.s.p \Rightarrow q]$	$[X \Rightarrow (? \bigodot Y).Z] \equiv [X.Y \Rightarrow Z]$
$[wlp.(s; s').q \equiv wlp.s.(wlp.s'.q)]$	$[((? \bigodot Y) \star Z).T \equiv (? \bigodot Y).((? \star Z).T)]$

Darbe [33] išlaikomas konceptualusis požiūrus, t.y. nagrinėjamos kompozicinės sistemos nenurodant, kad nagrinėjami komponentai yra programiniai. Darbe laikomasi nuomonės, kad bendrųjų komponavimo teorijų narinėjimas gali padėti identifikuoti esminius bet kokių komponavimo formų bruožus.

Viena didesnių komponavimo problemų yra tai, kad komponentų (ir sistemų taip pat) specifikacijos yra kur kas abstraktesnės nei jų realizacijos, o tokio detalumo gali nepakakti renkantis komponentus, tam kad juos jungiant gautume korektišką programą.

Darbe [33] pristatomas įrodymų vaidmuo komponentinių programų kūrime (3.1 pav.). Pabrėžiama, kad didžiausias teoremų įrodymo sudėtingumas turi būti paliktas atskirų komponentų įrodymams. Tuo tarpu visos sistemos įrodymas turi būti toks paprastas, koks tik gali būti. Komponentų įrodymai, sistemos įrodyme yra tik naudojami, tačiau jokiu būdu negali būti modifikuojami. Pasak [33], komponentų įrodymų sudėtingumas padeda užtikrinti, kad sistemos įrodymas būtų kuo paprastesnis, t.y. Komponento įrodymas atlieka ir tarpininko tarp kompozicinių savybių ir realizacijos detalių vaidmenį.

3.3 lentelėje palyginti *M. Charpentier* predikatų transformatoriai.



3.1 pav. Įrodymų vaidmuo automatizuotame komponentinių programų kūrimo procese.

3.2 pav. Programų sintezės metodų tarpusavio ryšiai.

M. Charpentier kompozicinių sistemų savybių prognozavimo metodas gali būti taikomas numatyti ar komponentinės programų sistemos atitiks nefunkcinius reikalavimus, jei šiuos reikalavimus atitinka komponentai.

3.2. Deduktyviosios sintezės metodas

Deduktyvioje programų sintezėje taikomas deduktyvus samprotavimų būdas, kuris kitaip dar vadinamas saugiu. Naudojant šį metodą programa gaunama pasinaudojant uždavinio sprendinio egzistavimo teoremos įrodymu. Jei uždavinys išsprendžiamas, programa sudaroma įrodymo procedūros pagrindu. Tai realizuojama dviem būdais: interpretuojant įrodymą arba iš jo išgaunant programą pasirinkta kalba [111].

Deduktyvinės programų sintezės metodai, kurie kaip formalizmą naudoja konstruktyviąsias teorijas, dar vadinami konstruktyviosios sintezės [55, 167], arba įrodomojo programavimo (angl. *proofs-as-programs*) [74, 142] metodais

(3.2 pav.). Toks metodas grindžiamas *Curry-Howard* izomorfizmu [133, 84] (žr. 3.4. poskyrį). Teisingos programos sintezuojamos iš specifikacijų intuicionistinio (konstruktyviojo) įrodymo.

Konstruktyviosios sintezės metodų klasėje dar skiriamas ir struktūrinės programų sintezės metodas. Taikant šį metodą įrodymo metu atsižvelgiama tik į struktūrines skaičiavimų savybes.
Plačiausiai žinomos deduktyviosios sintezės sistemos yra šios:

- *DEDALUS* [55]. Viena pirmųjų *Z. Manna* ir *R. Waldinger* sukurtų sistemų. Pirma versija realizavo transformacinę sintezę, tačiau vėliau ji buvo modernizuota į įrodomojo programavimo sistemą. Sistemai pateikiama formali specifikacija, išsami ir darni reikalavimų atžvilgiu, taip pat dalykinės srities teorija. Autoriai, atsižvelgdami į įrodomojo programavimo ypatumus, sukūrė savo teoremų įrodymo sistemą, pavadintą dedukciniu tablo (angl. *deductive tableau*).

- *Amphion* [6]. NASA iniciatyva sukurta sistema automatizuotam programų, skirtų dangaus kūnų padėtims ir palydovų trajektorijoms modeliuoti. Sistemai pateikiama formali specifikacija, išsami ir darni reikalavimų atžvilgiu, taip pat dalykinės srities teorija. Specifikacija sudaroma ir analizuojama grafinio naudotojo interfeiso pagalba. Įrodymo sistema iš esmės nesiskiria nuo DEDALUS įrodymo sistemos, tačiau čia naudojama automatinio teoremų įrodymo programa (angl. *prover*) SNARK [156]. Be to, sintezės rezultatas - programa gali būti išverčiama į kitą imperatyviojo programavimo kalbą, pvz., Fortran. Sėkmingiausi Amphion panaudojimo rezultatai yra mėnulio fazės kitimo modeliavimo programa (angl. *phase of moon*) ir saulės padėties *Galileo* palydovo atžvilgiu modeliavimo (angl. *solar incidence angle*) programa.

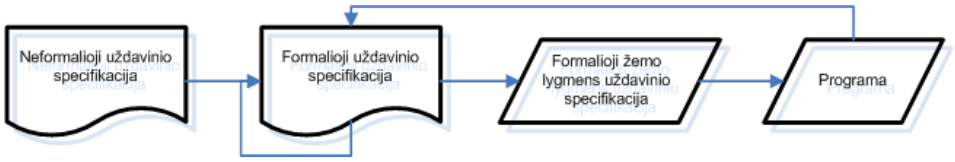
- *GeoLogica* [180]. Į geografinio pobūdžio klausimus atsakinėjanti sistema, taip pat naudojanti ATP *SNARK* ir natūraliosios kalbos analizatorių *Gemini*.

3.2.1. Programų sistemų deduktyviosios sintezės uždavinys

Kaip teigia *R. Bazler* [17], visiškai automatinio programų kūrimo proceso, kuriame nedalyvautų žmogus negali būti, todėl kalbama tik apie automatizuotą programavimą. Pagrindiniai automatizuoto programų sistemų kūrimo etapai yra šie (3.3 pav.):

- neformaliosios uždavinio specifikacijos sudarymas;

- formaliosios uždavinio specifikacijos sudarymas;

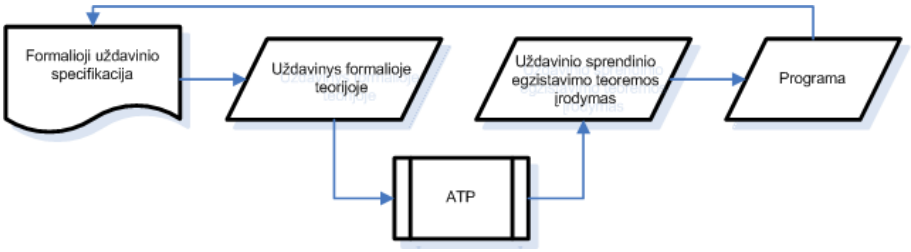


3.3 pav. Supaprastinta *R. Bazler* pasiūlyta automatizuoto programų kūrimo deduktyviosios sintezės metodu schema.

- neformaliosios specifikacijos peržiūra ir korekcija, jei nustatoma, kad tam tikrų elementų formalizuoti neįmanoma;
- formaliosios specifikacijos transformacija į formalų uždavinio aprašą tam tikroje teorijoje (šis aprašas dar vadinamas *žemo lygmens specifikacija*);
- programos sintezė, remiantis formaliąja žemo lygmens specifikacija. Nagrinėjamo uždavinio kontekste sąvokas *sintezė* ir *surinkimas* galima sutapatinti. Yra numatyta ir formaliosios specifikacijos koregavimo galimybė. Koregavimas negali būti visiškai automatinis, todėl formuojant aukšto lygio specifikaciją ir ją transformuojant į žemo lygio specifikaciją būtinas žmogaus dalyvavimas [17, 45].

Egzistuoja ir kita programų sistemų automatizuoto surinkimo iš jau sukurtų komponentų uždavinio sprendimo schema [113]. Joje kalbama tik apie formaliąją uždavinio specifikaciją, kuri dar vadinama tiesiog uždaviniu. Be to tarp formaliosios žemo lygio specifikacijos ir galutinės programos dar gali būti viena ar daugiau tarpinių grandžių (3.4 pav.).

Pirmiausia uždavinio aprašymas (formalioji specifikacija) pateikiamas kuria nors deklaratyviąja kalba. Iš šios specifikacijos gaunama aibė formulių konkrečioje teorijoje. Šioje teorijoje programų kūrimo uždavinys formuluojamas kaip teorema, kurią reikia įrodyti. Jei uždavinio sprendinio egzistavimo teorema įrodoma, tuomet atliekama transformacija *teoremos įrodymas* $\rightarrow$ *programa*.



3.4 pav. Automatizuoto programų kūrimo deduktyviosios sintezės metodu schema.

Realizuojant deduktyviosios sintezės metodą naudojamos *automatinio teoremų įrodymo programos* (ATP) (3.4 pav.). Keletas jų apžvelgiama 3.2.2. skyriuje.

3.2.2. Automatinio teoremų įrodymo programos

Automatinio teoremų įrodymo programa tikrina ar duota formulė (teorema) yra teisinga [152].
Nors apie automatinį įrodymą pradėta kalbėti gana anksti, trečiajame XX a. dešimtmetyje, reikšmingesnių rezultatų pradėta pasiekti tik šeštajame dešimtmetyje [20].

G. Huet 1970-aisiais metais pradėjo SAM įrankio projektą pirmos eilės logikai su ekvivalentumo sąryšiu Tuo metu pažangiausiomis buvo laikomos ATĮ sistemos visus teiginius pervedančios į literalų disjunktų konjunkcijas, kvantorius keičiančios *skoleminėmis formomis*. Toliau šiems konjunktams buvo taikomas rezoliucijos metodas [57]. Procesas dažnai nekonverguodavo ir ATP dažnai darbą baigdavo kai persipildydavo kompiuterio darbinė atmintis [20].

Didesni pasiekimai pastebimi nuo 1970-ųjų, kuomet *D. Knuth* ir *P. Bendix* pasiūlė įrodymo sisteminimo metodiką naudojant baigimo sekas (angl. *termination orders*). Šią metodiką realizavo *J.M. Hullot* ir *G. Huet* [85] ir taip įgalino automatizuoti sprendimus dirbant su algebrinėmis struktūromis. Tuo pačiu metu pradėti siūlyti (pvz. [22]) automatinio įrodymo indukcijos būdu metodai. Be to, rezoliucijų metodas buvo apibendrintas ir pritaikytas aukštesnio lygmens logikoms.

Apie 1980-uosius metus Eidinburgo universiteto mokslininkų grupė realizavo LCF sistemą, kurios vienas esminių bruožų – įrodymo *taktikų* aprašymo ML kalba galimybė.

Svarbią vietą automatinių teoremų įrodymo įrankių raidoje turėjo ir *P. Martin-Löf* konstruktyvioji tipų teorija. Vieni pirmųjų darbų tęsiančių šias idėjas: NuPRL sistema ir „*Programavimo metodologijos*“ projektas vykdytas Čalmerso universitete (*Chalmers university*).

Ilgą laiką λ-skaičiavimas buvo pagrindinis įrankis įrodymų teorijoje. Ypač po to, kai buvo įrodytas *Curry-Howard* izomorfizmas [133, 84] parodantis, kad galima vienareikšmė atitiktis tarp įrodymo ir funkcinių programų struktūrų. 1980-ųjų metų pradžioje buvo pradėtas tyrimas, kurio tikslas – sukurti įrodymų sistemą praplečiančią LCF idėjas ir pritaikyti ML kalbą ne tik kaip taktikų aprašymo metakalbą, bet ir kaip visos įrodymų sistemos realizavimo kalbą. Vienas iš tyrimo rezultatų buvo *Caml* o, dar vėliau, *Objective Caml* kalba. Reminatis *T. Coquand* ir *G. Huet* darbais konstruktyvioji *P. Martin-Löf* tipų teorija pritaikyta *Automath* sistemos plėtiniui – *konstrukcijų skaičiavimui* (angl. *Calculus of Constructions-CoC*). Vėliau *T. Coquand* realizavo *sekvencinio stiliaus* (angl. *sequent-style*) įrodymo sintezės algoritmą kuris

įgalino konstruoti įrodymo termus *progresyviais pertvarkymais* (angl. *progressive refinement*), naudojant taktikų aibes (kaip LCF sistemoje). Vykdant *Formel* [86] projektą buvo nuspręsta CoC taikyti sertifikuotoms programoms gebneruoti, panašiai kaip NuPRL [68] sistemoje. Kuriant sistemą paaiškėjo, kad CoC negalima aprašyti induktyviųjų duomenų struktūrų, todėl buvo sukurtas *induktyviųjų konstrukcijų skaičiavimas* (angl. *Calculus of Inductive Constructions – CoIC*).
Devintojo XX a. dešimtmečio pabaigoje jau buvo žinoma daug teoremų įrodymo programų [182] tokios kaip *Isabelle* [185], *Vampire* [179]ir pan.

Egzistuoja įvairios automatinio teoremų įrodymo programų taksonomijos [179, 152, 182]. Grupės, į kurias skirstomos ATP, priklauso ir nuo to, kokiems tikslams rengiamasi šias programas naudoti. Tam, kad teoremų įrodymo programą būtų galima panaudoti programų sistemų inžinerijos kontekste (programų kūrimui įrodomojo programavimo būdu, programų modelių tikrinimui ar pan.) ji taip pat turi atitikti tam tikrus reikalavimus. Išsamiausią reikalavimų tokioms ATP sąrašą pateikia *J. Schumann* [152]:

1. **Bendrieji reikalavimai**

   - **raiškos galia** Kaip pastebima [152, 182], dažnai efektyviau yra panaudoti jau egzistuojančią ATP, nei kurti naują. Be to, pirmos eilės logikas naudojančios ATP įrodymo kelio ieško greičiau. Tačiau būtina atsakyti į klausimą ar logikos kurioje ATP atlieka veiksmus (vadinsime *tikslo logika*) ir logikos, kurios teoremą norima įrodyti (toliau – *šaltinio logika*) raiškos galia yra vienoda? Ar galima atvaizduoti tas pačias esybes, neprarandant reikšmingos informacijos?

   - **Pagrįstumas ir baigtumas**

     Sprendžiant realaus pasaulio problemas dažnai reikalaujama, kad įrodymo paieška truktų ilgiau nei iš anksto numatytas laikas. Jei įrodymas nerandamas, kyla klausimas ar jis iš viso neegzistuoja ir teorema klaidinga ar tik įrodymui rasti reikia daugiau resursų. Kitaip sakant, kyla klausimas ar ATP gali įrodyti ar paneigti visas teoremas?
     Vertinant ATP darbo baigtumą skiriami du atvejai [152]:

     – *gerybinio nebaigtumo* atveju ATP nepajėgi įrodyti visų teoremų, tačiau per nustatytą laiką randa visus naudotoją dominančius trumpus įrodymas.

     – *piktybinio nebaigtumo* atveju neįrodoma daug svarbių teoremų.

Kartais, siekiant paspartinti įrodymų paieškos procesą apribojamas naudojamų aksiomų kiekis.

- **Patikimumas** Automatinio teoremų įrodymo bei tikrinimo programos turi būti ypač kruopščiai kuriamos, nes net jei tokia programa nustato, kad įrodymas rastas, gali būti gana sunku patikrinti, ar tas įrodymas yra korektiškas.

2. **Jungimosi su ATP savybės** Praktiškai naudojant ATP tampa svarbios komunikavimo su šia programa detalės:

   - **Įrodymo užduoties įvestis**. Ji apima du etapus [152]: užduoties skaitymą ir užduoties paruošimą. Skaitymo metu patikrinama pateikiamų aksiomų ir norimos įrodyti teoremos signatūra.

     Užduoties paruošimo procesą sudaro keletas žingsnių, kurių skaičius ir paskirtis dažnai priklauso nuo pačios ATP programos:

     – sintaksinės transformacijos;
     – formulės transformacijos (dažnai iš aukštesnės eilės logikos į ATP pirmos eilės logiką);
     – aksiomų atranka (angl. *preselection*);
     – supaprastinimas.

   - **ATP startavimas ir stabdymo sparta**

   - **Rezultatų analizė**. Pageidautina ATP savybė – kiek įmanoma išsamesnė informacija apie nesėkmingos įrodymo paieškos priežastis ir įrodymo paieškos kelią (kurios aksiomos panaudotos, o kurios ne ir pan.). Taip pat labai svarbi ir ATP galimybė termus pakeisti jų keitiniais (angl. *substitution*) programų sistemų kūrime yra. Įrodomojo programavimo sistemose, tokiose kaip NUT [175] ir *Amphion* [6] keitiniai naudojami programai generuoti remiantis įrodymu.

3. **Indukcija**. Jei naudojamos rekursyvios duomenų struktūros arba rekursyviosios funkcijos, įrodymas gali būti rastas tik indukcijos būdu [152]. *J. Schumman* nuomone [152] indukcinės ATP geba įrodyti gana

4. **Rūšys** (angl. *Sorts*) Kai kurie formalieji metodų programų sistemų inžinerijoje naudoja tipizuotą arba *sortiruotą* logiką. Tokiu atveju ATP tai pat turi gebėti atlikti operacijas tokioje logikoje.

5. **Aksiomų parinkimas ir generavimas**. ATP pateikiama teorema, kurią reikia įrodyti, aksiomos, hipotezės ir pan. Tokiu būdu susidaro keletas šimtų aksiomų. Aksiomų skaičiaus augimas labai padidina įrodymo paieškos laiką, todėl siekiant tą laiką sutrumpinti kartais apsiribojama tik tam tikru aksiomų poaibiu [175, 152]. Siekiant pagreitinti įrodymo paiešką [152] siūloma kartu su aksiomomis (arba vietoj kai kurių iš jų) įtraukti ir lemas, kurių teisingumu neabejojama.

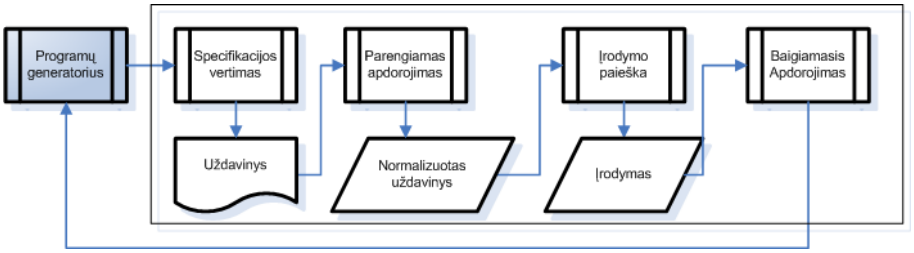
6. **Neteoremų atpažinimas**. Jei ATP neatpažįsta *neteoremų* (t.y. nekorektiškų formulių), ji gali dirbti naudodama itin daug resursų, pailgėja įrodymo paieškos laikas, taip pat gali atvejai kai ATP nebaigia darbo.

7. **Valdymas** ATP darbą galima vertinti pagal keletą kriterijų [152]:

   - ATP turi turėti patogią naudotojams sąsają
   - ATP turi būti sklandi, t.y. mažam įrodymui, naudoti mažai resursų, būti nuspėjama
   - Palaikyti vieną iš ATĮ paieškos strategijų:
     (a) Per fiksuotą laiką T ATP turi išspręsti kiekvieną uždavinį kaip galima greičiau.
     (b) Per fiksuotą laiką T ATP turi išspręsti kiek galima daugiau uždavinių.
   - Paieškos srities dydis.
     Pirmos eilės logikai paieškos laukas yra eksponentinis ir nedeterministinis [152]. Paieškos algoritmai neturi galimybių prognozuoti paieškos laiko ir likusios užduoties (použdavinių) sprendimo trukmės.
   - Parametrų įtaka.
     Kaip pastebima [152] labai sunku parinkti optimalų ATP parametrų skaičių. Jei parametrų per mažai - programa nelankstus, jei per daug - dažnai neišnaudojamos visos jos galimybės, o parinkus vienas kitam prieštaraujančius parametrus galima neigiamai paveikti įrodymo paieškos procesą.

8. **Papildomi reikalavimai**: stabilumas, robastiškumas, išsami dokumentacija.

J. Schumann reikalavimų automatinio teoremų įrodymo programoms sąrašas yra išsamus, tačiau atsižvelgiant į kontekstą jis gali kisti. Pastebėtina, kad net



3.5 pav. Automatinio teoremų įrodymo etapai.

NORA/HAMMR sistemoje naudojama ATP tenkina ne visus reikalavimus. Komponentinių programų sistemų kūrimo kontekste ATP apsiribosime tik šiais reikalavimais:

- ATP naudojamo vidinio formalizmo bei uždavinio specifikavimo kalbos *raiškos geba* turi būti pakankama komponento modelio statiniam ir dinaminiam aspektams išreikšti.
- teoremų įrodymo procesas turi būti visiškai automatinis, t.y. be žmogaus įsikišimo nuo įrodymo paieškos proceso pradžios iki pabaigos. Rankiniu būdu gali būti pateikti tik pradiniai duomenys (uždavinio teorija, paieškos algoritmas, taktika ir pan.) prieš proceso pradžią.
- Turi būti numatytas ATP ryšys su kitomis programomis (pvz., per API biblioteką), ATP turi būti skirta tai pačiai operacinei sistemai (OS), kuriai skirtas ir ją naudosiantis programų sistemų generavimo įrankis.

Pagal šiuos kriterijus paanalizuokime keletą automatinių teoremų įrodymo programų (3.4 lentelė):

1. **HERBY ir THEO**.

   M. Newborn [127] aprašo dvi automatinio teoremų įrodymo programas:

   - HERBY – semantinio medžio metodą naudojanti teoremų įrodymo programa.
   - THEO – rezoliucijų-atmetimo (angl. *refutation*) metodą naudojanti TĮP.

   Specifikacijoms konvertuoti į minėtoms programoms priimtiną formatą naudojama trečia programa - COMPILE.

2. **Coq**.

   Naudojant *Coq* [20] programą termų, tipų, įrodymų ir programų aprašymui naudojama speciali kalba *Galina* ir komandinės eilutės įrankis *Vernacular*. Kaip vidinis formalizmas naudojamas *CoIC* skaičiavimas. *Coq* branduolys kiekvieną sintezuotą įrodymo termą pertikrina remdamasis vartotojo pasirinkta *taktika*. Jau *Coq* 4.1 versija pasižymėjo galimybe iš įrodymų „išgauti" funkcines programas *Caml* kalba, vėliau pateikiama ir grafinė *CTCoq* sąsaja, leidžianti vartotojui nurodyti, kuriame įrodymo paieškos taške, kurias taktikas taikyti.
   Coq septintoji versijoje tapo įmanoma naudoti aukštesnio lygmens logikas, *CTCoq* sąsaja pakeista nauja, Java programavimo kalba sukurta PCoq sąsaja. Be to, panaudotas visiškai naujas įrodymo paieškos algoritmas. *Coq* pagrindu pradėti kurti kiti projektai [20]:

   - *CALIFE* – laiko automatų modeliavimo įrankis;
   - *Why* – imperatyviųjų programų kūrimo įrankis;
   - *Krakatoa* – Java įskiepių tikrinimo įrankis

   Coq – interaktyvus ATP, todėl yra netinkamas visiškai automatiniam teoremų įrodymui.

3. **Isabelle** – bendroji sistema loginių formalizmų realizavimui, sukurta naudojant ML kalbą [185, 139, 130]. Naudojama *ProofGeneral* [9] vartotojo sąsaja kuri savo ruožtu naudoja *Emacs* tipo teksto redaktorių. Uždaviniai aprašomi ir koordinuojami *Isar* kalba. Taip pat *Isabelle* gali būti naudojama ir iš kitų programų per komandinės eilutės kreipinius. Deja, *Isabelle* negali būti naudojama visiškai automatiniam teoremų įrodymui. Taip pat kaip ir *Coq, Issabelle* yra interaktyvus teoremų įrodymo įrankis.

   Isabelle sudėtiniai moduliai yra:

   - **bazinis įrodymų paieškos modulis** (angl. *classical reasoner*),
   - **supaprastinimo modulis** (angl. *simplifier*) galintis normalizuoti lygtis,
   - išorinių teoremų įrodymo programų panaudojimo modulis **Sledgehammer**, skirtas použdotį konvertuoti į pasirinktai išorinei įrodymo programai priimtiną formatą ir ją perduoti bei priimti rezultatus. Galimos jungtys su *E*, *SPASS* ir *Vampire* teoremų įrodymo programomis [185].

Isabelle naudoja plačią elementarios skaičių teorijos, analizės, algebros ir aibių teorijos aksiomų ir teoremų biblioteką. Ši biblioteka praplečiama ir internete prieinama spręstų uždavinių, jų įrodymų ir kitos medžiagos žinių baze [8]. Be to bibliotekos elementai aprašyti *SML* kalba, todėl *Isabelle* ATP galimybės gali būti plečiamos aprašant naujas teorijas kitoms dalykinėms sritims.

Plačiausiai naudojama šios teoremų įrodymo programos versija **Isabelle/ HOL** skirta specifikavimui ir tikrinimui aukštesnio lygmens logikoje HOL [130]. *Isabelle/ HOL* leidžia vykdomąsias specifikacijas konvertuoti į *SML*, *OCaml* ar *Haskell* programavimo kalbų kodą. Taip pat gali būti naudojamos ir kitos *Isabelle* atšakos darbui su pirmos eilės, *ZF*, skaičiuojamųjų funkcijų (*LCF*), sekventų, *Lambda Cube* logikomis bei konstruktyviąja tipų teorija (CTT) [130]. Be to programos autoriai sudarė galimybes ir kitų atšakų kūrimui naudojant *Isabelle/Pure* metalogiką.

Analizė parodė, kad ne visos ATP yra visiškai automatinės (3.4 lentelė). Be to, gana ribotos jų ryšio su kitomis programomis galimybės. Nors yra ATP, kurių pirminį programos tekstą galima keisti, naudojimo galimybes riboja operacinės sistemos pasirinkimas.

3.3. Struktūrinės programų sintezės metodas

Struktūrinė sintezė - dedukcijos principu grindžiamas programų sintezės metodas. Metodo pagrindas yra *Curry-Howard* izomorfizmas, užtikrinantis vienareikšmę atitiktį tarp uždavinio sprendinio teoremos konstruktyviojo įrodymo ir programos [84].

Nuo kitų deduktyvinių sintezės metodų, struktūrinė sintezė skiriasi tuo, kad konstruoti programas galima remiantis vien jų struktūrinėmis savybėmis. Tokiu atveju uždavinio sąlygą galima nusakyti aprašant turimų komponentų ir būsimos programų sistemos struktūrines savybes. Klasikinis struktūrinės sintezės metodas leidžia atlikti operacijas vienoje iš trijų teorijų [2]:

- λ-skaičiavime,
- konstruktyviojoje tipų teorijoje,
- intuicionistinėje logikoje.

3.3.1. Struktūrinės sintezės skaičiavimo modeliai

Tarkim $S_1, S_2, \ldots, S_k$ - aibės.

$$< s_1, s_2, \ldots s_k > \in S_1 \times S_2 \times \ldots \times S_k$$

Bet koks Dekarto sandaugos $S_1 \times S_2 \times \ldots \times S_k$ poaibis vadinamas sąryšiu. Sakoma, kad rinkinys $< l_1, \ldots l_k >$ tenkina sąryšį R, jeigu $< l_1, \ldots l_k > \in R$.

Lygtis $f(x_1, \ldots x_k) = 0$ apibrėžia sąryšį tarp kintamųjų $x_1, \ldots x_k$.

Jei $\mu_1, \mu_2, \ldots, \mu_k$ yra kintamųjų $x_1, \ldots x_k$ reikšmių kitimo sritys, tai bet kuris sąryšis $R \subseteq \mu_1 \times \mu_2 \times \ldots \times \mu_k$ yra vadinamas sąryšiu tarp kintamųjų $x_1, \ldots x_k$.

Panagrinėkime sąryšį $R(x)$. Tegu $u \subseteq v$, $v \subseteq x$. Sąryšis $R(x)$ apibrėžia kintamųjų u ir v (bendruoju atveju - nevienareikšmį) atvaizdį:

$$\phi_{R,u,v} : \mu_u \mapsto \mu_v,$$

kuris kiekvienai kintamojo u reikšmei **u** parenka tokią kintamojo v reikšmę **v**, kuri kartu su **u** yra kažkurio sąryšio R elemento dalis:

$$\phi_{R,u,v}(u) = \{\mathbf{v} | (\exists \mathbf{e})(\mathbf{e} \in R \wedge \mathbf{u} \subseteq \mathbf{e} \wedge \mathbf{v} \subseteq \mathbf{e}\}$$

Siekiant supaprastinti realizaciją apsiribojama tik funkciniais sąryšiais. Darbe [167], funkcinis dviejų aibių sąryšis vadinamas *operatoriumi*. Kitaip tariant, operatoriumi ϕ vadinama baigtinė priskyrimų

$$y_i := f_i(x_1, \ldots x_m),\ i = 1, 2, \ldots n$$

aibė. Čia $x_1, \ldots x_m$ - įeinantys kintamieji, o $y_1, \ldots y_n$ - išeinantys kintamieji. Jų aibės žymimos $in(\phi)$ ir $out(\phi)$ atitinkamai. Operatoriai $\phi_1, \ldots \phi_k$ vadinami *neprieštaringais*, jeigu taikant skirtingus operatorius (arba keičiant jų vykdymo tvarką) iš tų pačių įeinančių kintamųjų reikšmių, gaunamos tos pačios išeinančių kintamųjų reikšmės.
Daliniu sąryšiu vadinama baigtinė neprieštaringų operatorių aibė. Pastebima [168], kad atskiras operatorius visada sudaro dalinį sąryšį, jei $in(\phi) \cap out(\phi) = \emptyset$ Nagrinėjamas dalinis sąryšis R(x), nusakytas operatorių aibe $\Phi = \{\phi_1, \ldots, \phi_k\}$. *Kintamasis* $u \in x$ sąryšiui $R(x)$ vadinamas:

- **įeinančiu**, jei egzistuoja toks $\phi' \in \Phi$, kuriam $u \in in(\phi')$ ir $u \notin out(\phi)$, jokiam ϕ.
- **išeinančiu**, jei egzistuoja toks $\phi' \in \Phi$, kuriam $u \in out(\phi')$ ir $u \notin in(\phi)$, jokiam ϕ.
- **silpnai susietu**, jei egzistuoja tokie $\phi' \in \Phi$, $\phi'' \in \Phi$, kuriems $u \in in(\phi')$ ir $u \notin out(\phi'')$.

Dalinio *sąryšio rangu* $Rank(R)$ vadinamas minimalus šio sąryšio operatoriaus išeinančių kintamųjų skaičius.

Daliniai sąryšiai, pasižymintys maksimaliu galimu skaičiumi operatorių su maksimaliu galimu skaičiumi išeinančių kintamųjų, esant duotam rangui

$$Rank(R(x))$$

yra ypatingi. Visi susieti tokio sąryšio kintamieji – silpnai susieti, o operatorių skaičius - C_r^n, kur r - rangas, n - sąryšio susietų kintamųjų skaičius. Tokie sąryšiai yra lygčių sistemos apibendrinimas (arba atskirų lygčių, kai $r = 1$), jie vadinami *visiškai simetriniais sąryšiais*. Pastaruosius apibendrina *simetriniai sąryšiai*. Kiekvienam simetrinio sąryšio $R(x)$ operatoriui teisingos lygybės:

$$in(\phi) \cup out(\phi) = x \tag{3.2}$$

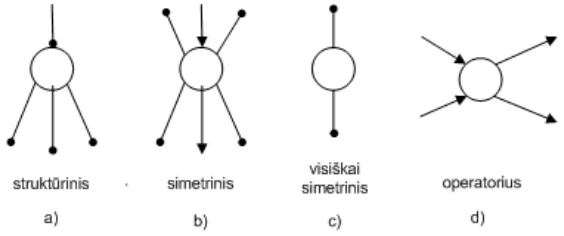
$$|out(\phi)| = Rank(R) = r \tag{3.3}$$

Tarkime, kad k – išeinančių simetrinio sąryšio kintamųjų skaičius. Tada bet kuriam deriniui po $r - k$ šio sąryšio silpnai susietų kintamųjų atsiras operatorius, kuriam šie kintamieji yra įeinantys. Ribinis atvejis yra kai $r = k$, tada sąryšis neturi susietų kintamųjų ir turi tik vieną operatorių. Bet kuris dalinis sąryšis, turintis tik vieną operatorių, yra simetrinis.
Struktūriniu sąryšiu vadinamas sąryšis turintis tik silpnai susietus kintamuosius, iš kurių vienas yra struktūrinis. m kintamųjų siejantis struktūrinis sąryšis turi lygiai m operatorių, kurie aprašomi taip:

$$x := < x_1, \ldots, x_{m-1} >, \tag{3.4}$$

$$x_i := select_i(x),\ i = 1, \ldots, m-1, \tag{3.5}$$

3.6 pav. SSP sąryšiai [167].



3.7 pav. SSP sąryšių tipai [167].

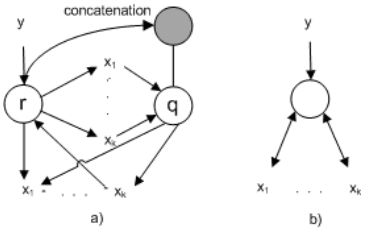
kur $select_i$ yra funkcija apskaičiuojanti i-ojo sudėtinio kintamojo elemento reikšmę.

E. Tyugu grupės mokslininkai [167, 168] *skaičiuojamuoju modeliu* vadina semantinį tinklą, kurį sudaro kintamieji susieti vienareikšmiais daliniais sąryšiais[2]. Tarkime, kad M - skaičiuojamasis modelis, tada jo kintamųjų aibė žymima $\mathbf{var}(M)$, o sąryšių aibė – $\mathbf{rel}(M)$.
Skaičiuojamieji modeliai gali būti interpretuojami kaip grafai [168, 167].

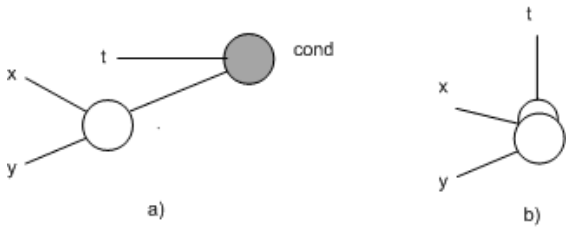
Pateiktas skaičiuojamasis modelis neleidžia specifikuoti sąlygų, kada operatorius galima taikyti. Siekiant panaikinti šį apribojimą, jau PRIZ sistemoje įvesti *valdymo kintamieji* [168, 167]. Grafe valdymo kintamuosius atitinka valdymo mazgai. Valdymo mazgai yra dviejų rūšių:

- ***cond*-mazgai**. Valdymo mazgas – valdymo kintamasis susietas su vienu

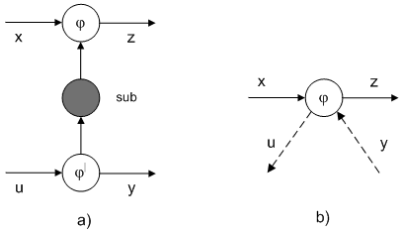
[2]Dalinis sąryšis vadinamas vienareikšmiu, jeigu visos to sąryšio operacijos yra vienareikšmės [167].



3.8 pav. Valdymo taškas SSP skaičiuojamajame modelyje [167].



3.9 pav. SSP sąlygos sąryšis [167].



3.10 pav. SSP použdavinio sąryšis [167]

sąryšiu. Šio kintamojo reikšmė rodo ar sąryšis galimas (3.9 pav.). Praplėtus skaičiuojamąjį modelį *cond* valdymo elementais, gautasis modelis yra pakankamas kad jame būtų galima išreikšti bet kurį algoritmą [167].

- ***sub*-mazgai**. Jie susieja (*angl. binds*) skaičiuojamojo modelio M sąryšį R ir sąryšį iš $rel(M)$. Pastarasis sąryšis vadinamas sąryšio R *použdaviniu* (3.10 pav.). Skaičiuojamieji modeliai kuriuos sudaro tik *sub* valdymo elementais, struktūriniais sąryšiais, ir sąryšiais: „*konstanta 0*“„,*plius 1*“, primityvios rekursijos operatorius, minimizavimo operatorius, yra pakankami kad jais būtų galima išreikšti bet kurią rekursyvią funkciją. Kiekvienai rekursyviai funkcijai gali būti sintezuojama programa, kuri apskaičiuoja funkcijos reikšmę [167].

Tipinis struktūrinės sintezės uždavinys yra

$$\text{Rasti } r_1, r_2, \ldots, r_s \text{ pagal } p_1, p_2, \ldots, p_l \text{ žinant } M,$$

kur M - skaičiuojamasis modelis, $r_1, r_2, \ldots, r_s$ - rezultatai (išeinantys kintamieji), $p_1, p_2, \ldots, p_l$ - argumentai (įeinantys kintamieji).

3.3.2. Struktūrinės sintezės algoritmai

Pirmoji sistema, kurioje realizuotas struktūrinės sintezės metodas – PRIZ. Įrodymo planavimo modulis buvo sukurtas 1975-1977 m. [169], o aštuntojo

dešimtmečio pradžioje galutinai suderintas [176]. PRIZ sistemoje siekiant paspartinti sintezės procesą naudojamos tik specialios formos aksiomos [166]. Jose išreikštiniu būdu naudojamas apskaičiuojamumo sąryšis: „*naudojant funkciją f iš u galima gauti (apskaičiuoti) v* “:

$$u \overset{f}{\mapsto} v.$$

Prie apskaičiuojamumo sąryšio prijungus formulę Γ, kuri išreiškia apskaičiuojamumo sąlygas, gaunamas apskaičiuojamumo sakinys:

$$\Gamma \vdash u \overset{f}{\mapsto} v, . \tag{3.6}$$

Kiekviena teorijos aksioma turį tokį pavidalą:

$$\mathcal{R} \vdash (\Gamma \vdash u \overset{f}{\mapsto} v),$$

kur sąryšis $\mathcal{R} = (R_1, R_2, \ldots R_n)$ rodo sąlygas, kuriomis galima išvesti apskaičiuojamumo sakinį.

3.3.2.1. Išvedimo taisyklės

Teoremos įrodymui naudojamos penkios išvedimo taisyklės:

$$\frac{y \subseteq x}{x \overset{selectx,y}{\longmapsto} y} \tag{3.7}$$

$$\frac{x \overset{f_1}{\mapsto} y, y \overset{f_2}{\mapsto} z}{x \overset{f_1;f_2}{\longmapsto} z}, kur \quad (f_1; f_2)(x) = f_2(f_1(x)) \tag{3.8}$$

$$\frac{x \overset{f_1}{\mapsto} y, x \overset{f_2}{\mapsto} z, \omega = y \cup z}{x \overset{f_1,f_2}{\longmapsto} \omega}, kur \quad (f_1, f_2)(x) = f_1(x) \cup (f_2(x)) \tag{3.9}$$

$$\frac{\Gamma, \Gamma \vdash x \overset{f}{\mapsto} y}{x \overset{f}{\mapsto} y} \tag{3.10}$$

$$\frac{R(a), R(z) \vdash (\Gamma \vdash x \overset{f}{\mapsto} y)}{\Gamma' \vdash x' \overset{f}{\mapsto} y'}, \tag{3.11}$$

kur Γ', x', y' gauti vietoj a įstačius z. Kaip teigiama [172], šių taisyklių pakanka išvesti bet kuriai *primityviai rekursyviai* funkcijai, aprašytai aksiomomis:

$$a \overset{f}{\mapsto} b \tag{3.12}$$

ir

$$\forall \phi (c \overset{\phi}{\mapsto} d \Rightarrow a \overset{G(\phi)}{\mapsto} b) \tag{3.13}$$

Teoremos įrodymo paieška sudaro du etapai:

1. Taikant penktąją (3.11) išvedimo taisyklę iš aksiomų išvedami apskaičiuojamumo sakiniai $\Gamma \vdash x \overset{f}{\mapsto} y$,.
2. Atliekama teoremos įrodymo paieška. Nauji sakiniai išvedami taikant pirmąją (3.7), antrąją (3.8) ir trečiąją (3.9) taisykles.

3.3.2.2. Tiesinės struktūros programų sintezė: algoritmas $\mathbf{A}_1$

1 žingsnis. $\omega' := u$

2 žingsnis. Taikoma pirmoji (3.7) išvedimo taisyklė siekiant gauti

$$\frac{v \subseteq \omega'}{\omega' \mapsto v}$$

.

Jei taisyklė taikytina, tai teorema įrodyta. Priešingu atveju

$$\omega := \omega'.$$

3 žingsnis. Ieškoma tokio apskaičiuojamumo ryšio

$$x \overset{f}{\mapsto} y,$$

kuriam

$$x \subseteq \omega \wedge y \not\subseteq \omega.$$

4 žingsnis. jei tokio ryšio nėra, tai teorema neįrodoma.

5 žingsnis. Taikoma antroji (3.8) išvedimo taisyklė siekiant gauti naują sakinį:

$$\frac{\omega \overset{f_1}{\mapsto} x, x \overset{f_2}{\mapsto} y}{\omega \overset{f_1, f_2}{\mapsto} y}$$

Taikoma trečioji (3.9) išvedimo taisyklė siekiant gauti naują būseną:

$$\frac{x \overset{f_1}{\mapsto} y, x \overset{f_2}{\mapsto} \omega, \omega' = \omega \cup y}{x \overset{f_1, f_2}{\longmapsto} \omega'}$$

6 žingsnis. Pereinama prie 2-ojo žingsnio.

Pastabos.

- 3-ame žingsnyje vyksta eilinio apskaičiuojamumo sąryšio parinkimas. Supaprastintoje schemoje tai daroma perrenkant visus galimus variantus, tačiau, kaip pažymima [166], kitų euristinių metodų naudojimas gali procesą pagreitinti.
- Algoritmas A_1 visada baigia darbą nes kiekvienam u ir sąlygai $\mathcal{R}$ klasės $\mathcal{T}_\infty$ teorijoje egzistuoja baigtinė būsena, kuri pasiekiama (suskaičiuojama) atlikus baigtinį žingsnių skaičių.
- Jei taikant algoritmą A_1 teorema neįrodoma, tai jos įrodymas neegzistuoja.

3.3.2.3. Besišakojančių programų sintezė: algoritmas $\mathbf{A}_2$

1 žingsnis. $\omega' := u$

2 žingsnis. Taikomas algoritmas A_1 siekiant įrodyti teoremą

$$\mathcal{R} \vdash \exists f(\omega' \overset{f}{\mapsto} v)$$

ir apskaičiuoti būseną ω, naudojant tik apskaičiuojamumo sakinius su tapačiai teisingomis apskaičiuojamumo sąlygomis. Jei teorema įrodyta, tai darbas baigiamas.

3 žingsnis. Jei nėra nei vieno skaičiuojamojo sakinio priklausančio tipui

$$P(z) \vdash x \mapsto y,$$

kuriam

$$z \subseteq \omega \wedge x \subseteq \omega \wedge y \not\subseteq \omega,$$

tai teorema

$$\mathcal{R} \vdash \exists f(u \overset{f}{\mapsto} v)$$

neįrodoma. Priešingu atveju su kiekvienu tokiu sakiniu

$$P(z) \vdash x \mapsto y$$

atliekami šie veiksmai:

3.1 Daroma prielaida, kad teisingas $P(z)$;

3.2 Rekursyviai taikant algoritmą A_2, ieškoma teoremos

$$\mathcal{R}' \vdash \exists f(\omega \overset{f}{\longmapsto} v)$$

įrodymo. ($\mathcal{R}'$ gautas iš $\mathcal{R}$ pakeičiant $P(z) \vdash x \mapsto y$ į $\vdash x \mapsto y$).

5 žingsnis. Jei nei viena iš teoremų neįrodoma, tai ir teorema

$$\mathcal{R} \vdash \exists f(u \overset{f}{\mapsto} v)$$

neįrodoma. Priešingu atveju visiems sakiniams $\omega \overset{f_i}{\longmapsto} v$, kur $i = 1, \cdots, k$, išvestiems teoremos

$$\mathcal{R}' \vdash \exists f(\omega \overset{f}{\mapsto} v)$$

įrodymo metu taikoma 4 taisyklės (3.10) modifikacija:

$$\frac{P_1(z_1) \vdash \omega \overset{f_1}{\mapsto} v, \ldots, P_k(z_k) \vdash \omega \overset{f_k}{\longmapsto} v}{\omega \overset{f}{\longmapsto} v}$$

kur f=**if** $p_1(z_1)$ **then** f_1 **elsif** $p_2(z_2)$ **then** f_2**elsif**... **elsif** $p_k(z_k)$ **then** f_k **else** failure **if**.

6 žingsnis. Taikoma antroji (3.8) išvedimo taisyklė siekiant išvesti sakinį $u \overset{f}{\longmapsto} v$:

$$\frac{u \overset{f_1}{\mapsto} \omega, \omega \overset{f_2}{\mapsto} v}{u \overset{f}{\longmapsto} v}$$

7 žingsnis. Teorema įrodyta.

Pastabos.

- Algoritmas A_2 visada baigia darbą, nes algoritmas A_1 baigia darbą klasės $\mathcal{T}_1$ teorijose, o kiekviena $\mathcal{T}_2$ klasės teorija yra tam tikros $\mathcal{T}_1$ teorijos praplėtimas prijungiant prie jos baigtinį aksiomų skaičių, kurių kiekviena įrodyme taikoma baigtinį kartų skaičių.

- Jei teorema $\mathcal{R} \vdash \exists f(u \overset{f}{\mapsto} v)$ įrodoma, tai išvestąjį sakinį $u \overset{f}{\mapsto} v$ atitinka besišakojanti programa, kuri apima procedūras, neturinčias procedūros tipo parametrų, nuoseklieji operatoriai ir sąlygos operatoriai.

- Kiekviena tokiu būdu gauta programa baigia darbą, nes visi teorijos aksiomomis aprašyti operatoriai yra pilni. Bet tokio uždavinio išsprendžiamumas priklauso nuo pradinių duomenų.

3.3.2.4. Programų su použdaviniais sintezė: algoritmas $\mathbf{A}_3$

1 žingsnis. $\omega'_i := u$

2 žingsnis. Taikomas algoritmas A_1 siekiant įrodyti teoremą

$$\mathcal{R} \vdash \exists f(\omega' \overset{f}{\mapsto} v)$$

ir apskaičiuoti būseną ω, naudojant tik apskaičiuojamumo sakinius su tapačiai teisingomis apskaičiuojamumo sąlygomis. Jei teorema įrodyta, tai darbas baigiamas.

3 žingsnis. Ieškoma tokio sakinio $\Gamma \vdash x \overset{f}{\mapsto} y$, kuriam

$$x \subseteq \omega \wedge y \not\subseteq \omega,\ \Gamma = \exists f(u_1 \overset{f}{\mapsto} v_1) \bigwedge \ldots \bigwedge \exists f(u_k \overset{f}{\mapsto} v_k)$$

rekursyviai taikant algoritmą A_3 kiekvienam použdaviniui iš Γ įrodoma $\mathcal{R} \vdash \exists f(u_i \overset{f}{\mapsto} v_i)$

4 žingsnis. Jei tokio sakinio nėra, tai pereinama prie 7 žingsnio. Priešingu atveju taikoma 4-oji (3.10) išvedimo taisyklė, siekiant išvesti sąryšį $x \overset{f}{\mapsto} y$ iš skaičiuojamojo sakinio $\Gamma \vdash x \overset{f}{\mapsto} y$.

5 žingsnis. Rekursyviai taikant algoritmą A_3 konstruojamas teoremos $\mathcal{R}' \vdash \exists f(\omega \overset{f}{\longmapsto} v)$ įrodymas.

6 žingsnis. Jei teorema $\mathcal{R} \vdash \exists f(\omega \overset{f}{\longmapsto} v)$ įrodoma, tai pereinama prie 10 žingsnio. Priešingu atveju teorema neįrodoma, darbas baigiamas.

7 žingsnis. Kiekvienam tokiam sakiniui $\Gamma \vdash x \overset{f}{\mapsto} y$, kurio suskaičiuojamumo sąlygos Γ apima realizuojamą predikatą $P(z)$, kuriam

$$x \subseteq \omega \wedge x \subseteq \omega \wedge u \not\subseteq \omega$$

atliekami šie veiksmai:

7.1 Daroma prielaida, kad predikatas $P(z)$ teisingas.

7.2 Rekursyviai taikant algoritmą A_3 ieškoma teoremos

$$\mathcal{R}' \vdash \exists f(\omega \overset{f}{\longmapsto} v)$$

įrodymo. $\mathcal{R}'$ gaunamas taip pat, kaip ir A_2 algoritmo 3.2 žingsnyje.

8 žingsnis. Jei neegzistuoja toks skaičiuojamumo sakinys $\Gamma \vdash x \overset{f}{\mapsto} y$, kurio skaičiuojamumo sąlygos Γ turėtų predikatą $P(z)$, kuriam

$$z \subseteq \omega \wedge x \subseteq \omega \wedge u \not\subseteq \omega,$$

tai teorema neįrodoma.

9 žingsnis. Jei nei viena 7.2 žingsnyje nagrinėjamų teoremų neįrodoma, tai uždavinys neišsprendžiamas. Priešingu atveju sakiniams

$$P_1(z_1) \vdash \omega \overset{f_1}{\mapsto} v, \ldots, P_k(z_k) \vdash \omega \overset{f_k}{\mapsto} v,$$

atitinkantiems teoremas $\mathcal{R}' \vdash \exists f(\omega \overset{f}{\longmapsto} v)$ taikoma 4-oji (3.10) taisyklė:

$$\frac{P_1(z_1) \vdash \omega \overset{f_1}{\mapsto} v_1, \ldots, P_k(z_k) \vdash \omega \overset{f_k}{\longmapsto} v_k}{\omega \overset{f}{\longmapsto} v},$$

kur f=**if** $p_1(z_1)$ **then** f_1 **elsif** $p_2(z_2)$ **then** f_2**elsif**... **elsif** $p_k(z_k)$ **then** f_k **else** failure **if**.

10 žingsnis. Taikoma 2-oji (3.8) išvedimo taisyklė siekiant išvesti $u \overset{f_1}{\mapsto} v$:

$$\frac{u \overset{f_1}{\mapsto} \omega, \omega \overset{f_2}{\mapsto} v}{u \overset{f}{\longmapsto}} v$$

11 žingsnis. Teorema įrodyta.

Pastabos. Kiekvieną įrodyme naudojamą sakinį $\Gamma \vdash x \overset{f}{\mapsto} y$, kurio skaičiavimo sąlygose yra použdavinių, generuojamoje programoje atitinka procedūros realizuojančios funkciją f iškvietimas ir použdavinių sprendimo procedūrų aprašymas. Jei teorema $\mathcal{R} \vdash \exists f(u \overset{f}{\longmapsto} v)$ įrodoma, tai įrodymo būdu išvestąjį sakinį $u \overset{f}{\longmapsto} v$ atitinka programa su paprogramėmis, kurią sudaro procedūros, nuoseklieji ir sąlyginiai operatoriai ir uždavinio sprendimo procedūrų aprašymai.

3.3.2.5. Duomenų srautai: algoritmas $\mathbf{A}_4$

Skaičiuojamąjį modelį galima nagrinėti kaip duomenų srautų modelį (*angl. data flow*). Tokiu atveju galima naudoti dar vieną sprendinio paieškos algoritmą [167].

Kiekvienam operatoriui priskiriamas kintamasis (skaitliukas), kurio reikšmė parodo, kelių argumentų reikšmės yra dar neapskaičiuotos. Algoritmas seka

kintamuosius kurių reikšmės tapo žinomos, tačiau nėra patikrinta ar jos bus naudojamos. Tokie kintamieji surašomi į aibę N. Pirmieji aibės N elementai yra uždavinio argumentai. Algoritmas baigia darbą tada, kai N tampa tuščia aibe.

Iš aibės N imamas kintamasis (tarkim x) ir su juo atliekami tokie veiksmai:

1. Jei kintamasis negali būti nei vieno operatoriaus argumentu (t.y. grafe iš kintamąjį atitinkančio mazgo nėra išėjimo rodyklės), tai jis pašalinamas iš aibės N ir daugiau nebenaudojamas.

2. Jei kintamasis gali būti kokio nors operatoriaus argumentu (t.y. grafe iš kintamąjį atitinkančio mazgo yra bent vienas išėjimo lankas), tai šis lankas pašalinamas (arba pažymimas kaip nepasiekiamas), o operatoriaus mazgas į kurį jis veda, patikrinamas. Jei operatoriaus skaitliuko reikšmė yra didesnė už 1, tai ji vienetu sumažinama.
   Jei ši reikšmė lygi 1, tai kintamasis x yra paskutinis argumentas, kurio reikėjo operatoriui. Tokiu atveju operatorius įvykdomas, o jo rezultatai-kintamieji į traukiami į aibę N.
   Šis žingsnis taikomas kiekvienam lankui vedančiam iš nagrinėjamo mazgo.

3. Procesas kartojamas kiekvienam aibės N kintamajam.

Pastabos. Kaip duomenų srautų modelį galima nagrinėti ir skaičiuojamąjį modelį su použdaviniais. Tačiau tuo atveju į použdavinį gali būti kreipiamasi keletą kartų. Dėl šios priežasties labai sumažėja algoritmo sparta. Algoritmo sudėtingumas yra polinominis atminties atžvilgiu (PSPACE), kaip ir naudojant teoremų įrodymą intuicionistiniame teiginių skaičiavime [167, 176].

3.3.3. Struktūrinės programų sintezės realizacijos

Struktūrinės sintezės metodas naudojamas jau daugiau kaip 20 metų. Per tą laiką jis buvo pritaikytas tiek struktūrinei, tiek objektinei, tiek paslauginei paradigmoms. Šiame poskyryje bus apžvelgti konkretūs jo taikymai.

3.3.3.1. Klasikinė struktūrinės sintezės sistema

Kuriant sistemą PRIZ, jai buvo keliami tokie reikalavimai [167]:

- Sistema turi išsaugoti ir naudoti visas bent kartą jai suteiktas žinias.

- Programos yra nedalomos esybės skirtos atlikti konkretiems veiksmams *(angl. actions)*. Jos gali būti sukurtos bet kuria programavimo kalba (Pvz.: Fortran, Cobol, Assembler ir t.t.)

- Programos „jungiamos“ tik per parametrus ir neturi sukelti jokio „pašalinio efekto“. Sinchronizacijos kintamieji taip pat laikomi programų parametrais. Galutinis tipų patikrinimas (angl. *type checking*) turi būti atliekamas programų lygmenyje o ne žinių atvaizdavimo lygmenyje.

PRIZ sistemoje skaičiuojamasis modelis ir uždavinys specifikuojamas UTOPIST kalba [167, 168]. UTOPIST programą sudaro skaičiuojamojo modelio specifikacija (**let** blokas) ir teorema, kurią reikia įrodyti (**actions** blokas). Pvz.:

program Pirmoji
let X,Y,Z:numeris;
X + Y = 2.5
Z = X - Y;
actions $\vdash$ X $\mapsto$ Z;
end;

UTOPIST kalba užrašytose specifikacijose naudojama operacinė semantika. Semantinės kalbos programa sudaryta iš dviejų dalių [168, 167]:

- **programos modelio**, kurį sudaro visi programos kintamieji ir operatoriai. Kintamųjų tipus ir operatorių tinkamumą apibrėžia semantinė mašina. Programos modelis yra skaičiuojamasis modelis kuriame kiekvienas sąryšis turi tik vieną operatorių.

- **valdymo medžio**, kuris nurodo operatorių vykdymo tvarką. Tokio medžio šaknimi yra programa, o jo terminaliniai mazgai – operatoriai, kuriuos tiesiogiai gali vykdyti abstrakčioji mašina. Šie operatoriai yra vaizduojami ir programos modelyje. Neterminaliniai mazgai yra valdymo mazgai. Jei valdymo tipas sudėtinis, jį gali sudaryti keletas valdymo kintamųjų. Labai svarbu kokia tvarka iš neterminalinės viršūnės lankai eina į kitas viršūnes. Sukeitus lankus vietomis gali būti gaunama visai kita, specifikacijos neatitinkanti programa.

Valdymo medžio neterminalinėmis viršūnėmis gali būti tik keturių tipų valdymo mazgai:

- *seq* (nurodo kokia tvarka vykdomi operatoriai),

- *par* (nurodo kad vykdymo tvarka nesvarbi, galimas ir lygiagretusis vykdymas),

- *case* (nurodo kokioms sąlygoms esant, kurie operatoriai turi būti vykdomi),

- *iter* (nurodo kad operatorius turi būti vykdomas tol, kol valdymo kintamasis įgis neigiamą reikšmę).

O tai, kaip pastebima [167], daro semantinę kalbą specializuotąja kalba. Siekiant išvengti šio apribojimo programos modelis papildomas sąryšiais su použdaviniais. Tokie daliniai sąryšiai gali pakeisti valdymo mazgus. Tokiu atveju neterminalinės viršūnės atitinka programas, kurios kviečia kitas programas. (Vykdymo logika tampa tų programų vidine savybe kuri SSP nenagrinėjama). Taigi, valdymo tipai *seq, par, case, iter* keičiami iš anksto sukurtais operatoriais su použdaviniais.

3.3.3.2. Objektinės struktūrinės sintezės sistemos

Struktūrinės sintezės metodas realizuotas dviejose sistemose skirtose objektinėms programų sistemoms kurti: NUT sistemoje ir kitoje Kibernetikos Institute Estijoje sukurtoje sistemoje skirtoje objektinių programų Java kalba sintezei.

NUT sistema sukurta 90-ųjų metų pirmoje pusėje kaip projekto START rezultatas [171]. Lyginant su PRIZ sistema, NUT sistemoje pagrindinės sistemos naujovės yra šios [2, 175, 140, 170]:

- SSP metodo pritaikymas objektinei paradigmai;
- grafinė vartotojo sąsaja, leidžianti paprasčiau specifikuoti uždavinį;
- galimybė uždavinio specifikacijoje naudoti C kalbos konstrukcijas ir generuoti programas C kalba. [177]
- skirtingų skaičiavimo modelių integracija [89].
- platesnė praktinio pritaikymo sritis. PRIZ sistema buvo naudojama skaičiuojamųjų matematikos ir fizikos uždavinių sprendimui [170, 167]. Tuo tarpu NUT buvo panaudota vandens šildymo sistemų valdymo [2], automobilio su mechanine greičių dėže modeliavimo, optimalaus radarų išdėstymo Estijos teritorijoje uždaviniams spręsti. Be to šią sistemą galima naudoti ir kitų sričių, pvz. tekstų apdorojimo uždavinus spręsti [89].

Apie 1995 m., *M. Addibpour* ir *V. Vlasov* praplėtė NUT sistemą, sukurdami *rNUT* įrankį lygiagrečioms programoms sintezuoti [3]. NUT sistemoje orientuojamasi į objektinę paradigmą, todėl NUT kalba užrašyta specifikacija lyginant su UTOPIST kalba turi ypatumų:

- UTOPIST būdinga **let** sritis, kurioje aprašomi kintamieji ir jiems suteikiamos pradinės reikšmės, pakeista į dvi sritis **var** (kintamiesiems aprašyti) ir **init** (kintamųjų pradinėms reikšmėms nurodyti).
- Siekiant trumpiau užrašyti klasės elementų ir jų sudėtinių dalių (poelemenčių) pavadinimus naudojami pseudonimai. Jiems aprašyti skirta **alias** sritis.

- Klasė gali paveldėti kitos klasės savybes. Paveldinčios klasės apraše, **super** srityje nurodomas paveldimosios klasės vardas.

- Dinamiškai sukurti naujiems objektams naudojamas operatorius **new**.

Šiek tiek skiriasi ir pats sintezės procesas. NUT sistemoje vyksta dviejų tipų sintezė:

- klasės elementų (laukų) reikšmių sintezė (skaičiavimas) kaip metodo **compute** rezultatas;

- programos, realizuojančios objekto elementų sąryšį sintezė (ir jei reikia použdaviniu realizacijų sintezė), kai kreipiamasi į objekto elementų sąryšį.

Sintezės metu naudojamos objekto elementų reikšmės (būsenos) ir objekto ryšiai, jau turintys realizacijas. Sąryšis kurio aprašyme yra naudojami #**curr** ar #**next** vardai iš tiesų apima ryšių grupę. Konkretūs sąryšiai gaunami vietoj #**cur**r, #**nex**t įstatant #i ir #(**i**+**1**), kur i-teigiamas skaičius, ne didesnis nei eilės ilgis. NUT sintezės sistemoje naudojamos šios išvedimo taisyklės:

1. $$\frac{o \supset e}{o \vdash d}, \tag{3.14}$$

   kur $o.e$ reikšmė apibrėžta, o $x \vdash y$ *reiškia remiantis* x *galima apskaičiuoti* y.

2. Jei objektas o turi realizuotą sąryšį (*angl. preprogrammed relation*) ir yra žinomi visi to sąryšio argumentai (įskaitant ir použdavinius) bei visi jo silpnai susieti (angl. *weak*) parametrai (galima išimtis - nežinomas tik vienas silpnai susietas parametras), tai visi jo rezultatai (angl. *output parameters*) ir trūkstamas silpnai susietas parametras yra apskaičiuojami:
   $$\frac{o \supset ins \mapsto weaks, w \mapsto outs; o \vdash ins, weaks}{o \vdash w, outs}, \tag{3.15}$$

3. Jei du objekto elementai yra ekvivalentūs ir vienas iš jų suskaičiuojamas, tai ir kitas yra suskaičiuojamas

4. Jei objektas o turi pseudonimą a su elementais $e1, e2 \ldots e2$, tai a apskaičiuojamas iš o. Jei a apskaičiuojamas iš o, tai ir $e1, e2 \ldots e2$ apskaičiuojami iš o.

5. Jei objekto elementas e yra sudėtinis (pvz. masyvas, struktūra ir pan.) ir visos jo dalys apskaičiuojamos iš objekto o, tai ir pats elementas e apskaičiuojamas iš o.

6. Jei objekto saryšio *priklausomų použdavinių* rezultatų apskaičiuojamumas gali būti įrodytas taikant hipotezę, kad použdavinių argumentai yra apskaičiuojami objekte, tai použdaviniai yra apskaičiuojami tame objekte.

7. Jei objekto sąryšio *nepriklausomų použdavinių* rezultatų apskaičiuojamumas naujame specifikacijoje aprašytos klasės objekte gali būti įrodytas taikant hipotezę, kad použdavinių argumentai yra apskaičiuojami tame pačiame naujame objekte, tai použdaviniai yra apskaičiuojami tame objekte.

8. Jei objekto sąryšio, neturinčio realizacijos, rezultatų apskaičiuojamumas gali būti įrodytas taikant hipotezę, kad to sąryšio argumentai yra apskaičiuojami, tai sąryšis yra apskaičiuojamas.

Aprašant NUT [2], pagrindinis dėmesys skiriamas grafinei sintezės sistemos vartotojo sąsajai (angl. *visual graphic language*). Specializuojant NUT sistemą konkrečiai dalykinei sričiai, jos esybės aprašomos klasėmis, ir vėliau vizualiai sujungiamos į vientisą sistemą.

Apie 2000-uosius metus Kibernetikos Institute Estijoje pradėta kurti nauja sintezės sistema, naudojanti SSP metodą skirta programoms Java kalba generuoti. Java kalba pasirinkta dėl savo lankstumo ir nepriklausomumo nuo platformos [75]. Pirmuosiuose darbuose [103] aprašoma, kad sintezė vyksta klasių operacijų arba funkcijų lygmenyje. Skiriamos trys sudėtinių elementų (angl. *building blocks*) tipai:

- klasių operacijos, kurias išrenka ir "sujungia" pati sintezės sistema,
- *Java* klasės atitinkančios konkrečias esybes,
- metaklasės.

Logikos kalba (taip pat naudojama ILP) išreiškiamos tokios programinės įrangos savybės:

- duomenų srautai (angl. *dataflow*) tarp modulių;
- modulių kvietimo tvarka (angl. *control variables*);
- hierarchinės duomenų struktūros;
- jau realizuotų (angl. *pre-programmed*) valdymo struktūros (použdaviniai);
- alternatyvūs modulių rezultatai (daugiausia aprašant išimtis – angl. *Exceptions*;

- tiesioginis modulių jungimas (angl. *implicit linking*).

Taip pat numatytas ir sintezės sistemos plėtimas komponentinėms CORBA sistemoms kurti [75], tačiau realizacijos pavyzdžių nepateikiama.
Kiekviena Java klasė yra papildoma lauku

```
public static String[] SSPspec
```

kuriame ir užrašoma deklaratyvioji specifikacija aprašanti klasės laukų sąryšius. Specifikacijoje naudojami dviejų tipų sąryšiai: lygybės ir ekvivalentumo. ekvivalentumo sąryšis nurodo, kad vieną elementą galima pakeisti kitu, o lygtis naudojamos naujų formulių išvedimui. Jas apdoroja atskira sintezės sistemos dalis – lygčių redaktorius [75].

Java programų sintezės sistemą suadro 6 sąveikaujantys komponentai [75]:

- žinių bazė (ŽB),
- kompiliatorius, analizuojantis deklaratyviąją specifikaciją ir jos rezultatus patalpinantis ŽB,
- dekoratorius, kuris sukuria specialią duomenų struktūrą, skirtą greitam planavimui, suteikia bandomąsias reikšmes šios struktūros elementams ir ją perduodantis planavimo moduliui,
- planavimo modulis pagal specifikaciją generuojantis programos struktūrą (algoritmą),
- kodo generatorius pagal programos struktūrą generuojantis programinį kodą Java kalba.

Tačiau sintezės sistemos rezultatas - yra ne komponentinė, o objektinė programa.
Planavimo modulyje naudojamos šios įrodymo paieškos strategijos:

- *Prielaidomis grįsta paieška pirmyn.* Įrodymo paieška pradedama nuo tų elementų, kurių reikšmės jau yra žinomos (t.y. jie paminėti specifikacijoje, kaip sistemos įvestis (angl. *inputs*)) ir „judama pirmyn“, nagrinėjant tik besąlyginius sąryšius (angl. *unconditional relations*). Iš pradžių visi duoti elementai įtraukiami į žinomų aibę. Kiekviename žingsnyje sąryšis, kurio visų argumentų (angl. *input objects*) reikšmės žinomos, tačiau nežinomas bent vienas rezultatas, įtraukiamas į sintezuojamąjį algoritmą. Panaudojus tokį sąryšį, jo rezultatai (angl. *output objects*) tampa žinomi ir papildo žinomų elementų aibę.

- *Tikslu grįsta paieška atgal* naudojama tuomet, kai prielaidomis grįsta paieškos pirmyn nepakanka. Operuojama tik su sąryšiais turinčiais použdavinius, kurių elementų pradinės reikšmės yra žinomos. Planavimo modulis rekursyviai naudojamas kiekvienam sąryšio použdaviniui spręsti. Jei visi použdaviniai išspręsti, sąryšis traukiamas į sintezuojamąjį algoritmą. Po kiekvieno sąryšio su použdaviniais iškvietimo algoritme naudojamas tiesinis planavimas.

- *Minimizavimas.* Anksčiau naudojamos paieškos strategijos negarantuoja, kad bus surastas optimalus sprendinys, mažiausiai laiko sugaištantis algoritmas tikslui pasiekti. Sprendinyje gali būti naudojami ir tokių sąryšiai, kurie tikslui pasiekti yra nebūtini. Minimizavimo metu tokie sąryšiai pašalinami.

3.3.3.3. Paslauginės struktūrinės sintezės sistemos

Viena komponentų rūšis – pasaulinio tinklo paslaugos – PTP (*angl. web-servises, WS*) [183] yra ir paslauginės architektūros (angl. *service oriented architecture*) realizacija. Yra dvi struktūrinės sintezės metodo realizacijos paslauginės architektūros programoms kurti: Lammerman [102] ESSP metodas ir J.Rao [145] metodas.
Darbe [102] yra pristatoma išplėstoji struktūrinė sintezė (ESSP) ir jos taikymai pasaulinio tinklo paslaugoms iš komponentų generuoti. ESSP metode kaip logikos kalba naudojama intuicionistinis teiginių skaičiavimas. Logikos kalba yra praplėsta įvedant šiuos elementus [102]:

- kvantorius,
- disjunkciją,
- konstantą $\perp$.

Kvantoriai įvesti atsižvelgiant į pasikeitusį sprendžiamo uždavinio pobūdį. Anksčiau taikant struktūrinės sintezės metodą buvo remiamasi prielaida kad sintezė vyksta tam tikros struktūros ar klasės viduje ir visi tos struktūros elementai yra žinomi uždavinio specifikavimo metu. Sintezuojant PTP susiduriama su problema, kad konkretūs komponentai tampa žinomi tik sintezės metu, specifikavimo metu jie dar nežinomi.

Disjunkcija įvesta siekiant atvaizduoti išimtis (angl. *exceptions*), o konstantą $\perp$ – į jas reaguoti. Tiek disjunkcijos ženklas tiek FALSE gali būti tik dešinėje sąryšį aprašančios formulės pusėje. $\perp$ dešinėje pusėje vaizduoja programos darbo nutraukimą įvykus išimčiai. Disjunkcija dešinėje pusėje negali būti naudojama, nes, kaip įrodyta [166], tuomet perrinkimų skaičius didėtų eksponentiškai:

$$(((2^n)^n)^n \ldots)^n$$

Bendroji ESSP formulės struktūra yra tokia:

$$[\forall \bar{x}] \bigwedge_{i=1}^{n} \left([\forall \bar{u}] \bigwedge_{j=1}^{n_i} U_{i,j} \mapsto [\exists v]\, V_i \right) \wedge \bigwedge_{j=1}^{m} X_j \mapsto \bigvee_{i=1}^{k} \left(\bigwedge_{j=1}^{l_j} [\exists y]\, Y_{i,j} \right) \quad (3.16)$$

kur $n, m \geq 0$, $n, k, l_i \geq 1$, $U_{i,j}, V_i, X_j$ ir $Y_{i,j}$ yra teiginiai(angl. *propositional variables*) arba *monadiniai predikatai*, kurių kintamieji nėra laisvi. $\bar{x}$ ir $\bar{u}$ yra predikatų X_j ir $U_{i,j}$ kintamieji, o v, y predikatų V_i ir $Y_{i,j}$ kintamieji. Nauji teiginiai W ir Z bei aksiomos:

$$W \vdash \bigvee_{i=1}^{k} Z_i \quad (3.17)$$

$$Z_i \vdash_{j=1}^{l_j} Y_{i,j} \quad (3.18)$$

Į logikos kalbą įvedus naujus elementus išvedimo taisyklių sąrašas taip pat pasipildė naujomis taisyklėmis. Įvestos $\perp$ eliminavimo

$$\frac{\Gamma \vdash \perp}{\Gamma \vdash C},$$

kvantorių įvedimo ir eliminavimo

$$\frac{\Gamma \vdash [a/x]\, A}{\Gamma \vdash \forall x.A} \quad (3.19)$$

$$\frac{\Gamma \vdash \forall x.A}{\Gamma \vdash [t/x]\, A} \quad (3.20)$$

$$\frac{\Gamma \vdash [t/x]\, A}{\Gamma \vdash \exists x.A} \quad (3.21)$$

$$\frac{\Gamma \vdash \exists x.A \Gamma, u : [a/x]\, A \vdash C}{\Gamma \vdash C} \quad (3.22)$$

$$(3.23)$$

bei disjunkcijos eliminavimo

$$\frac{\vdash \bar{W} \mapsto A \vee B \Gamma \vdash \bar{W} \Sigma, u : A \vdash C \Delta, w : B \vdash C}{\bar{\Gamma}, \Sigma, \Delta \vdash C},$$

taisyklės. Ryšium su naujų elementų įvedimu ESSP naudojamas šiek tiek pakeistas SSP įrodymo paieškos algoritmas. Aksioma, kurioje yra disjunkcija perrašoma kitu pavidalu (kaip įprasta taikant SSP, normalizuojama) tada ir tik tada jei yra naudojama. Toliau įrodymo paieška tęsiama naudojant naująjį (ką tik viena aksioma papildytą) aksiomų rinkinį. Kiekvienai tokiu būdu perrašytai aksiomai jos realizacija generuojama automatiškai, priešingai nei klasikinio

SSP atveju, kur tokios aksiomos realizacija jau iš anksto turėjo būti pateikta. Visuotinumo kvantorius panaikinamas įstatant konkrečią reikšmę, o egzistavimo kvantorius panaikinamas panašiai kaip ir disjunkcija. Siekiant sumažinti įrodymo paieškos algoritmo sudėtingumą šie keitimai atidedami vėliausiam įmanomam laikui. Pirmiausia ieškant įrodymo bandoma taikyti besąlyginius skaičiuojamuosius sakinius ir tik tada, jei tikslas dar nepasiektas, eliminuojami kvantoriai. Kvantorius $\exists x.P(x)$ gali būti pakeistas į $P(a)$, jei galima įrodyti $\Gamma \vdash P(a) \mapsto G$, kur G - tolimiausias tikslas (angl. *innermost goal*), a - laisvas parametras, nesįeinantis š patį G ir G išvedimo hipotezės.
Tam kad išvesti G iš $\Gamma \cup P(a)$ turi būti aksioma

$$\forall y.P(y) \wedge L \mapsto A,$$

kur L yra bet kokia (taip pat ir tuščia) leistina logikos kalbos pagrindinės implikacijos kairės pusės subformulė, y neturi būti laisvas L ir A termuose ir A yra leistina dešinė pagrindinės implikacijos pusė. Prieš taikant visuotinumo kvantoriaus eliminavimo taisyklę, turi būti išvestas L ir žinomas $P(a)$. Eliminavus kvantorių gaunama:

$$P(t) \wedge L \mapsto A.$$

Darbuose [114, 145] ši idėja plėtojama. *J. Rao* įvedė šiuos struktūrinės sintezės patobulinimus:

- pasaulinio tinklo paslaugų specifikavimui naudojama DAML-S/OWL-S kalba, o teoremų įrodymui – *Tiesinė logika (Linear Logic)*. Iki tol nenaudota Tiesinė Logika parinkta atsižvelgiant į PTP sintezės uždavinio specifiką: daugybinės konjunkcijos (angl. *multiplicative conjunction*) sąryšis reikalingas išreikšti paslaugai reikalingų resursų skaičių; logikos modalumo savybės (pvz. predikatas „*Būtinai*“) leidžia lanksčiau modeliuoti paslaugos pasiekiamumą;

- pirmą kartą struktūrinės sintezės metode iš specifikacijos į loginę teoriję žinios perkeliamos ne tiesiogiai, bet naudojant dar vieną formalizmą - *Petri* tinklus;

- naudojami ir nefunkciniai PTP ribojimai;

- skirtingai nei Lammerman darbe [102], atsisakyta kvantorių.

3.4. *Curry-Howard* protokolas

Curry-Howard izomorfizmas [133, 84] tapo ištisos automatizuotų programų kūrimo metodų klasės, dar vadinamos *įrodomuoju programavimu* (angl. *proofs-as-programs*), pagrindu.
H. Curry ir W.A. Howard įrodė, kad bet kuriam termui p, kurio tipas yra A, egzistuoja vienareikšmė atitiktis tarp p konstrukcijos (sekos termų, nurodančių, kaip jis gautas) ir A išvedimo natūraliosios dedukcijos sistemoje [131]. *Curry-Howard* izomorfizą nusako ši teorema [142]:

3 teorema. Tarkime $\Gamma = G_1, \ldots G_n$ yra prielaidų aibė, o $\Gamma' = x_1^{G_1}, \ldots, x_n^{G_n}$ – tipizuoti įrodymo termo kintamieji (angl. *proof-term variables*). Tada,

1. Turint natūraliosios dedukcijos įrodymą $\Gamma \vdash A$, naudojantis tipų išvedimo taisyklėmis galima sukonstruoti taisyklingą įrodymo termą (angl. *well-typed proof-term*) p^A, kurio laisvieji įrodymo termo kintamieji priklauso aibei Γ'

2. Turint įrodymo termą p^A kurio įrodymo kintamieji yra Γ' galima sukonstruoti natūraliosios dedukcijos įrodymą $\Gamma \vdash A$

Yra žinoma keletas įrodomojo programavimo metodų pavyzdžių: *NUT*, *Amphion* ir kt. Iki 2005 m. šie metodai buvo kuriami ir tobulinami nepriklausomai, neprisilaikant kokios nors bendros metodikos [142]. *I. Poernomo* grupė įrodomojo programavimo metodų klasę apibendrino ir pasiūlė *Curry-Howard* protokolą, nusakantį tokių metodų kūrimo ir naudojimo taisykles. Siekdama universalumo *I. Poernomo* grupė tiek logikos, tiek programavimo kalbos pasirinkimui [142] nekelia jokių apribojimų (3.5 lentelė). Pastebima, kad gali būti įvestos ir papildomos rolės, tačiau tuo atveju būtų sunku metodą apibendrinti.

Nusakant *Curry-Howard* protokolą naudojamos tipų teorijos, logikos (arba naturaliosios dedukcijos sistemos), loginės tipų teorijos ir skaičiuojamosios tipų teorijos sąvokos.

Tipų teorija
Tipų teoriją

$$TT = \langle Terms(TT), Types(TT), \vdash_{TT}, (:), TIR \rangle \tag{3.24}$$

sudaro šie elementai:

- Termų ir tipų su rekursyviosiomis gramatikomis aibės.

- Egzistuoja kintamųjų aibė, termų aibės poaibis $Var_{Terms(TT)}$.
- $(:)$ – dvinaris termų ir tipų atitikties sąryšis.
- tipų išvedimo sąryšis $\vdash_{TT}$ tarp aibės $\Gamma = x_i : S_{ii=1,\ldots,n}(x_i \in Var_{Terms(TT)}, S \in Types(TT)$ (vadinamojo tipų konteksto) ir vienintelio sąryšio termui $t : S(t \in Terms, S \in Types)$. Sąryšis $\Gamma \vdash_{TT} t : S$ vadinamas tipo išvedimu.
- Sąryšis $\vdash$ apibrėžtas aibe tipo išvedimo taisyklių TIR, Aibę sudaro taisyklės pradedant keletu tipų (prielaidų) išvadų baigiant vienu (išvados) išvedimu:
$$\frac{\Gamma_1 \vdash_{TT} t_1 : S_1 \ldots \Gamma_n \vdash_{TT} t_n : S_n}{\Gamma \vdash_{TT} t : S}(R)$$
kur (R) - unikalus taisyklės vardas. SIR aibėje gali būti ir begalinė. Jei $t_1, t_2 \in Terms$, tai bet kokiam kontekstui Γ $\Gamma \vdash t_1 : S \Leftrightarrow, \Gamma \vdash t_2 : S$, tai sakoma, kad ir t_1, ir t_2 turi tą patį tipą S. Rašoma $t^{\Gamma} : S$ norint denotuoti termą t ir parodyti faktą, kad $\Gamma \vdash t : S$ gali būti išvestas ir t vadinamas *taisyklingas* (angl. *well typed*) Γ kontekste. Jei kontekstas aiškus, rašoma ne $t : S$, o $t^{\Gamma} : S$.
- $\vdash_{TT}$ turi būti apibrėžtas taip, kad kiekvienas termas būtų $t \in Terms(TT)$ taisykligai tipizuotas.

Logika

I. Poernomo grupė [142], remdamasi *D. Gabbay* deduktyviųjų sistemų notacija naturaliosios dedukcijos sistemą apibrėžia taip:

$$D = \langle Formulae(D), \vdash_D, DR \rangle \tag{3.25}$$

- $Formulae(D)$ – taisyklingų (angl. *well-formed*), rekursyviosios gramatikos sakinių aibė, samprotavimams deduktyviojoje sistemoje D atlikti.
- $\vdash_D$ – sąryšis tarp prielaidų formulių $\Gamma_i \in Formulae(D)$ ir vienos išvados formulės $C \in Formulae(D)$,
$$D = \Gamma \vdash_L C \tag{3.26}$$
Šis sąryšis vadinamas išvedimu (angl. *inference*). Išvedimas aprašytas aibe dedukcijos taisyklių DR, susidedančių iš taisyklių iš keleto (prielaidų) išvedimų į vieną (išvadinį) išvedimą tokia forma:
$$\frac{\Gamma_1 \vdash_D F_1 \ldots \Gamma_n \vdash_D F_n}{\Gamma \vdash_D F}(R) \tag{3.27}$$
DR taip pat gali būti ir begalinė aibė.

Loginė tipų teorija (LTT)

Loginė tipų teorija yra konkrečios dedukcijos sistemos vaizdavimas (angl. *presentation*) tipų teorijoje. Laikantis *Curry-Howardo* izomorfizmo LTT tipai yra logikos formulės, termai atitinka tų tipų išvedimo žingsnius, redukcijos sąryšis termams apibrėžia įrodymų normalizavimo strategiją.
Formaliai loginė tipų teorija dedukcinei sistemai L aprašoma taip:

$$LLT(L) = \left\langle PT(L), Formulae(L), (.)^{(.)}, \vdash_L, PTR, \triangleright \right\rangle \tag{3.28}$$

kur $PT(L)$ – įrodymų termai, $Formulae(L)$ – tipai, $(.)^{(.)}$ – tipizavimo (angl. *type judgment*) sąryšis, $\vdash_L$ – tipų išvedimo sąryšis apibrėžtas PTR taisyklėmis. Daromos tokios prielaidos:

- įrodymų termų aibė $PT(L)$ turi atskirą kintamųjų poaibį $Var_{PT(L)}$.
- $PT(L)$ apima lambda skaičiavimą.
- tipizavimo sąryšis $(p)^{(T)}$, apibrėžtas tarp įrodymo termų $p \in PT(L)$ ir formulių $T \in Formulae(L)$.
- PTR aibė gali būti begalinė
- egzistuoja normalizavimo sąryšis $\triangleright$ apibrėžtas virš įrodymų termų $PT(L)$, generuotas kaip tranzityvus vieno žingsnio redukcijos sąryšio $\triangleright$ uždarinys (angl. *closure*). Sąryšis $\triangleright$ apibrėžtas taisyklių aibe virš įrodymų termų $p_1 \triangleright p_2$. Reikalaujama [142], kad:
  - normalizavimas išlaikytų įrodymo termų tipus, taip kad $\Gamma \vdash_{PT(L)} p_1^A$ ir $p_1 \triangleright p_2$ duos $\Gamma \vdash_{PT(L)} p_2^A$;
  - normalizavimo sąryšis turi būti griežtai normalizuojantis: kiekvienam įrodymo termui p_i, kiekviena vienžingsnių redukcijų seka yra baigtinė ir baigiasi termu p_n;
  - sąryšis turi tenkinti *Church-Rosser* savybę: bet kokiems įrodymo termams $p, p_1, p2$, kuriems $p \triangleright p_1, p \triangleright p_2$, turi egzistuoti ir bendras termas p_3, toks kad $p_1 \triangleright p_3, p_2 \triangleright p_3$.

Skaičiuojamoji tipų teorija (CTT)

CTT apibrėžta taip, kad tiktų kiek galima daugiau programavimo kalbų:

$$CTT = \langle Terms(C), Types(C), :, \vdash_C, TIR, \triangleright \rangle\,, \tag{3.29}$$

kur $Term(C)$ - termų aibė, apibrėžianti programas, $Type(C)$ – tipų aibė abibrėžianti programų tipus, $t : T$ – tipizavimo sąryšis, išlaikomas atitinkamai tarp programų ir jų tipų pagal tipų išvedimo sąryšį $\vdash_C$ ir taisykles TIR.

I. Poernomo [142] pastebi, kad pagal šį apibrėžimą imperatyviosios programavimo kalbos taip pat gali būti laikomos skaičiuojamosiomis tipų teorijomis.

Curry-Howard protokolas

Curry-Howard protokolas yra išlaikomas tarp loginės tipų teorijos L ir skaičiuojamuosios tipų teorijos C tada kai:

1. Egzistuoja atitikties schemos (angl. *extraction maps*) iš L formulių į C tipus (*etype* schema) ir iš L įrodymo termų į C programas *extract* **schema**:

$$extract: \ PT(L) \mapsto Terms(C) \tag{3.30}$$

$$etype: \ Formulae(L) \mapsto Types(C) \tag{3.31}$$

   toks, kad bet kuriam įrodymui $d \in PT(L), \Gamma \dashv d^A \ extract(d)$ bus C dalis ir priklausys tipui $etype(A)$.

2. Tarp programų ir formulių egzistuoja realizuojamumo sąryšis r, toks kad bet kuriam įrodymui

$$\emptyset \vdash_L p^A \in PT$$

   teisingas ir $extract(p) r A$.

Curry-Howard protokolo naudojimo procesą sudaro keturi etapai [142]:

1. apibrėžiamas loginis skaičiavimas,
2. loginiam skaičiavimui apibrėžiama loginė tipų teorija,
3. parenkama realizavimo (programavimo) kalba ir aprašoma kaip skaičiuojamoji tipų teorija,
4. įrodoma, kad *Curry-Howard* protokolas išlaikomas pasirinktose srityse. (angl. *protocol to hold over the domains*).

3.5. Induktyviosios sintezės metodas

Induktyvioji sintezė plačiai taikoma loginių programų kūrimui [14, 54, 71, 91, 164]. Egzistuoja net atskira loginio programavimo šaka – induktyvinis loginis programavimas [125].

Taikant induktyviųjį metodą turimos žinios skiriamos į dvi grupes: bazinę teoriją (*angl. background theory*) B ir apibendrintinus duomenis E (be to, $B \not\models E$). Apibendrintini duomenys dar skirstomi į teigiamus E^+ ir neigiamus E^- pavyzdžius. $B \wedge E^- \not\models \perp$. Iškelta hipotezė H laikoma induktyviąja išvada jeigu ji neseka iš turimų žinių, tačiau paaiškina teigiamus pavyzdžius $B \wedge H \models E^+$ neprieštaraudama neigiamiems $B \wedge H \wedge E \not\models \perp$. Egzistuoja bendrieji algoritmai hipotezėms generuoti ir joms tikrinti [125].

Pagrindinės induktyviojo loginio programavimo naudojimo sritys yra šios: vaistų gamyba, palydovinė orų prognozė, automatinis programavimas. Induktyvusis loginis programavimas naudojamas funkcinių programų kūrimui [14, 125], tačiau pastaruoju metu daugėja jo taikymų struktūrinei, objektinei ir aspektinei paradigmoms [21, 54, 164].

Toliau šiame poskyryje nagrinėjami konkretūs induktyviojo metodo taikymo programinei įrangai kūrti atvejai.

Darbe [91] apie programų sintezę (konkrečiai programų transformavimą) kalbama kaip apie priemonę teoremoms įrodyti induktyviuoju metodu. Išskiriamos *metaskaičiavimo* (angl. *metacomputation*), *mišriųjų skaičiavimų* ir *superkompiliavimo* technikos.

Kaip teigiama [91], induktyviesiems spėjimams būdinga bendra struktūra, t.y. tam, kad įrodyti indukcinę išvadą (angl. *conjecture*) reikia parinkti *indukcijos schemą* ir *hipotezę*. Įrodymai gali reikalauti tarpinių lemų arba turi būti generuota ir įrodyta bendresnė formulė. Įrodyme gali būti panaudojama ir informacija gauta ankstesniame nesėkmingo įrodymo žingsnyje. Indukcinio kintamojo parinkimas, termo ir indukcinės hipotezės generavimas yra realizuoti kaip euristika apimanti indukcijos taisyklę ir įrodymo sistemą, taip užtikrinant didesnę sėkmingo įrodymo radimo tikimybę. Viena tokių euristikų yra *rekursinė analizė* (angl. *recursion analysis*).

Indukcinės teoremų įrodymo programos induktyviųjų teoremų įrodymui naudoja indukcijos taisykles. *R. S. Boyer* ir *J. S. Moore* [22] remiasi rekursijos ir indukcijos sąsajomis ir naudoja teoremų įrodymo programą realizuojančią *struktūrinės indukcijos* metodą. Jų sukurta teoremų įrodymo programa BMTP naudoja išreikštines (angl. *explicit*) indukcines taisykles. Rekursinės analizės proceso metu konstruojama atitinkama indukcinė taisyklė ir pasiūlomi indukciniai kintamieji indukcinei išvadai (angl. *conjecture*) [91]. Jei kintamasis yra funkcijos rekursyviojo argumento pozicijoje, sakoma, kad tai yra atvejis be trūkumų (angl. *unflawed*). Jei kintamasis yra nerekursyviojo argumento pozicijoje, sakoma, kad atvejis yra su trūkumais (angl. *flawed*). Indukcijos kin-

tamaisiais gali būti parinkti tik universaliųjų kvantorių kintamieji neturintys atvejų su trūkumais. Atvejų su trukumais problema yra ta, kad jei tokie atvejai yra pakeisti indukcijos terme, jie negali būti perrašyti indukcijos išvadoje.

Tikrosios indukcijos spėjimuose kintamajam x yra siūlomos dvi schemos: dviejų ir vieno žingsnio. Rekursinė analizė siūlo dviejų žingsnių indukcijos taisyklę ir indukcinį kintamąjį x. Remiantis siekiama galutine indukcine išvada ir pateikta preliminaria indukcine išvada yra formuojami pagrindiniai ir žingsnio atvejai. Naujai formuluotiems tikslams perrašyti pagrindinė ir rekursyvioji funkcijų formulės naudojamos kaip perrašymo taisyklės.
Jei visi indukciniai kintamieji yra su trukumais, rekursinė analizė siūlo visus juos imti kaip indukcinius kintamuosius. Pastebėtina, kad rekursinės analizės metodas netaikytinas jei formulėse yra egzistavimo kvantorių.

M.H. Kabir [91] nagrinėja induktyviojo metodo taikymus metalygmenyje. Metaskaičiavimo procese ir pačios programos traktuojamos ne kaip duomenų subjektas, bet kaip duomenys. Programos, gebančios manipuliuoti programomis yra vadinamos metaprogramomis ($< Mprog >$) [91]. Atliekant metaskaičiavimą teoremų įrodymas gali būti visiškai automatizuotas naudojant sulenkimo (angl. *unfold-fold*) programų transformacijas. Teoremų naudojimui naudojama superkompiliatoriaus savybė atlikti gilų funkcijų aprašymų transformavimą. Siekiant įrodyti, kad savybė P yra teisinga teisinga visiems x ($\forall x.P(x)$), naudojant superkompiliatorių galima transformuoti originalų $P(x)$ apibrėžimą į $True$.
R. Backhouse [11] taip pat aprašo indukcijos ir invariantų naudojimą, tačiau daugiau akcentuoja formulių tikrinimo procesą naudojant naudojant matematinę indukciją.
Idėją panaudoti loginį programavimą (taip pat ir induktyvųjį loginį programavimą) komponentinėms programų sistemoms kurti taip pat plėtoja mokslininkų *K.-K. Lau* [97, 98] ir *M. Martelli* [112] vadovaujamos tyrimų grupės.
Nepaisant plačių induktyviojo metodo galimybių, būtina atkreipti dėmesį, kad jis turi ir trukumų:

- Kaip pabrėžia *S. Muggleton* [124], metodas gali būti neefektyvus tuo atveju, kai remiamasi vien tik pavyzdžiais. Pavyzdžių aibei sudaryti gali būti sunaudojama daugiau resursų (pvz.: laiko) daug daugiau nei programai kurti.

- Nėra garantijų, kad indukcijos rezultatas visada bus korektiškas [14, 125]. Struktūrinė programų sintezė ir kiti deduktyviniai metodai užtikrina, kad gauta išvada yra teisinga. Programų sintezės sistemos, naudojančios deduktyviuosius metodus rezultatas taip pat yra visada korektiška. Tuo tarpu taikant induktyvųjį metodą remiantis tais pačiais pavyzdžiais gali būti gautos skirtingos išvados.

Induktyviosios sintezės metodai leidžia kurti sistemų elementus pasinaudojant naujomis, ką tik išvestomis žiniomis, tačiau, siekiant išvengti nekorektiško rezultato, visos induktyviosios išvados privalo būti papildomai tikrinamos.

3.6. Transformacinės sintezės metodas

Transformacinė sintezė bendruoju atveju apibrėžiama, kaip manipuliavimas programos atvaizdu, pakeičiant tos programos formą ar sintaksę [160]. Vadovaujantis šiuo apibrėžimu, galima teigti, kad transformacinė sintezė apima ir kitus generavimo metodus, įskaitant ir anksčiau aptartus struktūrinės sintezės bei induktyvųjį. Vis dėl to šioje disertacijoje transformacinę sintezę nagrinėsime siauresniu, darbuose [145, 55] išreikštu požiūriu, t. y. kaip atskirą metodą.

Šiuolaikinė transformacinė sintezė neatsiejama nuo *modelinės architektūros* (angl. *model-driven architecture - MDA*) metodo. Šiame poskyryje analizuojama transformacinės sintezės ir MDA taikymo komponentinėms programų sistemoms kurti automatizuotu būdu galimybės.

Viena dažniausiai nagrinėjamų automatizuotos programų sintezės problemų yra tikslios specifikacijos tikslumo problema. Egzistuoja didelis atotrūkis tarp dalykinės srities, kuriai kuriama programinė įranga, ir realizacinės srities (konkrečių komponento modelių, karkasų, operacinių sistemų ir t. t.) konceptų. Šis atotrūkis sunkina komponentinių sistemų specifikavimą ir realizavimą (konkrečiai - komponentų paiešką ir adaptavimą). Sąlyginai nauja modelinė architektūra sukurta siekiant tokį atotrūkį paversti pranašumu. Plačiai aprašomi tiek bendrieji modelinės architektūros principai [94, 118, 138], tiek jos taikymo konkrečiai programų kūrimo paradigmai atvejai [5, 106, 189]. Tačiau [**?**, 95] siūlomi MDA taikymai komponentinių programų sistemoms kurti nevisiškai atitinka komponentinę paradigmą. Pažeidžiamas vienas komponentinės paradigmos principų - komponentų ir komponentinių sistemų kūrimo procesų atskyrimas [92, 37, 159], nes naudojant [5, 95] metodus sistemos kuriamos iš čia pat generuotų komponentų. Modelinė architektūra – tai OMG konsorciumo palaikoma programinės įrangos kūrimo iniciatyva, pabrėžianti modelių svarbą programinės įrangos kūrimo procese. Programų sistemų, kuriamų naudojant MDA, gyvavimo ciklą, kaip ir klasikinį gyvavimo ciklą sudaro: reikalavimų rinkimas, analizė, projektavimas, kodavimas, testavimas bei tiražavimas ir palaikymas [94]. Tačiau skiriasi kiekvieno iš šių etapų automatizavimo laipsnis ir rezultatai.

Analizės etapo rezultatas - abstraktusis modelis (angl. *platform-independent model - PIM*). Kuriant programų sistemos PIM modelį operuojama tik dalykinės srities sąvokomis, nesigilinant į realizacines detales, tokias kaip programavimo kalba, DBVS ir pan.

Projektavimo etapo rezultatas - konkretusis modelis (angl. *platform-specific model - PSM*). PSM modelyje operuojama jau realizacinėmis sąvokomis, detalizuojami nuo konkrečios operacinės sistemos, karkaso, programavimo kalbos ir kitų veiksnių priklausomi sprendimai. Darbe [118] pabrėžiama, kad šiuolaikinė programinė įranga yra itin sudėtinga, todėl MDA procesas gali apimti ne vieną, o keletą PIM ir keletą PSM modelių.

Dar vienas MDA pranašumas - galimybė automatizuoti programų sistemų kūrimo procesą. Visiško arba dalinio automatizavimo objektai yra šie [59, 138]:

- abstrakčiojo modelio neprieštaringumo tikrinimas;
- abstrakčiojo modelio transformacija į konkretųjį modelį;
- konkrečiojo modelio transformacija į programinį kodą;
- vieno konkrečiojo modelio transformacija į kitą (pvz., kitai OS skirtą) konkretųjį modelį.

Automatizuotas pakartotinis PSM modelio ir atitinkamai programinio kodo generavimas pasikeitus PIM modeliui sudaro galimybes operatyviai reaguoti į reikalavimų pokyčius.

Abstraktusis modelis gali būti aprašomas tik dalykinės srities sąvokomis, tačiau siekiant, kad modelis atitiktų komponentinę paradigmą, tikslinga ir jame naudoti komponento konceptus. Akivaizdu, kad čia būtų operuojama dalykinės srities (verslo) komponentais, o ne programiniais komponentais [109]. *Z. Stojanovič* [154] skiria du abstrakčiuosius modelius: verslo komponentų modelį ir taikomosios programos modelį (angl. *application component model*). *S. Gobel* [67] siūlo naudoti adaptyviuosius komponentus, kurie gali būti konfigūruojami ir pritaikomi tiek jų kūrimo, tiek paskirstymo, tiek vykdymo metu. Siekiant užtikrinti kuo didesnį modelio tikslumą siūloma naudoti UML 2.0 kalbos poaibį - vykdomąją UML kalbą [117]. Pagrindinis šios kalbos pranašumas - galimybė itin išsamiai aprašyti esybių dinamiką, ir tai sudaro sąlygas gautus vykdomuosius modelius įvykdyti ir patikrinti jau šiame etape. Be tradicinėmis tapusių UML kalbos priemonių, vykdomojoje UML įvesta klasių veiksmų (angl. *Class Actions*) semantika [117]. Klasių (šiuo atveju - esybių) veiksmai aprašomi specialia klasių veiksmų kalba (KVK) [136]. Klasių veiksmų naudojimas padeda aprašyti dalykinės srities logiką, o vėliau - vykdymo logiką, abstrahuojantis nuo konkrečių platformų ir kitų detalių. Pažymėtina, kad kaip veiksmų aprašymo kalbą galima naudoti ne tik KVK, bet ir kitas veiksmų aprašymo kalbas, pavyzdžiui, *Schlaer-Mellor* [117] kalbą. Ribojimus, kuriuos privalo tenkinti dalykinės srities komponentai, tikslinga aprašyti OCL kalba. Vykdomoji UML kalba sukurta profiliavimo būdu praplečiant UML 2.0 kalbos bazę [118] ir gali būti lengvai modernizuojama sukuriant kitą profilį, labiau pritaikytą komponentinei paradigmai. Be to, naudojant MOF (angl. *Meta Object Facility*)

metamodelių kūrimo konstrukcijų aibę [137] galima kurti konkrečių dalykinių sričių modeliavimo priemones.
Konkretusis modelis, aprašytas vykdomąja UML kalba, yra suvokiamas kaip modelio kompiliavimo proceso parametras [117]. Šiame modelyje jau atsispindi komponento modelio, karkaso ir kitos realizacinės detalės. Realizuojant programų sistemą gali tekti atsižvelgti į įvairius jos aspektus, todėl tikslinga naudoti keletą konkrečiųjų (vienas iš kito gaunamų) modelių, kurių skaičius priklauso nuo programų sistemos sudėtingumo [94, 118]. Pavyzdžiui, *S. Gobel* [67] išskiria tris konkrečiuosius modelius: kūrimo, paskirstymo ir vykdymo. Po atskirą konkretųjį modelį turėtų būti kuriama ir kiekvienam kitam komponento modeliui. Pavyzdžiui, nusprendus naudoti nebe CORBA, o .NET komponentus, būtina generuoti naują konkretųjį modelį. Konkrečiajam modeliui aprašyti tikslingiausia taip pat naudoti vykdomąją UML kalbą [94], nors galimi ir kitų kalbų [52, 138] naudojimo atvejai. Pažymėtina, kad automatizuotu būdu iš abstrakčiojo modelio transformacijos būdu gautas konkretusis modelis gali būti neišsamus, neatitikti realizacinės aplinkos. Pavyzdžiui, jame aprašyti kol kas neegzistuojantys komponentai, arba kelių realių komponentų funkcijos aprašytos kaip vieno. Modeliui patikslinti siūloma naudoti detalizavimo (angl. *model elaboration*) priemones.

S. Gobel [67] nagrinėja adaptyviuosius komponentus, be kitų interfeisų turinčius ir valdymo bei parametrizavimo interfeisus, kurie naudojami komponento konfigūravimui ir adaptavimui. Manoma, kad kiekvienas abstrakčiajame modelyje naudojamas dalykinės srities komponentas turi konkrečius realizacinius atitikmenis. Tokiu atveju modelių transformacija yra gana paprasta. Tačiau iš tikrųjų dalykinės srities ir realizacinės dalies komponentų prasmė, skaičius ir kitos savybės gali įvairuoti, vienareikšmė atitiktis įmanoma tik idealiu atveju, todėl darbe [67] išdėstytas požiūris yra netikslus. Abstrakčiojo modelio transformacijos į konkretųjį modelį etape būtina atsižvelgti į *Išrink-Pritaikyk-Testuok* gyvavimo ciklo specifiką. Tradiciniu būdu taikant MDA šiame etape generuojamas konkretusis modelis, kuris parodo, kokia sistema turėtų būti sukurta. Šiuolaikiniuose komponento modeliuose komponentas suprantamas kaip „juodoji dėžė“ su minimaliomis konfigūravimo priemonėmis, todėl konkretusis modelis turi parodyti, kokie esami komponentai ir kaip turi būti panaudoti. Iš to išeina, kad transformacijos metu privalo būti atliekami dar ir papildomi veiksmai:

- esamų realizacinių komponentų paieška ir atranka saugykloje;
- atskirų realizacinių komponentų konfigūracijų kitimo ribų tyrimas ir komponentų konfigūravimas (jei įmanomas);
- trūkstamų realizacinių komponentų identifikavimas.

Transformacijos uždavinys patenka į grupę uždavinių, kurie jau yra sprendžiami kuriamoje komponentinių programų sistemų sintezės sistemoje [66]. Tai įgalina naudoti MDA kuriant komponentines programų sistemas, nes pakanka sukurti specialius abstrakčiojo modelio transformacijos įrankius naudojančius struktūrinės sintezės ir induktyvųjį generavimo metodus.
komponentinės programos konkrečiojo modelio transformacija į programinį kodą ypatinga tuo, kad realizaciniai komponentai yra juodosios dėžės:

- Generuojamo kodo apimtis yra daug mažesnė nei tradicinėje MDA. Generuojamas tik jungiantysis kodas ir trūkstami realizaciniai komponentai identifikuoti ankstesnės transformacijos metu (jei tokių buvo).
- Būtina patikrinti, ar realizaciniai komponentai („juodosios dėžės“) ir jų veiksmų semantika atitinka dalykinės srities komponentams keliamus reikalavimus (įskaitant ir veiksmų semantiką). Šiam palyginimui atlikti reikalingas specialus įrankis. Pavyzdžiui, .NET komponentų atveju toks įrankis turėtų formuoti atitiktis tarp refleksijos būdu apklausto komponento IL kodo ir vykdomosios UML veiksmų semantikos.

Pažymėtina, kad tiek abstrakčiojo modelio transformacija, tiek konkrečiojo modelio transformacijos gali būti iteratyviai kartojamos.

3.7. Skyriaus išvados

1. Programų generavimo metodai nėra „gryni“ – juos sudaro kelių metodų visuma, apimanti ir nekomponentinėms sistemos kurti skirtus metodus.

   1.1. Klasikiniai formalieji metodai[3] tiesiogiai nėra taikomi komponentinėms programų sistemoms kurti, tačiau yra kitų, išvestinių, programų sintezės metodų pagrindas.

   1.2. *M. Charpentier* kompozicinių sistemų savybių prognozavimo metodas gali būti taikomas numatyti ar komponentinės programų sistemos atitiks nefunkcinius reikalavimus, jei šiuos reikalavimus atitinka komponentai.

2. *Curry-Howard* protokolas apibendrinantis įrodomojo programavimo metodų klasę yra realizuotas imperatyvinei, funkcinei ir struktūrinei programų kūrimo paradigmoms.

3. Konkrečioms komponentinių programų sistemų surinkimo proceso automatizavimo problemoms spręsti gali būti naudojami struktūrinės sintezės, induktyviosios sintezės ir transformacinės sintezės metodai:

   3.1. Deduktyvieji metodai palaiko „juodosios dežės“ abstrakcijas, be to, taikant deduktyvųjį struktūrinės sintezės metodą užtikrinama rezultatų (taikomųjų programų) atitiktis specifikacijai

   3.2. Induktyvusis metodas gali padėti spręsti šias deduktyviuoju metodu neišsprendžiamas problemas: specifikacijos neišsamumo problemą, neapibrėžtųjų komponentų problemą ir nefunkcinių reikalavimų problemą.

   3.3. Transformacinė sintezė įgalina mažinti atotrūkį tarp dalykinės srities, kuriai kuriama programinė įranga, ir realizacinės srities (konkrečių komponento modelių, karkasų, operacinių sistemų ir t. t.) konceptų.

4. Deduktyviojo metodo realizacijos grindžiamos automatinių teoremų įrodymo įrankių naudojimu. Šių įrankių analizė parodė, kad jie netenkina šių reikalavimų: vidinio formalizmo bei uždavinio specifikavimo kalbos raiškos gebos; teoremos įrodymo proceso visiško automatizavimo; ryšio su kitomis programomis.

[3] *E. Dijkstra, C.A.R. Hoare, M. Charpentier, R. Backhouse ir J. Schumann*

3.3 lentelė Predikatų transformatorių palyginimas [33]

Pavadinimas	Žymėjimas	Prasmė
Silpniausias egzistencinis transformatorius	$WE.X$	Kokia komponento savybė turėtų būti įrodyta, kad užtikrinti jog bet kuri sistema naudojanti tą komponentą turės savybę X.
Silpniausias universalusis transformatorius	$SE.X$	1. Kas gali būti gauta (angl. *deduced*) iš sistemos, kuri turi bent jau vieną komponentą pasižymintį savybe X 2. Sistemų turinčių komponentą pasižymintį savybe X charakteristika
Stipriausias universalusis transformatorius	$SU.X$	1. Kas gali būti gauta iš sistemos, kuri sukurta vien tik iš komponentų pasižyminčių savybe X 2. Sistemų turinčių tik komponentus pasižyminčius savybe X charakteristika
WE konjugatas	$WE^*.X$	1. Kas gali būti gauta visiems komponentams esantiems sistemoje pasižyminčia savybe X 2. komponentų esančių sistemoje, kuri pasižymi savybe X, charakteristika
SE konjugatas	$SE^*.X$	Kas turi būti įrodyta sistemoje, kad užtikrinti jog visi jos komponentai tenkina X
SU konjugatas	$SU^*.X$	Kas turi būti įrodyta sistemoje, kad užtikrinti jog bent vienas jos komponentas tenkina X

3.4 lentelė Automatinių teoremų įrodymo įrankių lyginamoji analizė.

	HERBY ir THEO	Coq	Isabelle	SNARK
Uždavinio specifikavimo kalba/įrankis	specifinė/ *COMPILE*	*Galina/ Venecular*	*Isar/ Proof-General*	*Lisp*/-
Vidinis formalizmas	Pirmos eilės predikatų logika	*CoC, ICoC*	*HOL, ZF, LCF, Lambda Cube ir kt.*	Pirmos eilės predikatų logika
Įrodymo paieškos būdas	*Herby* naudoja semantinį medį, THEO - rezoliucijų metodą	priklauso nuo naudojamo formalizmo ir automatizavimo taktikos	priklauso nuo naudojamo formalizmo ir automatizavimo taktikos	rezoliucijų metodas
Ar naudojamos *taktikos*?	Ne	Taip	Taip	Ne
Ar naudojama uždavinių teorijų biblioteka?	Taip, numatytas ryšys su išorine TPTP biblioteka	Taip, daugiau kaip 15 skirtingų teorijų	Taip: skaičių teorijai, analizei, algebrai ir aibių teorijai	Ne
Įrodymo paieškos automatizavimo laipsnis	visiškai automatinė paieška	interaktyvi paieška	interaktyvi paieška	visiškai automatinė paieška
Ryšys su kitomis sistemomis (API)	nėra	nedokumentuotas API	API	tik iš *Lisp* programų
Ar pirminis programos tekstas yra pasiekiamas?	Taip	Taip	Taip	Taip
Palaikoma OS	UNIX	MacOS ir Windows	Linux, Mac OS ir Windows	UNIX
Gaunamų programų tipas	-	funkcinės *Caml* programos	*SML, OCaml, Haskell*	*Lisp*

3.5 lentelė *Curry-Howard* protokolo elementai [142].

Rolė	*Curry-Howard* protokolo elementas	Elemento savybės	Paskirtis
Logika	Natūraliosios dedukcijos sistema	Formalioji sistema apibrėžianti loginį skaičiavimą	Uždaviniui ir teiginiams (angl. *assertions*) apie jį aprašyti.
Įrodymai	Loginė tipų teorija LTT	Tipų teorija įgalinanti operuoti įrodymais loginiuose skaičiavimuose, pagal *Curry-Howard* izomorfizmą: tipai apibrėžia sakinius, termai – įrodymus, o tipų išvedimas atitinka loginį išvedimą.	Parodo teiginių teisingumą arba klaidingumą.
Realizacinė (programavimo) kalba	Skaičiavimo tipų teorija CTT	Tipų teorija programavimo kalbos operacinei semantikai išreikšti	Skaičiuojamosioms programoms kurti.

4 skyrius

Komponentinių programų kūrimo proceso automatizavimo metodas

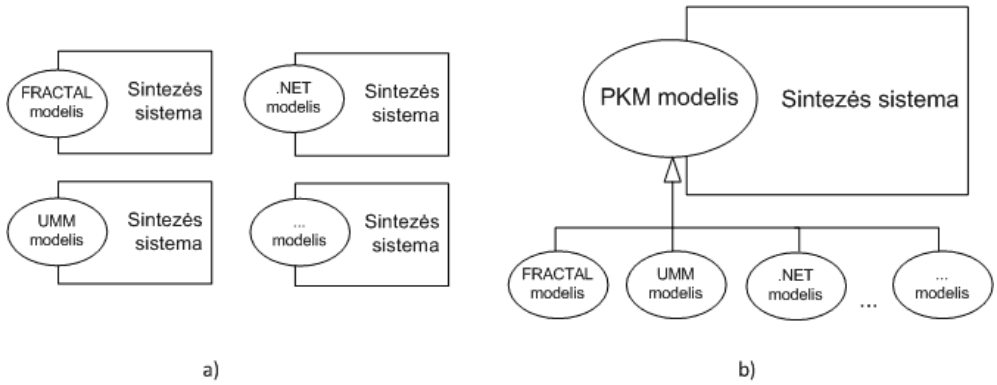
Šio skyriaus tikslas – aprašyti automatizuoto komponentinių programų kūrimo metodą, sukurtą remiantis ankstesniuose skyriuose analizuotais faktais. Pirmiausia 4.1. poskyryje formalizuojamas *Programinio komponento Modelis*, po to (4.2. poskyryje) pateikta *Curry-Howard* protokolo realizacija komponentinei paradigmai. Poskyriuose 4.3. ir 4.4. atitinkamai atskleidžiamas struktūrinės sintezės ir induktyviosios sintezės metodų pritaikomumas komponentinėms sistemoms kurti automatizuotu būdu.

4.1. Programinio komponento modelis

Skirtinguose komponento modeliuose komponento sąvoką apibrėžiama skirtingai [34, 122, 123, 134, 157, 159]. Egzistuoja kelios dešimtys formaliųjų [56, 61, 51, 104, 28, 104] ir su konkrečiomis komponentinėmis technologijomis susietų [122, 135, 134, 157, 183] komponento modelių.

Dažniausiai kiekvienam komponento modeliui kuriama atskira sistema (4.1 pav., a). Evoliucionuojant komponentinei paradigmai, atsiradus naujų komponento modelių iškyla būtinybė vėl kurti naujus sintezės metodus, naują sintezės sistemą.

Kita galimybė, kurią siūlo ir disertacijos autorius – abstrahuojantis nuo konkrečių komponento modelių savybių operuoti tik su abstrakčiu programinio komponento modeliu ir siūlyti automatizavimo sprendimus jam (4.1 pav., b). Tokio abstraktaus komponento modelio naudojimas praplečia sintezės metodo naudojimo ribas, metodas tampa mažiau priklausomas nuo komponentinės paradigmos evoliucijos ir ilgiau išlaiko aktualumą.



4.1 pav. Galimi komponento modelių ir sintezės sistemų sąveikos modeliai.

Šiame skyriuje aprašomas disertacijos autoriaus siūlomas *Programinio komponento modelis* (PKM) gautas remiantis mokslinėje literatūroje nagrinėjamų ir praktikoje naudojamų komponento modelių klasių analize.

Siekiant apibendrinti komponento modelius jų esybės iš pradžių buvo suskirstytos į keturias grupes (2.1.19. pav.):

- struktūrinės esybės,
- aprašomosios esybės,
- jungiančios (aprašančios sudėtinių komponentų ir jų sistemų sudarymo taisykles),
- procesinės (aprašančios komponentinių sistemų vykdymą ir jo palaikymą vykdymo aplinkoje, valdant tranzakcijas ir kt.).

Siekiant nustatyti, kurie komponento modelių elementai gali būti išskiriami bendrajame *programinio komponento modelyje*, taikyti du metodai: *klasterizavimo* ir *ekspertinio vertinimo*.

Klasterizavimo tikslas – sudaryti komponento modelių grupes, stebėti, pagal kuriuos požymius modeliai galėtų būti grupuojami. Klasterizuota komponento modelių savybių lentelę (2.1.19. pav.) ir komponento abstrakcijos lygmenų (2.3) lentelę interpretuojant kaip kategorinių kintamųjų aibę. Atstumams matuoti parinktas *Euklido atstumo kvadrato* matas. Klasterizavimas atliktas statistinės analizės programa *SPSS 16.0*.

Klasterizavimo rezultatai pateikti B priede. Gauti komponento modelių klasteriai aprašyti B.1. poskyryje, o komponento modelių savybių klasteriai – B.2. poskyryje.

Klasterizavimo metodas yra kiekybinis ir negarantuoja, jog yra apibendrinamos tapačios sąvokos. Siekiant tikslesnių rezultatų taikytas ir *ekspertinio*

vertinimo metodas, kurio rezultatai aprašomi 4.1.1.-4.1.2. poskyriuose. Programinio komponento modelis nagrinėjamas dviem aspektais: struktūriniu (4.1.1. poskyris) ir dinaminiu (4.1.2. poskyris).

4.1.1. Struktūrinis programinio komponento modelio aspektas

Komponentinę programų sistemą sudaro procesų sąrašas PRC ir vykdymo aplinkų sąrašas VA, procesų (jei jų daugiau kaip vienas) komunikavimo protokolas $InterPRC$, ir vykdymo aplinkų komunikavimo protokolas $InterVA$ (jei sistema heterogeninė):

$$KPS = \langle PRC, KK, InterPRC, InterVA \rangle , \tag{4.1}$$

kur $InterPRC$ – procesų komunikavimo protokolas, $InterVA$ – vykdymo aplinkųkomunikavimo protokolas, PRC – procesų sąrašas:

$$PRC = pr_1, pr_2, \ldots, pr_p \tag{4.2}$$

o VA – komponentinės sistemos veikimui reikalingų vykdymo aplinkų

$$VA = va_1, \ldots \tag{4.3}$$

Vykdymo aplinką komponentinėse technologijose (pvz. .NET) suteikia *komponentinis karkasas* (angl. *framework*), o kartais (pvz. EJB atveju) – ir komponentus aptarnaujantys konteineriai.
Jei komponentinė programa yra monolitinė, PRC sąrašą sudaro vienintelis procesas. Jei programų sistema išskirstytoji, išvardijami visi jos procesai $pr_1, pr_2, \ldots, pr_p$.

Bet kuris procesas aprašomas pora:

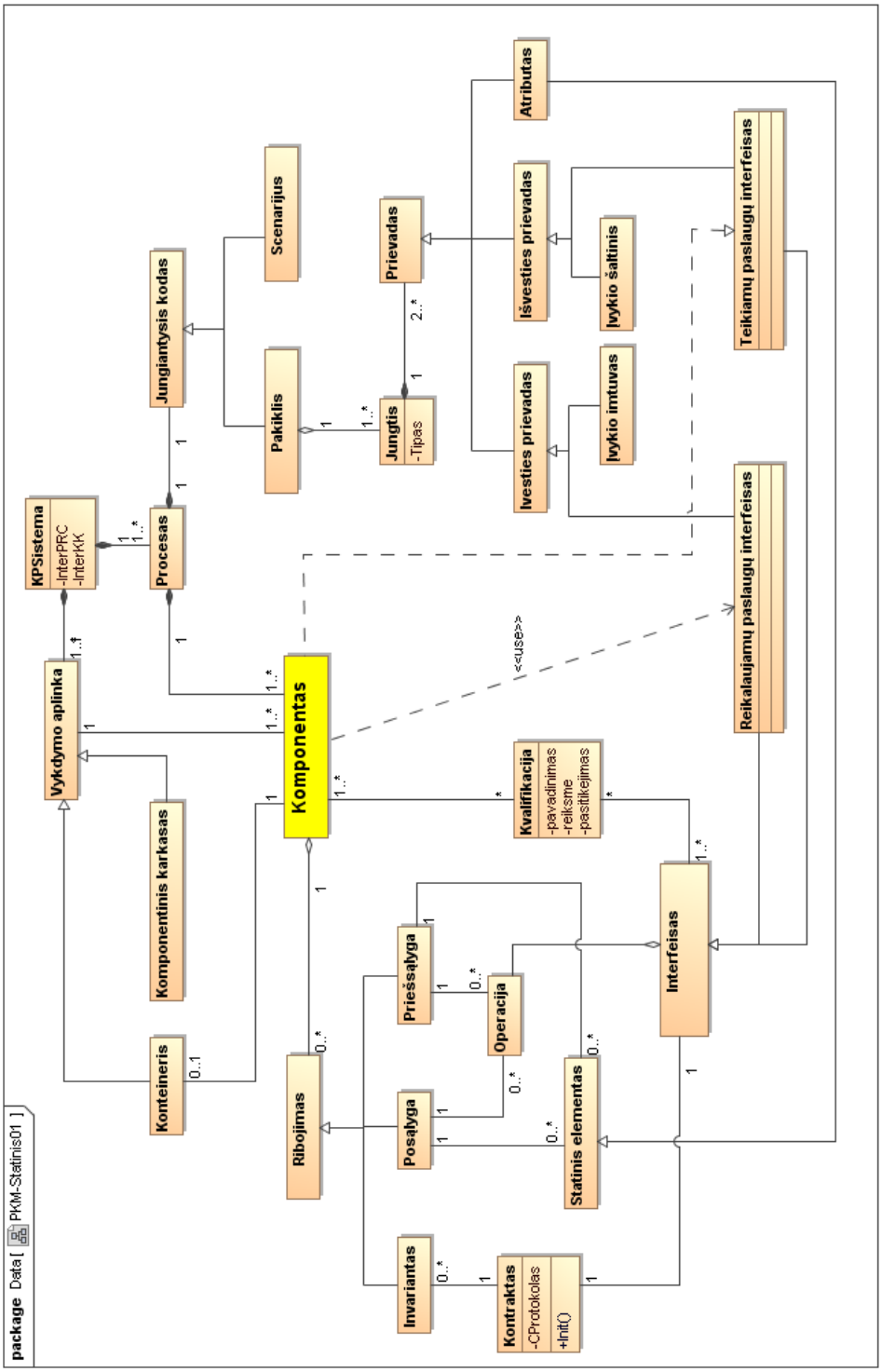
$$pr_i = \langle KS_i, JK_i \rangle , \tag{4.4}$$

kur yra KS_i – komponentų sąrašas, o JK_i – jungiantysis kodas.

Pažymėtina,kad jungiantysis kodas gali būti dviejų tipų:

- *pakiklių tipo* (angl. *wrappers*) [188, 41] jungiantysis kodas JK_i apibūdinamas jungčių $jng_{i,j}$ sąrašu;
- *scenarijus* [149]. Šis jungiamasis kodo tipas nebus nagrinėjamas, nes jis naudojamas valdymo srautų architektūroje.

Jungtys sujungia komponento prievadus. Prievadu čia vadinami bet kokie komponento įeities ar išeities taškai [41, 61, 4]. Prievadu gali būti laikomas: interfeisas, operacija, savybė, laukas, kintamasis. Pagal lygmenis prievadus klasifikuotini taip:



4.2 pav. Programinio komponento modelis

1. interfeisai,;
2. operacijos, laukai, savybės, įvykiai;
3. operacijų parametrai.

Jungtimi gali būti jungiami tik to paties lygmens prievadai. Kiekviena jungtis aprašoma:

$$jng_{i,j} = \langle P_{in}, P_{out}, JTipas \rangle, \quad jng_{i,j} \in JK_i, \tag{4.5}$$

kur j yra jungties lygmuo, P_{in} ir P_{out} – prievadai, kuriuos jungtis jungia, $JTipas$ – jungties tipas (*binding, mapping* ir kt.). komponentas vaizduojamas tokiu trejetu:

$$C = \left\langle PRT_{out}, PRT_{in}, \Delta, \Omega^C \right\rangle, \tag{4.6}$$

kur PRT_{in} ir PRT_{in} yra įvesties ir išvesties prievadų aibės atitinkamai:

$$PRT_{in} = \Im_{in} \cup E_{in} \cup S \tag{4.7}$$

$$PRT_{out} = \Im_{out} \cup E_{out} \cup S \tag{4.8}$$

Prievadu laikomas interfeisas, įvykis arba statinis elementas:

- $\Im_{out}$ yra $\Im_{in}$ atitinkamai siūlomų paslaugų (*angl. provided*) interfeisai ir naudojamų paslaugų (*angl. required*) interfeisai:

$$\Im_{out} = \left\{ I_1^{out}, I_2^{out}, \ldots, I_n^{out} \right\}; \Im_{in} = \left\{ I_1^{in}, I_2^{in}, \ldots, I_m^{in} \right\}, \tag{4.9}$$

Δ yra ribojimai, o Ω^C komponento kvalifikacijos (angl. *credentials*). komponento kvalifikacijų idėja nagrinėjama darbuose [37, 10]. Kiekviena kvalifikacija aprašoma trejetu:

$$\Omega^C = \langle KV, KR, KPTM \rangle, \tag{4.10}$$

kur KV - kvalifikacijos pavadinimas, KR – jos reikšmė, $KPTM$ – pasikliautinumas, t.y. nurodoma, kokiu būdu reikšmė buvo suteikta (pvz. ekspertui nurodžius, eksperimento būdu nustačius ir pan.). Kiekvienas interfeisas I gali būti specifikuojamas tokiu ketvertu:

$$I_i = \left\langle O_i, S_i, \Xi, \Omega_i^I, \right\rangle, \tag{4.11}$$

kur O - interfeiso operacijų aibė:

$$O_i = \{ o_{i,1}, o_{i,2}, \ldots, o_{i,k} \}; \tag{4.12}$$

S - interfeiso *statinių elementų* (darbuose [78, 4] interfeisą realizuojančių klasių laukų, šiuolaikinėse komponentinėse technologijose [13, 75, ?] – savybių (angl. *properties*)) aibė;
Ξ – interfeiso invariantai, Ω_i^I – interfeiso I kvalifikacijos. Kiekviena operacija $o_{i,j}$ specifikuotina kaip ketvertas:

$$o_{i,j} = \left\langle P, R, PreC, PostC, \Omega_{i,j}^O \right\rangle, \tag{4.13}$$

kur P ir R yra operacijos argumentų ir rezultatų aibės atitinkamai.:

$$P = \{p_1, p_2, \ldots, p_l\}; R = \{r_1, r_2, \ldots, r_s\}, \tag{4.14}$$

$PreC$ yra žymima operacijos "prieš"-sąlyga, $PostC$ – "po" sąlyga. Kaip ir interfeisas, kiekviena operacija turi tam tikras kvalifikacijas $\Omega_{i,j}^O$.

- E_{in} ir E_{out} – įvykių imtuvų ir įvykio šaltinių aibės atitinkamai.

$$E_{in} = (ev_{in,1}, ev_{in,2} \ldots ev_{in,n}) \tag{4.15}$$
$$E_{out} = (ev_{out,1}, ev_{out,2} \ldots ev_{out,m}) \tag{4.16}$$

- S – statinių elementų aibės (angl. *properties* arba *fields*).

$$S = (s_1, s_2 \ldots s_l) \tag{4.17}$$

Komponentinėje programų sistemoje gali būti naudojami ne tik vadinamieji *horizontalieji interfeisai*, jungiantys vieno lygmens elementus (pvz. Komponentus) tarpusavyje, bet ir *vertikalieji interfeisai* jungiantys skirtingų lygmenų elementus (pvz. Komponentą ir komponentinį karkasą). Komponentinės programų sistemos pateikties lygmuo yra ribinė sistemos dalis ir literatūroje yra nagrinėjama labai retai. Viena to priežasčių – komponentų vartotojo sąsaja (jos išvaizda, ergonomiškumas ir pan.) yra priklausoma nuo komponentų kūrėjų. Komponfentai yra juodosios dėžės ir sistemų kūrėjai turi labai mažas galimybes jas konfigūruoti.

Kita vertus vartotojai reikalavimų specifikacijoje gali pateikti reikalavimus vartotojo sąsajai, vadinasi tam, kad sukurti komponentinę programą būtina turėti ir komponentų vartotojo sąsajos (jei tokia yra) specifikacijas. Programinio komponento modelyje tiek komponento, tiek komponentinės sistemos pateiktis specifikuosime taip, kaip darbe [163] aprašomas *sisteminis interfeisas*:

$$SI = \langle E, Fd, Smd_p \rangle \tag{4.18}$$

kur E – baigtinė įvykių, į kuriuos reaguojama, aibė, Fd ir Smd_p – savybių ir operacijų aibės atitinkamai. Be to

$$E \subset E_{in} \cup E_{out} \tag{4.19}$$

$$Fd \subset S \tag{4.20}$$

$$Smd_p \subset \bigcup_{i} O_i \tag{4.21}$$

Pastebėtina, kad programinio komponento modelyje nenagrinėjami *uždarieji komponentai*, t.y. tokie, kurių teikiamų prievadų aibė tuščia. Visiems nagrinėjamiems komponentams

$$PRT_{out} \neq \emptyset. \tag{4.22}$$

Disertacijoje nagrinėjami tik *komponento specifikacijos* ir komponento realizacijos lygmenys, todėl bus naudojamas ne visas programinio komponento modelis, o tik tik jo dalis aktuali šiems lygmenims (4.3 pav.).

4.1.2. Dinaminis programinio komponento modelio aspektas

Komponento modelio dinaminis aspektas parodo galimas veiksmų su komponentu sekas bei apibrėžia kokią įtaką komponento egzempliorių būsenoms (jei tokios stebimos) daro vienos ar kitos operacijos, aprašytos interfeise [138]. Programinio komponento modelyje šį aspektą nusako kontraktas. Skiriami du kontaktų tipai [10, 34]:

1. **komponento kontraktas**, nusakantis komponento teikiamas ir reikalaujamas paslaugas [159, 121],
2. **sąveikos kontraktas** nusakantis roles, kurias komponentas atlieka sistemoje [46, 10, 162].

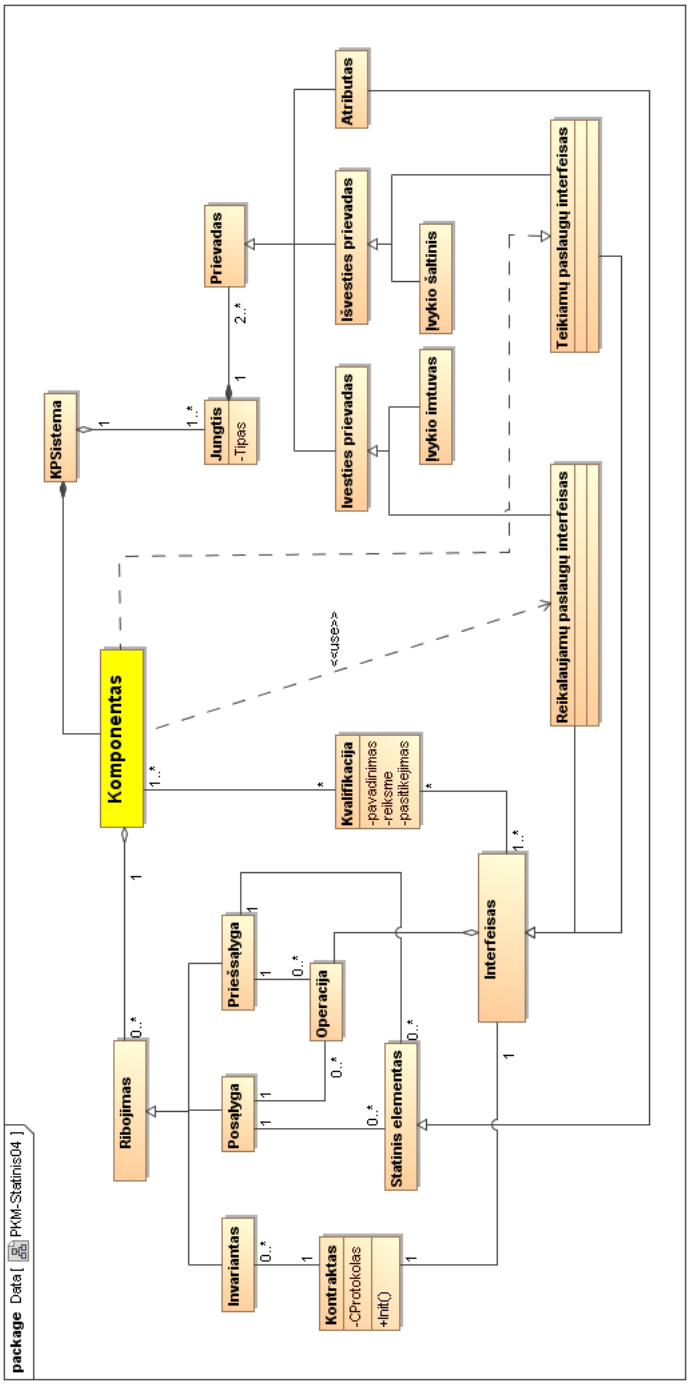
Programinio komponento modelyje komponento kontraktas nusakomas šiuo ketvertu:

$$Ctr = \langle I, Init, \Xi, CP \rangle , \tag{4.23}$$

kur I – interfeisas, kurį aprašo kontraktas, $Init$ – inicializavimo funkcija, Ξ – interfeiso invarinatai, CP – protokolas aprašantis, kokia tvarka gali būti kviečiamos operacijos, kokie yra ribojimai joms. Kokiomis priemonėsmis nusakomi ribojimai komponentams kituose lygmenyse parodyta 4.1 lentelėje.

komponentiniam protokolui aprašyti naudojama viena iš šių priemonių:

1. CSP (*Communicating sequential processes*) specifikacija [40];
2. Aprašant būsenas, įvykius ir leistinus veiksmus [142, 31];



4.3 pav. Programinio komponento specifikacijos ir realizacijos lygmenų komponento modelis.

3. Pranešimų sekomis [19];

4. Sąveikos šablonais [123].

4.1.3. Komponavimo kalba

Komponavimo kalba [1], turi turėti kalbos konstrukcijas skirtas aprašyti visus tris komponentinės paradigmos elementus: komponento modelį, komponavimo techniką ir pačią komponavimo kalbą. *U. Abman* taip pat įveda „komponavimo receptų“ (angl. *composing recipes*), kurie ir aprašomi būtent komponavimo kalba, sąvoką. „Komponavimo receptai“ gali įgauti įvairias formas. Darbe [1] pateikiami šie „komponavimo receptų“ pavyzdžiai:

- scenarijai (pvz.: [149]),
- vizualiųjų kalbų konfiguracijos (*editing sequences*),
- taisyklių rinkiniai,
- programos funkcinėmis kalbomis,
- imperatyviosios programos.

U. Abman iškelia šios reikalavimus komponavimo kalboms:

1. **Produkto dermė** (angl. *product-consistency*). Komponavimo receptai turi užtikrinti komponuojamos sistemos dermę/neprieštaringumą. Komponavimo kalba turi pasižymėti konkrečiomis savybėmis, kurios galima būtų patikrinti.

2. **Programinės įrangos proceso palaikimas** (angl. *software-process support*).

4.1 lentelė Ribojimų lygmenys.

Lygmuo	Konceptas	Elementai
Operacijos	$PreC$, $PostC$	
Interfeiso	I_i	$\Xi - -invariantai$
	Ctr	$\Xi - invariantai$
		CP-protokolas
komponento	C	$\Delta - ribojimai$
Programų sistemos	KPS	$InterPRC$-protokolas
Heterogeninės programų sistemos	KPS	$InterVA$-protokolas

PKM elementai	Atitinkmenys teorijose																	
	Apperly	Szypersky	UNU/IIST	UniFrame	FRACTAL	PECOS	UML komponentai	Poernomo	Cervantes	Salzmann	Berger	Aguirre ir Maibaum	Nierstrasz grupė	Yoshida ir Honiden	Lau grupė	Cox ir Song	Whitehead	Moschoyiannis
Komponentas	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
Interfeisas		+	+	+	+		+			+	+						+	
Teik. interfeisas		+	+	+	+	+	+								+		+	+
Reik. interfeisas			+		+										+		+	+
Komponentinis karkasas		+			+				+	+	+			+	+			
Komponentinė sistema			+									+		+		+		
Jungiantysis kodas		+											+					
Pakiklis		+											+					
Jungtis					+			+		+	+	+				+		
Prievadas						+										+		
Atributas					+											+		
Statinis elementas		+		+							+				+	+		
Įvykio imtuvas						+								+			+	
Įvykio šaltinis						+								+			+	
Operacija		+	+		+			+			+			+				+
Ribojimai			+					+			+			+				
Kontraktas			+	+		+	+	+		+								
Kvalifikacija *(Credential)*				+		+		+										

4.4 pav. Akademinių komponento modelių ryšys su Programiniu komponento modeliu.

PKM elementai	Atitinkmenys komponentinėse technologijose						
	.NET	CORBA	CCM	EJB	JavaBeans	WS	SCA
Komponentas	Assembly	+	+	SessionBeans, MessageBeans	+	WS	+
Interfeisas	+	+	+	+			+
Teik. interfeisas	+	+	+	+	+		+
Reik. interfeisas	+		+				+
Komponentinis karkasas	.NET Framework	ORB	ORB+ CCM	J2EE	JVM		
Komponentinė sistema	+		+				+
Jungiantysis kodas					+		
Pakiklis				+	+		
Jungtis			+	+	+		+
Prievadas			+	+	+	+	+
Atributas	Property		Attribute	Property	Property		+
Statinis elementas	+		+	+	+		
Įvykio imtuvas	+		+	+	+		+
Įvykio šaltinis	+		+	+	+		+
Operacija	+	+	+	+	+		
Kvalifikacija (Credential)	Custom attributes			Anotation			
Komponavimo kalba			CSD/CAD		BML	WSFL	XML
Komponento aprašymas	Manifest	IDL	CPD/CID	Manifest		WSDL	XML

4.5 pav. Komponentinių technologijų ryšys su Programinio komponento modeliu

3. **Metakomponavimas**. Patys „komponavimo receptai“ taip pat privalo būti komponentiniai.

Praktikoje naudojamos įvairios komponavimo kalbos. Beveik kiekviename iš 2.1. skyriuje apžvelgtų komponento modelių naudojama vis atskira komponavimo kalba: CDL [161], TrustME [141], rCOS [77], RADL [146], FML [82], konkurencinė logika [123]. Panašesniais galima būtų laikyti tik du sprendimus: [188] naudojama XML kalba, o [148] – jos pagrindu sukurta kalba SADL. PECOS komponento modelis yra ypatingai įdomus – čia naudojomos net dvi kalbos: πL ir PICOLLA.

Tokia pat įvairovė pastebima ir analizuojant populiariausiais komponentines technologijas: JavaBeans naudojama BML kalba [186], CCM – CSD/CAD [135], pasaulinio tinklo paslaugų modelyje – [184]. Įdomu tai, kad CORBA, .NET ir EJB komponento modeliuose atskira komonavimo kalba nenaudojama, komponentai jungiami programiniu kodu.

Palyginus komponavimo kalbas, matyti, kad plačiausiai naudojama XML kalba arba jos pagrindu sukurtos kalbos [148, 135, 184, 186, 188, 18]. Be to, kad XML yra plačiausiai naudojama kalba specifikavimui, padedanti spręsti nesuderinamumo problemas teigia ir K. Vaithed (*Katerine Whitehead*) [188]. Anot jos naudojant XML nebelieka parametrų sekos nesutapimo problemos, kadangi kiekvienas parametras XML faile yra identifikuojamas ne pagal poziciją o pagal gairę (angl. *tag*). Dėl išvardintų priežasčių disertacijoje komponavimui specifikacijos lygmenyje taip pat bus naudojama XML kalba.

4.2. Komponentinė Curry-Howard protokolo realizacija

4.2.1. Loginis skaičiavimas komponentinei paradigmai

Komponentinės sistemos gali būti formalizuojamos įvairiomis priemonėmis: λ-skaičiavimu [128], π-skaičiavimu [149], Petri tinklais [145], koalgebromis [77]. Šioje disertacijoje *programinio komponento modelis* nagrinėjamas *komponento specifikacijos* ir *komponento realizacijos* abstrakcijos lygmenyse, todėl jam formalizuoti pasirinktas *intuicionistinis teiginių skaičiavimas (ITS)*:

$$ITS = \langle Formulae(ITS), \vdash_{ITS}, DR \rangle , \tag{4.24}$$

- $Formulae(ITS)$: Didžiosiomis raidėmis (pvz X) su indeksais (arba be jų) žymėsime teiginius „*Programinio komponento Modelį atitinkantis komponentas turi prievadą* X“ ir toliau juos vadinsime propoziciniais kintamaisias (angl. *proposition variable*):

  $$X \in PRT_{in} \cup PRT_{out} \tag{4.25}$$

  Išskiriamas atskiras propozicinių kintamųjų atvejis – valdymo kintamieji. Šie propoziciniai kintamieji žymimi didžiosiomis arba mažosiomis raidėmis iš abiejų pusių apimant laužtiniais skliaustais, pavyzdžiui:

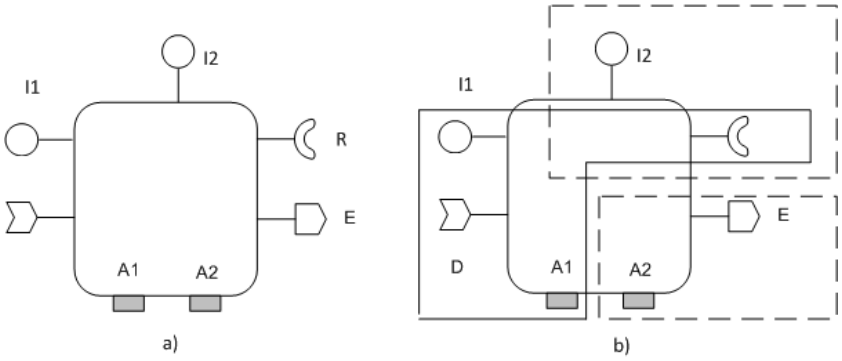
  $$[PuslapisAtspausdintas]. \tag{4.26}$$

  Be propozicinių kintamųjų dar naudojama konjunkcija ($\wedge$), implikacija ($\rightarrow$) ir skliaustai (()).
  Taisyklingų formulių sudarymo taisyklės apibrėžiamos rekursyviai:

  1. Kiekvienas termas yra formulė.
  2. Jei A yra formulė, tai (A) taip pat formulė.
  3. Jei A ir B yra formulės, tai $A \wedge B$ – formulė.
  4. Jei A ir B yra formulės tai $A \rightarrow B$ – formulė

  Kiekvieno komponento struktūra apibrėžiama tokio pavidalo aksiomomis:

  $$\bigwedge X_i \rightarrow \bigwedge_j Y_j \tag{4.27}$$



4.6 pav. Komponento prievadų loginių sąryšių pavyzdžiai

Pavyzdžiui, jei žinoma komponento įvesties ir išvesties prievadų tarpusavio priklausomybė (4.6 pav., b), komponentas specifikuojamas aksiomų aibe :

$$((A1 \wedge D) \wedge R) \rightarrow I1 \tag{4.28}$$

$$A2 \rightarrow E \tag{4.29}$$

$$R \rightarrow I2 \tag{4.30}$$

Jei komponento įvesties ir išvesties prievadų tarpusavio priklausomybė nežinoma (o taip gali būti, jei komponentas – *juodoji dėžė*, 4.6 pav., a), komponentas specifikuojamas viena aksioma :

$$A1 \wedge A2 \wedge D \wedge R \rightarrow I1 \wedge I2 \wedge E \tag{4.31}$$

- Naudosime intuicionistinio teiginių skaičiavimo **išvedimo taisykles (DR)** pateiktas 4.2 lentelėje.

Disertacijoje nenagrinėjami *įdiegto komponento* ir *komponentinio objekto* lygmenys, todėl *programinio komponento modelyje* neatsispindi ir šiems lygmenims būdingos esybės, tokios kaip lygiagretieji konkurenciniai procesai. Programinio komponento modeliui formalizuoti pasirinkto *intuicionistinio teiginių skaičiavimo* raiškos geba yra pakankama.

4.2.2. Loginė tipų teorija komponentinei paradigmai

Šiame poskyryje aprašoma loginė tipų teorija (LTT), sukurta intuicionistinio teiginių skaičiavimo (ITS) pagrindu:

$$LTT(ITS) = \left\langle PT(LTT), Formulae(LTT), (.)^{(.)}, \vdash_{LTT}, PTR, \triangleright \right\rangle \tag{4.32}$$

4.2 lentelė ITS išvedimo taisyklės (DR).

Įvedimo taisyklės	Eliminavimo taisyklės
$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ (\wedge I)$	$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ (\wedge E_L) \quad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ (\wedge E_R)$
$\frac{\Gamma, u : A \vdash B}{\Gamma \vdash A \to B}\ (\to I^u)$	$\frac{\Gamma \vdash A \to B \quad \Gamma \vdash A}{\Gamma \vdash B}\ (\to E)$

4.3 lentelė LTT tipai

$PT(LTT)$ elementas	LTT tipas
$\langle M, N \rangle$	$A \wedge B$
$fst\ N$	A , jei $N^{A \wedge B}$
$snd\ N$	B , jei $N^{A \wedge B}$
$\lambda u : A.M$	$A \to B$

- $PT(LTT)$ įrodymų termų aibę sudaro šie elementai:
  1. didžiosiomis raidėmis žymimi elementarūs termai (pvz. M) priklausantys poaibiui $Var_{PT(L)}$;
  2. $\langle M, N \rangle$ – pora;
  3. $fst\ N$ – pirmoji projekcija;
  4. $snd\ N$; – antroji projekcija;
  5. $\lambda u : A.M$ - lambda termas;
  6. MN - termų aplikacija.
- $Formulae(LTT)$ termų tipai aprašyti 4.3 lentelėje.
- Tipizavimo sąryšis $(p)^{(T)}$, apibrėžtas tarp įrodymo termų $p \in PT(LTT)$ ir tipų $T \in Formulae(LTT)$.
- Išvedimo taisyklės (PTR) Loginės tipų teorijos išvedimo taisykles parodytos 4.4 lentelėje.
- **Normalizavimo sąryšis** $\triangleright$ apibrėžtas virš įrodymų termų $PT(L)$, apibendrintas tranzityvus vieno žingsnio redukcijos uždarinys (angl. *closure*). Jį apibrėžia šios redukcijos taisyklės:

$$fst\ (< a, b >^{(A \wedge B)}) \quad \triangleright_{LTT} \quad a^A$$
$$snd\ (< a, b >^{(A \wedge B)}) \quad \triangleright_{LTT} \quad b^B$$

4.4 lentelė LTT išvedimo taisyklės (PTR).

Įvedimo taisyklės	Eliminavimo taisyklės
$\frac{\Gamma \vdash M^A \quad \Gamma \vdash N^B}{\Gamma \vdash < M, N >^{A \wedge B}} \ (\wedge I)$	$\frac{\Gamma \vdash M^{A \wedge B}}{\Gamma \vdash fst\ M^A} \ (\wedge E_L) \quad \frac{\Gamma \vdash M^{A \wedge B}}{\Gamma \vdash snd\ M^B} \ (\wedge E_R)$
$\frac{\Gamma, u^A \vdash M^B}{\Gamma \vdash \lambda u.M^{A \to B}} \ (\to I^u)$	$\frac{\Gamma \vdash A \to M^B \quad \Gamma \vdash N^A}{\Gamma \vdash MN^B} \ (\to E)$

4.2.3. Tipų teorijų atitiktis

4.5 ir 4.6 lentelėse pateiktos LTT ir CTT termų bei formulių atitiktys, kurios yra naudojamos komponentinei programai gauti. Šiuo atveju „programa“ vadinama – XML kalba aprašyta komponentinės programų sistemos architektūra, kuri gali būti realizuota pasirinkta programavimo kalba arba panaudota tiesiogiai (pvz., atlikus XSLT į WSDL kalbą).

Atliekant *etype* transformaciją (4.5 lentelė) kartu tikrinama ar loginės tipų teorijos formulė nėra *Harropo* [76] formulė. Taip iš įrodymo termų pašalinama nekonstruktyvioji informacija. Pagal įrodymo žingsnius atitinkamai generuojama programa.

Pagal apibrėžimą [76, 142] formulė F vadinama *Harropo* [76] formule jei tenkinama bent viena sąlyga:

- F yra pagrindinė (angl. *atomic*) formulė;
- F yra $(A \wedge B)$ pavidalo, kur A ir B - *Harropo* formulės;
- F yra $(A \to B)$ pavidalo, kur B yra *Harropo* formulė;

Toliau predikatą „formulė F yra *Harropo* formulė“ žymėsime $H(F)$, o predikatą „formulė F nėra *Harropo* formulė“ žymėsime $\neg H(F)$.

Disertacijoje aprašomame automatizuoto komponentinių programų sistemų kūrimo metodas ne tik realizuoja *Curry-Howard* protokolą, bet ir naudoja kitų generavimo metodų elementus. Tolimesniuose poskyryje aprašoma kurie konkretūs struktūrinės programų sintezės (4.3. poskyris) ir induktyviojo (4.4. poskyris) metodų elementai panaudoti disertacijoje aprašomame generavimo metode.

4.3. Struktūrinės programų sintezės elementai

Kaip jau minėta, struktūrinės programų sintezės metodas (žr. 3.3. skyrių) buvo sukurtas programoms generuoti iš modulių, vėliau tirtos jo pritaikymo

4.5 lentelė LTT ir CTT termų atitikties funkcija $etype(F)$.

LTT termas F	$etype(F)$
P	*komponentas*
(P)	*komponentas*
$(A \wedge B)$	$\begin{cases} etype(A) & \text{jei } \neg H(B) \\ etype(B) & \text{jei } \neg H(A) \\ etype(A) * etype(B) & \text{kitais atvejais} \end{cases}$
$(A \to B)$	$\begin{cases} etype(B) & \text{jei } \neg H(B) \\ etype(A) \to etype(B) & \text{kitais atvejais} \end{cases}$

objektinei ir paslauginei paradigmoms galimybės. Šiame poskyryje pateikiami tik tie jo elementai, kurie papildo *Curry-Howard* protokolo realizaciją ir gali būti naudojami kaip automatizuoto komponentinių programų kūrimo metodo dalis.

Įrodomojo programavimo metodų klasė, kurios naudojimą apibendrina *Curry-Howard* protokolas, apima ir struktūrinės programų sintezės metodą. Konkrečiai disertacijoje aprašomoje *Curry-Howard* protokolo realizacijoje panaudoti šie SSP elementai:

1. teiginių išvedimo intuicionistiniame teiginių skaičiavime taisyklės (4.2 lentelė),

2. uždavinio sprendinio egzistavimo teoremoms įrodyti naudojama ATP, realizuojanti struktūrinės sintezės algoritmus,

3. uždavinio formulavimo būdas. Komponentinių sistemų sintezės uždavinio atveju pakartotinai panaudojami artefaktai yra komponentai. Tačiau sintezės procese veiksmai su komponentais atliekami prievadų lygmenyje:

   $$\text{rasti } P_1^{out}, P_2^{out}, \ldots, P_n^{out}, \Omega^C \text{ turint } P_1^{in}, P_2^{in}, \ldots, P_m^{in} \text{ remiantis } M.$$

   kur P_P^{out} Pr P_j^{in} yra komponento išvesties ir įvesties prievadai, Ω^C – komponento kvalifikacijos. Kiekvienas propozicinis kintamasis atitinka konkretų prievadą, o kiekviena aksioma aprašo ryšį tarp įvesties ir išvesties prievadų.

4.6 lentelė LTT ir CTT formulių atitikties funkcija $extract(p^T)$.

LTT formulė (p^T)	$extract(p^T)$
Bet kuris įrodymo termas p^T	(), jei $H(T)$
u^A	$\begin{cases} x_u \text{jei } \neg H(A) \\ () \qquad\qquad \text{jei } H(A) \end{cases}$
$\langle a^A, b^B \rangle$	$(extract(a), extract(b))$
$fst\ a$	$fst\ (extract(a))$
$snd\ a$	$snd\ (extract(a))$
$c^{A \to B} a^A$	$\begin{cases} extract(c) & \text{jei H(A)} \\ (extract(c)\ extract(a)) & \text{jei } \neg H(A) \end{cases}$

Analizuojant struktūrinės sintezės metodą nustatyta, kad jį naudojant komponentinių programų sintezės sistemoje gali kilti šios problemos [64]:

- **Neišsamios specifikacijos problema**. Jei vartotojo pateikta specifikacija yra netiksli arba su klaidomis, sintezės rezultatas – nekorektiškas. Kitos uždavinio specifikavimo aukšto lygmens kalba problemos (pvz., specifikavimo kalbos raiškos geba, patogumas vartotojui) nėra būdingos vien SSP metodui [62, 63], todėl čia nebus nagrinėjamos.
  Klasikinis SSP metodas negali būti panaudojamas kitokio nei valdymo srautų ir duomenų srautų architektūros stilių programoms kurti. *S. Lammerman* ir *E. Tyugu* darbe [102] pasiūlytas *išplėstasis SSP metodas* (ESSP) įgalina kurti objektinio architektūros stiliaus programas, gebančias reaguoti į išimtis (*angl. exception*). Tačiau ir ESSP nesuteikia galimybės kurti įvykiais valdomo architektūros stiliaus programas.

- **Neapibrėžtųjų komponentų problema**. Kaip minėta 2.4. skyriuje, komponentinių programų sistemų gyvavimo ciklas skiriasi nuo struktūrinių programų gyvavimo ciklo, todėl kūrimo pradžioje dažniausiai nėra žinomi visi sistemai sukurti reikalingi komponentai. Tuo tarpu naudojant struktūrinės sintezės metodą, daroma prielaida, kad visos aksiomos turi jas realizuojančius komponentus ir reikia tik išspręsti jų pasirinkimo uždavinį. Sintezės sistemai neradus reikiamos aksiomos (o kartu ir komponento) daroma išvada, kad uždavinys neišsprendžiamas. Sintezės

sistema negali nustatyti, kokių konkrečiai aksiomų ir kokių konkrečiai komponentų trūksta.

Kita aktuali problema - SSP reikalavimas, kad visos būsimosios programos sudėtinės dalys (šiuo atveju, komponentai) būtų specifikuotos prieš sintezės procesą. SSP metodo kūrėjai daro prielaidą, jog visi reikiami komponentai jau yra, tik reikia išspręsti jų pasirinkimo uždavinį. Jei nors vienas sistemai sukurti reikalingas komponentas neaprašytas aksioma, gali susidaryti situacija, kai teoremai įrodyti pritrūks būtent šios aksiomos.
Tuo tarpu vienas iš komponentinių programų kūrimo etapų yra komponentų paieška [149]. Be to, ši paieška turi būti vykdoma realizacijos lygmenyje, jau suprojektavus programų sistemą. Vienas iš galimų neapibrėžtųjų komponentų problemos sprendimo būdų - pakeisti įrodymo paieškos algoritmą.

Tarkime, kad M - sintezės uždavinio aksiomų aibė. Kiekviena aksioma m turi pavidalą:

$$u_1, u_2, \dots u_k \longrightarrow v_1, v_2, \dots v_l, \tag{4.33}$$

K - visų bent vienai aksiomai priklausančių propozicinių kintamųjų aibė:

$$\forall i, ju_i, v_j \in K \tag{4.34}$$

o A - algoritmas, analizuojantis aibes K ir M, ieškantis teoremos

$$p_1, p_2, \dots p_n \longrightarrow r_1, r_2, \dots r_m; \forall i, jp_i, r_j \in K \tag{4.35}$$

įrodymo. Tarkime, kad A baigė darbą, tačiau teoremos įrodymas nebuvo rastas. Tada aibė M papildoma *virtualiomis aksiomomis*:

$$a : u_1, u_2, \dots u_k \longrightarrow v_1, v_2, \dots v_l, k, l > 0, a \in M' \tag{4.36}$$

ir gautai aibei $M' = M \cup M''$ vėl taikomas algorimas A. Galima įrodyti, kad aibėje M' algoritmas A visada baigs darbą ir ras mažiausiai vieną sprendinį.

Virtuliosiomis aksiomis (VA) vadinamos aksiomos, aprančios vienų propozicinių kintamųjų reikšmių gavimo naudojant kitų propozicinių kintamųjų reikšmes, tačiau neturinčios realizacijos. Turimų komponentų aibėje nėra nei vieno tokio, kuris būtų susijęs su kokia nors VA ryšiu *„vienas-su-vienu“*.

Algoritmui A baigus darbą ir radus teoremos įrodymą, sintezės sistemos naudotojui tampa prieinama informacija apie tai, kokios VA įrodyme turi

4.7 lentelė SSP algoritmų sudėtingumas laiko atžvilgiu.

Programos struktūra	Sudėtingumas
tiesinė	$c_1 \cdot n^2$
besišakojanti	$c_1 \cdot n^2 \cdot 2^n$
su použdaviniais	$c_1 \cdot n^3 \cdot 2^n$

būti panaudotos, o tai savo ruožtu parodo, kokių komponentų trūksta gauti galutiniam rezultatui - komponentinei programai.

Metodas yra gana paprastas, tačiau turi ir trūkumų. Įvertinkime papildomų aksiomų įvedimo įtaką teoremos įrodymo paieškos algoritmui. Tarkime, kad po papildymo aibėje M' yra visos aksiomos, kokias tik galima sudaryti iš K aibės propozicinių kintamųjų. Tada, bendras aibės $M\prime$ aksiomų skaičius bus

$$n(M') = \sum_{i=1}^{k} C_k^i \cdot \sum_{j=1}^{k} C_k^j, \tag{4.37}$$

kur k – propozicinių kintamųjų aibėje K skaičius.
Atmeskime iš šios aibės $A \longrightarrow A$ pavidalo aksiomas, nes, kaip minėta anksčiau, tokio tipo aksiomos laikomos trivialiomis ir dažniausiai nenaudojamos. Tokių aksiomų gali būti tiek, kiek yra skirtingų gretinių sudarytų iš aibės K elementų, todėl:

$$n(M') = \sum_{i=1}^{k} C_k^i \cdot \sum_{j=1}^{k} C_k^j - \sum_{i=1}^{k} C_k^i = \sum_{i=1}^{k} C_k^i (\sum_{j=1}^{k} C_k^j - 1). \tag{4.38}$$

Yra žinoma [105], kad

$$\sum_{i=0}^{k} C_k^i = 2^k,$$

todėl

$$n(M') = (2^k - 1)(2^k - 2) \approx 2^{2k}. \tag{4.39}$$

Palyginus gautus rezultatus su [166] pateiktais SSP algoritmų sudėtingumo įverčiais (4.7 lentelė) matyti, kad papildomų aksiomų įvedimas gali labai padidinti įrodymo paieškos laiką.

Kita vertus, didelis VA skaičius gali sukelti įrodymo nevienareikšmiškumo ir rezultatų nepatikimumo problemas.
Kuo daugiau virtualiųjų aksiomų yra įvesta, tuo daugiau skirtingų teoremos įrodymų gali būti rasta. Jei kiekviename iš šių įrodymų yra

panaudota bent viena VA, tai sėkmingam sintezės procesui reikia rasti ją atitinkantį komponentą. Jei atitinkamas komponentas neegzistuoja, sintezės sistemos naudotojas gali jį sukurti, arba atmesti nagrinėjamą teoremos įrodymą kaip nepagrįstą ir procesą tęsti su kitu gautu įrodytų, kitais komponentais. Įvertinus komponentų paieškos ir skirtingų įrodymų įvertinimo laiką, galima teigti, kad sintezės procesas pailgėja.
Didelis virtualiųjų aksiomų skaičius leidžia įrodyti beveik bet kokią 4.35 pavidalo teoremą. Teorema neįrodoma tik tuo atveju, jei jos sąlygoje yra bent vienas propozicinis kintamasis nepriklausantis aibei K. Dėl šios priežasties gali būti surastas toks teoremos įrodymas, kad VA atitinkantys komponentai arba neegzistuoja, arba yra nesuderinami semantine prasme. Pavyzdžiui, teoremos įrodyme panaudota aksioma

$$PinigineSuma \longrightarrow TaskoEkraneSpalva$$

neturi ją realizuojančio komponento.

4.4. Induktyviojo metodo elementai komponentinių programų sistemų kūrimo metode

Vienas induktyviojo metodo privalumų, lyginant jį su dedukciniu metodu – galimybė atlikti loginių programų sintezę remiantis ir nebaigtomis specifikacijomis [125, 71]. Šią metodo savybę galima panaudoti ir komponentinių programų specifikacijos problemai spręsti.

Tokiu atveju generavimo procesas pasipildytu dar vienu etapu, kuriame indukcijos būdu būtų tikslinamos nebaigtos ar nekorektiškos specifikacijos. Po šio etapo sektų specifikacijos analizavimo ir įrodymo paieškos etapai.

Kaip teigiama [125], induktyvusis metodas padeda nustatyti ryšius tarp konceptų. Pastaroji metodo savybė yra labai svarbi sprendžiant neapibrėžtųjų komponentų problemą. Generavimo procese po įrodymo paieškos etapo siūloma įterpti dar vieną etapą, kuriame būtų analizuojama dėl kokių priežasčių nepavyko įrodyti teoremos. Jei teoremos įrodymo etape gautas teigiamas rezultatas (t.y. teorema įrodyta) – vadinasi uždavinys išspręstas ir neapibrėžtųjų komponentų problemos nėra. Šiuo atveju papildomas etapas būtų tiesiog praleidžiamas. Tačiau, jei teoremos įrodyti nepavyko, papildomame etape naudojant indukcijos metodą, būtų bandoma nustatyti, kokių aksiomų trūksta ir ar jos gali būti realizuotos. Papildžius uždavinio specifikaciją naujomis generuotomis aksiomomis būtų vėl grįžtama prie teoremos įrodymo etapo.

Tam, kad būtų išspręsta nefunkcinių reikalavimų problema, generavimo metodas turi užtikrinti galimybę įvertinti komponentus nefunkcinių reikalavimų atžvilgiu, t.y. sistema turi gebėti iš komponentų aibės parinkti tokį komponentą, kurio veikimo sparta, apsaugos lygmuo ir kitos nefunkcinės savybės

geriausiai atitinka specifikaciją. Induktyvusis metodas galėtų padėti nustatyti komponentų alternatyvas sistemos nefunkciniams reikalavimams patenkinti. Be to, induktyvusis metodas naudotinas ieškant aukštesnio lygmens dėsningumų (*angl. high-level regularities*) [125], o tai gali padėti ieškant bendresnių komponentų ar jų prievadų.

Induktyviojo metodo elementus komponentinių programų sistemų generavimo procese siūloma naudoti siekiant išspręsti šias problemas:

- specifikacijos problemą;
- neapibrėžtųjų komponentų problemą;
- nefunkcinių reikalavimų problemą.

Tačiau, siekiant išvengti nekorektiško rezultato, visos sintezės metu padarytos induktyviosios išvados privalo būti papildomai tikrinamos.

4.5. Metodo taikymo pavyzdžiai

Panagrinėsime du supaprastintus komponentinių programų sistemų kūrimo pavyzdžius.

4.5.1. Vaistų dozatoriaus programa

Reikia sukurti programų sistemą pagal asmens fizinius duomenis rekomenduojančią tam tikro medikamento dozę.
Turimi komponentai:

1. Komponentas **CKMI**, gebantis pagal asmens fizinius duomenis apskaičiuoti žmogaus kūno masės indeksą (KMI). Šis komponentas realizuoja interfeisą **IKMI**:

   ```
   interface IKMI {
   float KMIcalc(float svoris, float ugis)
   }
   ```

   jokių kitų prievadų nėra, komponentą aprašo vienintelė aksioma:

   $$\rightarrow IKMI \tag{4.40}$$

2. Komponentas **CMD**, gebantis pagal paciento kūno masės indeksą apskaičiuoti maksimalią leistiną vaistų dozę. Šis komponentas realizuoja interfeisą **Idoze**:

```
interface Idoze {
float doze(int kartai, time periodas, float kiekis)
}
```

komponentą aprašo aksioma:

$$IKMI \rightarrow Idoze \tag{4.41}$$

3. Komponentas **CVR**, gebantis apskaičiuoti rekomenduojamą pagal vaistų dozę atsižvelgiant į maksimalios leistinos vaistų dozės reikšmę. Šis komponentas realizuoja interfeisą **IVR**:

```
interface IVR {
float rd(Collection simptomai, float kmi, int amzius)
}
```

ir reikalauja komponentų realizuojančių tokius interfeisus: **IMKI**, **Idoze**. Komponentą aprašo aksioma:

$$IKMI, Idoze \rightarrow IVR \tag{4.42}$$

Komponentinės programos uždavinį užrašysime kaip teoremą:

$$\rightarrow IVR \tag{4.43}$$

Paeiliui taikydami išvedimo taisykles gauname teoremos įrodymą:

$$\dfrac{\dfrac{\rightarrow IKMI \quad IKMI \rightarrow Idoze}{\rightarrow Idoze} \qquad IKMI, Idoze \rightarrow IVR}{IKMI, Idoze \rightarrow IVR}$$

4.5.2. Videomedžiagos transliacija

Tarkime, kad komponentų saugykloje yra keturi komponentai, skirti vaizdo medžiagos apdorojimui:

1. Vaizdo medžiagos konvertavimo (VMK) komponentas. Komponentas vieną pasirikto vaizdo formato rinkmeną konvertuoja į kito pasirikto vaizdo formato rinkmeną. Jei yra įdiegti papildomi *kodekai*, komponentas geba dirbti su šių formatų rinkmenomis: **.avi*, **.mpeg*, **.wmv*:

$$(AVI \to MPEG) \to ([r] \to MPEG)$$
$$(AVI \to WMV) \to ([r] \to WMV)$$
$$(WMV \to AVI) \to ([r] \to AVI)$$
$$(WMV \to MPEG) \to ([r] \to MPEG)$$
$$(MPEG \to WMV) \to ([r] \to WMV)$$
$$(MPEG \to AVI) \to ([r] \to AVI)$$

2. Kodekas K1 – komponentas realizuoja duomenų konvertavimo iš **.avi* į **.wav* formatą algoritmą:

$$AVI \to WMV \tag{4.44}$$

3. Kodekas K2 – komponentas realizuoja duomenų konvertavimo iš **.mpeg* į **.avi* formatą algoritmą:

$$AVI \to MPEG \tag{4.45}$$

4. Vaizdo medžiagos transliavimo internetu komponentas (VMT), kurio įvestis *.wav formato vaizdo rinkmena, o išvestis - transliacija internetu:

$$WMV \to [t] \tag{4.46}$$

Spręskime tokį uždavinį – reikia sukurti programų sistemą, kuri bet kurią AVI formato laikmeną transliuotų internetu:

$$[r], AVI \to [t] \tag{4.47}$$

Teoremos įrodymas:

$$\dfrac{\dfrac{AVIAVI \to WMV}{AVI \to WMV} \quad \dfrac{(AVI \to WMV) \to ([r] \to WMV)}{[r] \to WMV} \qquad WMV \to [t]}{[r] \to [t]}$$

4.6. Skyriaus išvados

1. Išanalizavus komponento modelius, reprezentuojančius esamas jų klases, nustatyta:

   - Gali būti sukurtas apibendrintas *programinio komponento modelis*.
   - Komponento specifikacijos ir realizacijos lygmenyse naudojamas apibendrintas *programinio komponento modelis* turi būti sudarytas iš šių elementų: komponento, prievado, jungties, kontrakto, ribojimų, kvalifikacijų ir komponentinės sistemos.

2. Programų sintezės uždavinys ir jo sprendimo būdas gali būti taip suformuluoti apibendrinto komponento modelio terminais, kad gautas sprendinys gali būti pritaikomas kiekvienam iš apibendrintų komponentų modelių panaudojant konkretizavimo, patikslinimo ir papildymo operacijas.

3. Programų sintezės metodas, komponentinėje paradigmoje realizuojantis *Curry-Howard* protokolą, įgalina automatizuotu būdu kurti komponentines programas ir užtikrina šių programų atitiktį specifikacijai.

5 skyrius

Komponentinių sistemų generavimo metodo realizacija

Šiame skyriuje aprašoma disertacijos autoriaus sukurta komponentinių programų sintezės sistema SoCoSyS[1]. Ši sistema sukurta siekiant realizuoti ir eksperimentiškai patikrinti 3 skyriuje pasiūlytą komponentinių programų sistemų generavimo metodą. Šio skyriaus tikslas – aprašyti metodo tikrinimo eksperimentą ir jo rezultatus.

5.1. SoCoSyS sistema

Metodas, kurį realizuoja SoCoSyS sistema operuoja dviejuose komponento abstrakcijos lygmenyse: *komponento specifikacijos* ir *komponento realizacijos*. Kadangi generavimo proceso rezultatas priklauso nuo lygmens, nuo lygmens priklauso ir pati generavimo sistema. Kas yra būdinga komponento specifikacijos lygmeniui skirtai generavimo sistemai, aptarsime 5.1.2. poskyryje, kas būdinga komponento realizacijos lygmeniui – 5.1.3. poskyryje, o čia (5.1. poskyryje) aptarsime bendrus abiems lygmenims generavimo sistemos elementus.

SoCoSyS sistemos įvestimi yra uždavinio specifikacija, uždavinio teorija ir pakartotinai panaudojami artefaktai – komponentai. SoCoSyS sistemos bendrieji elementai yra šie (5.2 pav.):

- analizatorius (angl. *parser*),
- planavimo modulis,

[1] pavadinimas parinktas naudojant pirmąsias sistemos paskirties apibūdinimo anglų kalba – *Software Component Synthesis System*– raides.

- optimizavimo modulis,

- architektūros generatorius,

Analizatorius kaip argumentus naudoja iš uždavinio specifikacijos atskirtus funkcinius reikalavimus. Juos išskaido į įvesties bei išvesties kintamuosius ir suformuoja uždavinio medį – uždavinį vaizduojančią duomenų struktūrą.

Planavimo modulis – pagrindinė sistemos dalis. Ji, kaip argumentus ima uždavinio medį ir uždavinio teoriją. Modulio rezultatas yra konstruktyvusis uždavinyje nurodytos teoremos įrodymas. Planavimo modulio vaidmenį sistemoje atlieka automatinio teoremų įrodymo programa. Eksperimento metu naudota disertacijos autoriaus sukurta ATP *SoCoProove* (žr. 5.1.1. poskyrį), nors galima naudoti ir kurią nors kitą ATP, gebančią įrodyti intuicionistinio teiginių skaičiavimo teoremas bei tenkinančią kitus reikalavimus (žr. 3.2.2. poskyrį).

Optimizavimo modulio tikslas – eliminuoti nereikalingus įrodymo žingsnius. Šie pertekliniai žingsniai gali būti generuojami planavimo moduliui vykdant prielaidomis grįstą paiešką pirmyn (angl. *assumption driven forward search*).

Jei įrodymas rastas, *generatorius* naudodamas gautą konstruktyvųjį įrodymą generuoja „programą“.

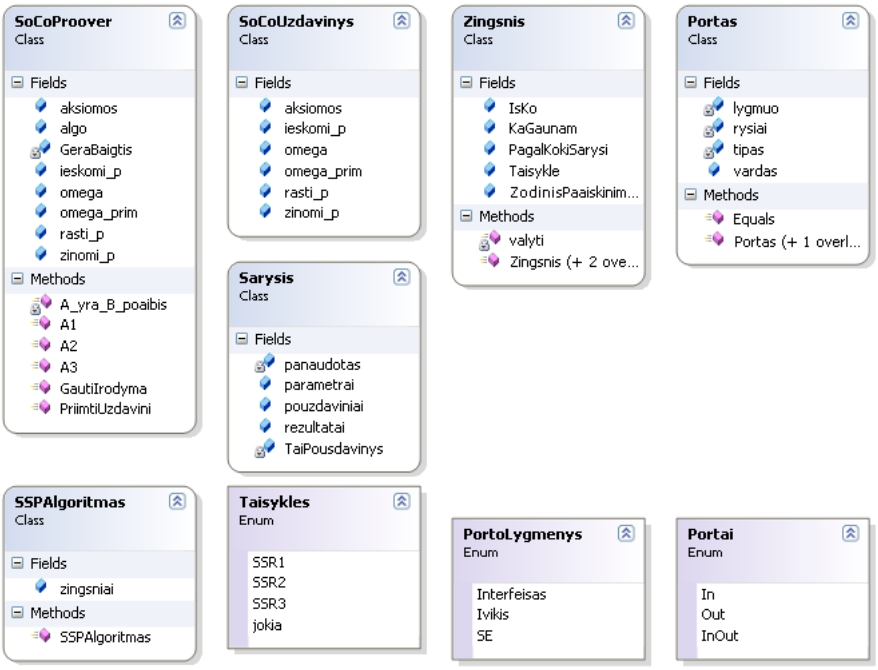
„Programos“ savoka priklauso nuo nagrinėjamo komponento abstrakcijos lygmens. Atskiros sintezės sistemos šakos, adaptuotos konkrečiam lygmeniui, aprašytos 5.1.2. ir 5.1.3. poskyriuose atitinkamai.

5.1.1. Teoremų įrodymo programa *SoCoProove*

SoCoProove yra disertacijos autoriaus sukurta automatinio teoremų įrodymo programa, atliekanti veiksmus intuicionistinio teiginių skaičiavimo pagrindu sukurtoje loginėje tipų teorijoje (4.2.2. poskyris).

SoCoProove realizuoja tris įrodymo paieškos algoritmus: A1 (3.3.2.2. poskyris), A2 (3.3.2.3. poskyris) ir A3 (3.3.2.4. poskyris). Programa nepriskiriama interaktyviųjų teoremų įrodymo programų klasei, teoremos paieška vyksta visiškai automatiniu būdu, be galimybės pateikti papildomus parametrus (pvz. parinkti taktikas) paieškos metu. Ši ATP neturi grafinės ar komandinės eilutės sąsajos, ji skirta naudoti kitose sistemose kaip API funkcijų biblioteka. *SoCoProove* naudojamos duomenų struktūros pavaizduotos 5.1 pav., o *SoCoProove* programa ir jos pirminis tekstas viešai skelbiami *http://ik.su.lt/∼vaigie/socosys/* svetainėje.

SoCoProove teoremų įrodymo programa bendroje SoCoSyS sistemoje atlieka planavimo modulio (5.2 pav.) vaidmenį.



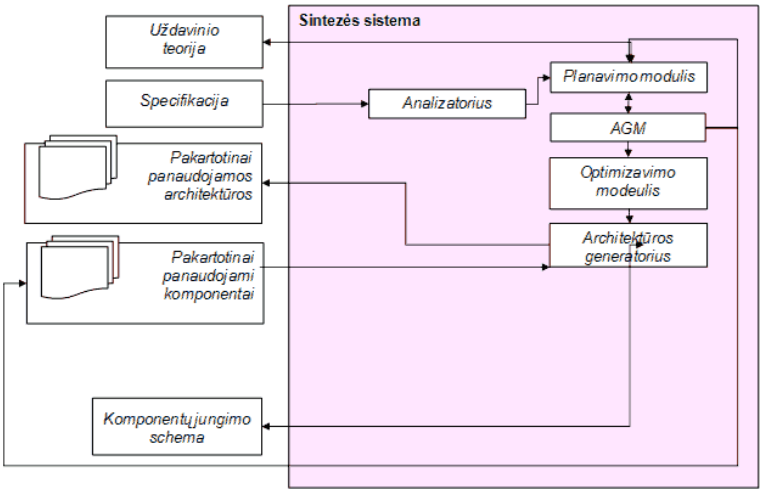
5.1 pav. SoCoProove klasių diagrama.

5.1.2. Komponento specifikacijos lygmens programų sintezės sistema

Šios sintezės sistemos rezultatas (anksčiau vadintas „programa“) – programų architektūra nusakanti komponentų jungimo schemą (5.2 pav.). SoCoSyS sistemoje komponentinės programos architektūra aprašoma *XML* pagrindu sukurta kalba. Naudojant XSLT transformacijas šią schema galima automatiniu būdu užrašyti ir kita (pvz. WSFL ar SCA modelyje naudojama) kalba. komponentinių programų sintezės sistema privalo būti pritaikyta gyvavimo ciklui *Išrink-Pritaikyk-Bandyk*, todėl ji, skirtingai nei dauguma generavimo programų, išsaugo informaciją apie sintezuotų programų architektūras.

Generatorius naudodamas teoremos įrodymą generuoja komponentų jungimo schemą, ją išsaugo architektūrų bazėje ir pateikia kaip sintezės proceso rezultatą.
komponento specifikacijos lygmens programų sintezės sistema automatizuoja šiuos komponentinių programų sistemų gyvavimo ciklo etapus:

- komponentų atranka ir įvertinimas,
- sistemos architektūros projektavimas.



5.2 pav. Komponento specifikacijos lygmens programų sintezės sistema.

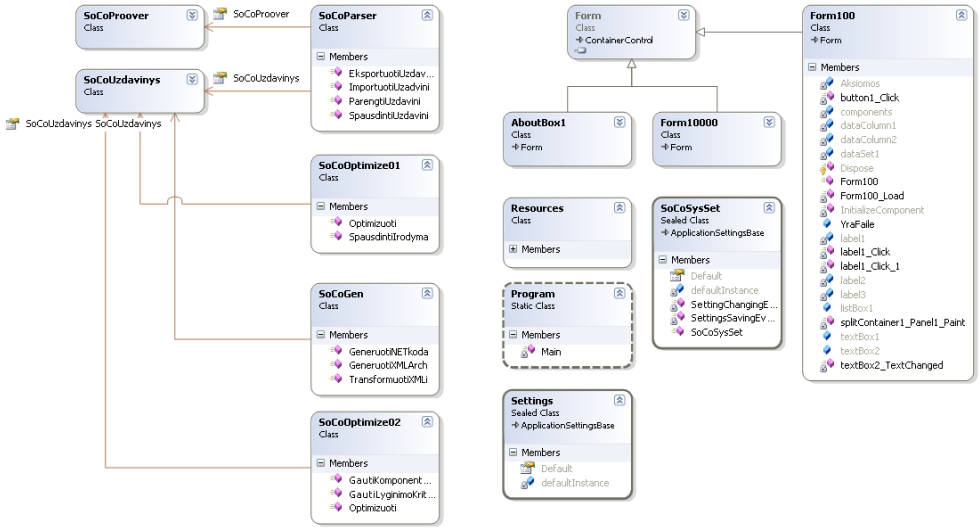
komponento specifikacijos lygmenyje neatsižvelgiama į nefunkcinius reikalavimus komponentinei sistemai ir pačių komponentų kvalifikacijas. Komponentų alternatyvų parinkimas įvertinant nefunkcinius reikalavimus paliekamas tolimesniam etapui.

SoCoSyS naudojamos duomenų struktūros pavaizduotos 5.3 pav., o pati *SoCoSyS* programa ir jos pirminis tekstas viešai skelbiami svetainėje *http://ik.su.lt/∼vaigie/socosys/*.

5.1.3. Komponento realizacijos lygmens programų sintezės sistema

komponento realizacijos lygmens programų sintezės sistemos rezultatas – komponentinės programų sistemos realizacija, t.y. Komponentinė programa.

Skirtingai nei specifikacijos lygmens sintezės sistemoje, čia naudojami du generatoriai. *Architektūros generatorius* savo darbo rezultatą – komponentų jungimo schemą perduoda *programos generatoriui*. Pastarasis komponentų bazėje suranda reikiamus komponentus ir įstatydamas juos į apibrėžtas vietas, generuoja programą. (5.4). Siekiant visiško suderinamumo su *Išrink-Pritaikyk-Bandyk* gyvavimo ciklu, sistema atsižvelgia ir į nefunkcinius reikalavimus. Tai sistemoje atlieka *antrasis optimizavimo modulis*. Šis modulis naudodamas anksčiau išsaugotą jungimo schemą iš komponentų bazės parenka tokius komponentus, kurie geriau užtikrina, kad programa tenkintų nefunkcinius reikalavimus ir jais (komponentais) pakeičia jau esančius sistemoje. Eksperimentinio sistemos testavimo metu buvo naudoti komponentai, su ne daugiau kaip viena kvalifikacija, todėl tinkamesnio komponento paieška yra elemen-



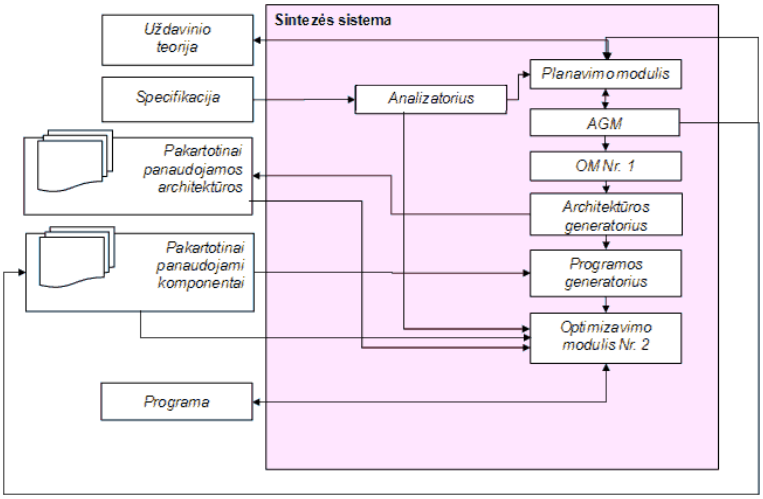
5.3 pav. SoCoSyS sistemos klasių diagrama

tari. Tuo atveju, jei komponentai galėtų turėti neribotą skaičių skirtingų kvalifikacijų, komponento kokybės klausimas galėtų būti sprendžiamas taikant standartines procedūras, pavyzdžiui laikantis ISO/IEC TR 9126 standarto [87].

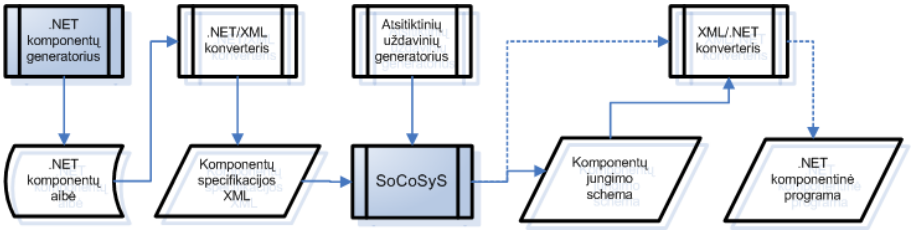
Neapibrėžtųjų komponentų problemai spręsti numatytas, tačiau dar nerealizuotas, aksiomų generavimo modulis (AGM), kurio paskirtis aprašyta 4.4. poskyryje. komponento realizacijos lygmens programų sintezės sistema automatizuoja šiuos komponentinių programų sistemų gyvavimo ciklo etapus:

- komponentų atranka ir įvertinimas,
- sistemos architektūros projektavimas,
- sistemos realizavimas,
- sistemos integravimas.

Be to disertacijoje siūlomas komponentinių programų automatizuoto kūrimo metodas sumažina tikrinimo ir įvertinimo etapo poreikį, nes gauta programinė įranga visiškai atitinka specifikaciją.



5.4 pav. Komponento realizacijos lygmens programų sintezės sistema.



5.5 pav. Eksperimento su SoCoSyS sistema eiga.

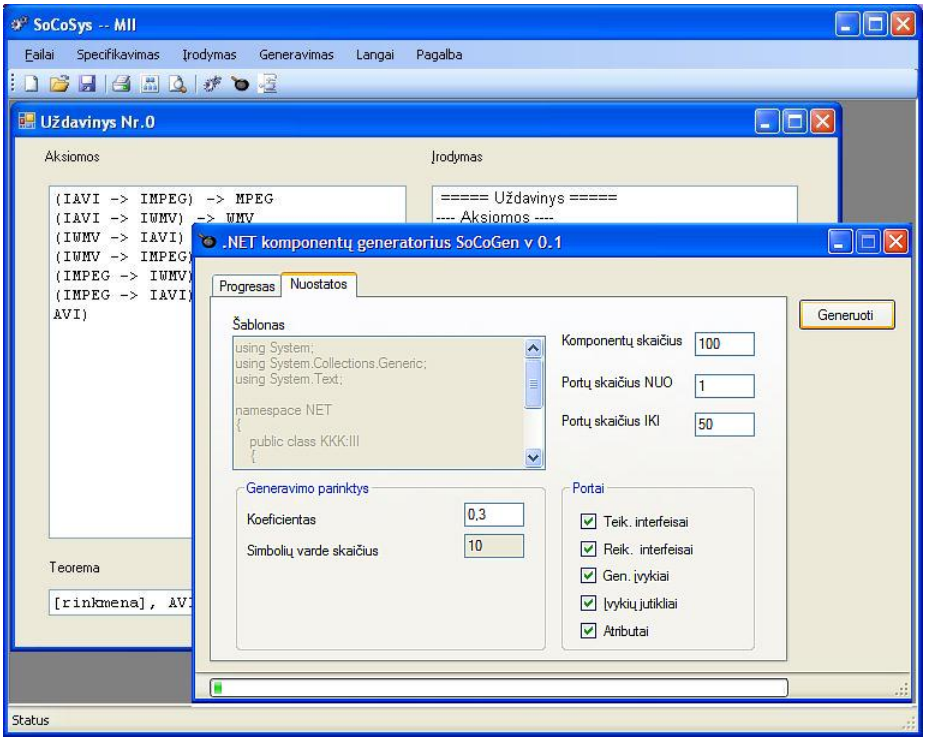
5.2. Sistemos eksperimentinis tyrimas

Siekiant patikrinti komponentinių programų sistemų generavimo metodą su jį realizuojančiu prototipu SoCoSyS buvo atlikti dviejų tipų eksperimentai:

- metodo principinio įgyvendinamumo ir tinkamumo komponentinėms programų sistemoms kurti eksperimentas,
- sistemos rezultatų patikimumo įvertinimo eksperimentas.

Atliekant pirmojo tipo eksperimentus siekta įvertinti ar iš tiesų konkretaus komponento modelio komponentus ir jų sistemas galima aprašyti abstraktaus programinio komponento modelio terminais, ar sintezės sistemos rezultatai atitinka prognozuotus rezultatus.

Sistemos eksperimentai atlikti sprendžiant komponentinių programų generavimo iš .NET komponentų uždavinius. .NET komponento modelis pasirinktas laisvai ir nedaro jokios įtakos eksperimento rezultatams.



5.6 pav. SoCoSyS ir .NET komponentų generatoriaus langai.

Dažnai pristatant sintezės metodus nagrinėjami elementarūs pavyzdžiai, sprendžiantys ne itin sudėtingus uždavinius. Siekiant įvertinti disertacijoje siūlomo metodo veiksmingumą sudėtingesnėms sistemoms kurti buvo atliktas toks eksperimentas (5.5 pav.):

1. Specialiu disertacijos autoriaus sukurtu įrankiu (5.6 pav.) automatizuotu būdu generuota .NET komponentų su atsitiktiniu prievadų skaičiumi, aibė. Kiekvienam komponentui generuota ir atitinkama PKM specifikacija (XML kalba).

2. SoCoSyS sistemai pateiktas generavimo uždavinys sudarytas iš aksiomų aibės ir teoremos, kurią reikia įrodyti. Aksiomų aibė sudaryta sistemai automatiškai analizuojant PKM komponentų specifikacijas esančias naudotojo nurodytame aplanke. Teorema įrodymui parinkta taip pat automatiniu būdu, atsitiktinai parenkant implikacijos kairiosios ir dešinės pusės narius. Pastebėtina, kad sistema leidžia ir pačiam naudotojui formuluoti teoremą. Šis procesas papildomai automatizuotas buvo tik eksperimento tikslais siekiant patikrinti kuo daugiau generavimo uždavinių.

Eksperimentas kartotas 50 kartų su įvairaus dydžio komponentų aibėmis, apimančiomis nuo 10 iki 200 .NET komponentų, taip pat kaitaliojant prievadų skaičių kiekviename komponente. Pirmo tipo eksperimentai parodė, kad komponentinės sistemos iš principo gali būti gaunamos taikant siūlomą metodą, tačiau jie dar neleidžia tvirtinti, jog gaunamas rezultatas yra prasmingas. Todėl siekiant įvertinti sistemos rezultatų patikimumą buvo atlikti antro tipo eksperimentai – sistema išbandyta su realiais .NET komponentais, kurie geba atlikti konkrečias užduotis. Tokiomis eksperimento sąlygomis lengva patikrinti ar gautoji programa iš tiesų veikia taip, kaip tikėtasi ir ar ji tikrai atlieka tą uždavinį, kuriam atlikti ir buvo sukurta. Iš viso šio tipo eksperimentai buvo atlikti su dvejomis .NET komponentų aibėmis:

- farmacijos sričiai skirtų komponentų aibe (žr. 4.5.1. poskyrį),
- multimedia medžiagos apdorojimo komponentų aibe (žr. 4.5.2. poskyrį).

Abiejų tipų testų rezultatai parodė, kad generavimo metodas, kurį realizuoja SoCoSyS, tinka komponentinėms programų sistemoms kurti.

5.3. Palyginimas su kitomis sistemomis

Šiame poskyriuje apžvelgiami ir su SoCoSyS sistema lyginami panašios paskirties mokslinių projektų rezultatai: NORA/HAMMR, QUASAR, *Component WorkBeanch* ir *CoSMIC* sistemos.

5.3.1. NORA/HAMMR sistema

NORA/HAMMR (*High adaptive multi-method retrieval*) programa buvo kurta kaip inžinieriaus darbo įrankis, siekiant inkapsuliuoti automatinę teoremų įrodymo programą, paslepiant jos vidinę specifikavimo kalbą [44, 152]. Sintezės uždavinys specifikuojamas VDM/SL kalba.
NORA/HAMMR ieško komponentų, tenkinančių silpnesnę „prieš“ salygą ir stipresnę „po“ salygą:

$$(pre_q \Rightarrow pre_c) \wedge (post_c \Rightarrow post_q)$$

NORA/HAMMR sudaro trys struktūriniai tiesinio vamzdyno architektūros moduliai:

1. signatūros atitikties (angl. *signature matchig*) filtras,
2. supaprastinimo modulis (angl. *simplifier*),
3. Kontrapavyzdžių generatorius (angl. *counter example generator*). Naudojamas *MACE* modelių generatorius, naudojamas parenkti tokioms kintamųjų reikšmėms su kuriomis formulė tampa neteisinga. Jis baigia darbą tik tuo atveju, jei uždavinio teorija yra baigtinė. Galimų kintamųjų reikšmių baigtumas emuliuojamas naudojant labiausiai tikėtinų kintamųjų reikšmių aibes [47, 152].

Yra numatyta filtruojančių modulių bazės plėtimo galimybė. Iš viso gali būti trijų tipų moduliai: sintaksės atitikties, atmetimo ir patvirtinimo. Pagrindinis filtrų uždavinys – atmesti klaidingas formules dar iki ATP panaudojimo.

Apdorotą filtrais užklausą toliau sprendžia ATP SETHEO [47]. Naudojamas standartinis *D. W. Loveland* 1978 m. aprašytas algoritmas. Teoremų įrodymo programai uždavinio sprendimui yra skiriama tik keletas sekundžių. Siekiant greičiau rasti rezultatą naudojami lygiagretieji algoritmai. Vienu metu tą patį uždavinį keliais būdais sprendžia kelios gijos tame pačiame, arba keliuose kompiuteriuose (jei dirbama kompiuterių telkinyje su *SciCoTHEO* ATP). Ta gija, kuri greičiausiai gražina rezultatą ir yra „laimėjusi“, visų kitų gijų vykdymas po to nutraukiamas. Kaip teigiama [152],taikant įvairius aksiomų atrankos algoritmus sėkmingai išspręstų uždavinių skaičius varijuoja nuo 55,9% iki 69,2%.

Sistema nėra specializuota komponentinėms programų sistemoms kurti. Komponentų atrankos komponentų saugykloje sudarytoje iš 119 komponentų uždavinys [152] nagrinėjamas tik siekiant pademonstruoti ATP panaudojimo galimybes [152]. Kaip pavyzdį *J. Schumann* [152] aprašo NORA/HAMMR programą, kuris skirtas generuoti uždavinį ATP siekiant atlikti paiešką komponentų bibliotekoje sudarytoje iš 119 komponentų.

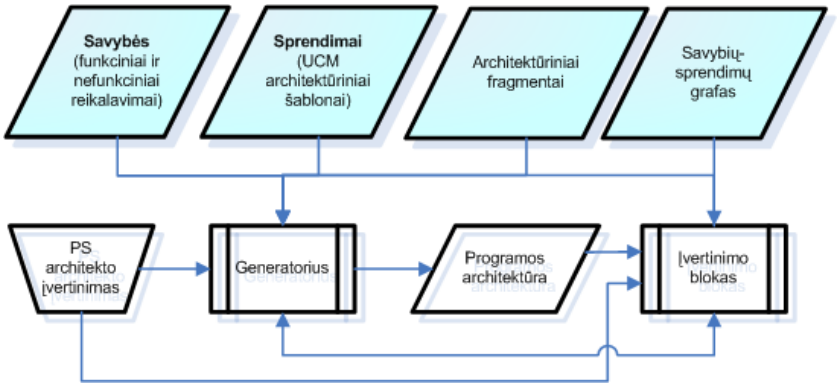
Sistemoje NORA/HAMMR naudojamas programų generavimo metodas yra panašus į SoCoSyS, nes abi sistemos naudoja automatinio įrodymo programą. Tačiau metodai turi ir skirtumų:

- SoCoSyS sistema yra specializuota būtent komponentinių programų sistemų generavimui;
- SoCoSyS sistemoje nekeliami griežti ribojimai uždavinio paieškos laikui;

5.3.2. QUASAR sistema

Vykdant projektą QUASAR (*QUAlity-driven Software Architecture*) [23, 70] pagrindinis dėmesys skirtas programinės įrangos kokybės užtikrinimo modeliams. taip pat tirtos ir komponentinių sistemų generavimo panaudojant tik jų funkcinius ir nefunkcinius reikalavimus galimybės.

QUASAR projekte nagrinėjamos konkrečioms produktų šeimoms priklausančios komponentinės programų sistemos [23, 70]. Pakartotinai panaudojami *pilkosios dėžės* (angl. *grey-box*) komponentai ir anksčiau kuriant kitas sistemas priimtų sprendimų fragmentai (architektūriniai ir projektavimo šablonai). Architektūriniai *sprendimų fragmentai* naudojami komponentams modifikuoti atsižvelgiant į *sprendimų-savybių* (angl. *Feature-Solution – FS*) grafą.
Sintezės sistemą sudaro:



5.7 pav. Programų architektūros generavimo sistema QUASAR.

- **Architektūrinio karkaso generatorius** remdamasis FS grafu užpildo UCM (angl. *Use Case Maps*) ruošinius konkrečiais fragmentais (angl. *snippets*). Darbe [23] tvirtinama, kad naudojama aspektiniam sistemų kūrimui būdinga metodika.

- **Įvertinimo blokas** (angl. *scenario-based evaluator*) tikrinantis ar generuota programų sistemos architektūra atitinka nefunkcinius reikalavimus. Tikrinama arba dalyvaujant ekspertui, arba naudojant *ontologinio matavimo* (angl. *Ontology-Driven Measurement – ODM*) metodą [70].

Lyginant SoCoSyS su QUASAR pastebimi šie skirtumai:

- QUASAR sistema yra specializuota darbui su komponentinių programų sistemomis priklausančiomis konkrečioms produktų šeimoms, operuojama tik su komponento specifikacijos lygmens komponentais. Tuo tarpu SoCoSyS leidžia operuoti tiek komponento specifikacijos, tiek komponento realiacijos lygmenyse.

- SoCoSyS sistemoje yra aukštesnis automatizavimo lygis, nes jos planavimo modulis, skirtingai nei QUASAR generatorius, veikia be žmogaus įsikišimo proceso metu.

- Naudojamas metodas lemia, kad SoCoSyS sistemos rezultatas yra *a priori* teisingas, tuo tarpu QUASAR sistemoje generatoriaus rezultatas dar tikrinamas įvertinimo bloke.

5.3.3. *Component WorkBeanch* sistema

J. Oberleitner ir *T. Gschwind* darbe [132] pristato automatizuotą komponentų programų kūrimo įrankį *Component WorkBeanch – CWB*. Pagrindinė problema, kuriai spręsti įrankis sukurtas – heterogeninių komponentų jungimas. Darbe [132] nagrinėjami trys komponento modeliai: DCOM, EJB ir CORBA. Anot autorių, nors teoriškai komponentinės programos kūrimui pakaktų sujungti atitinkamus interfeisus, praktiškai toks jungimas sunkiai įmanomas, dėl skirtingų duomenų tipų ir skirtingų realizacinių kiekvieno komponentinio modelio detalių. CWB aplinką sudaro:

1. vieningos prieigos prie komponentų lygmuo,

2. adaptavimo lygmuo,

3. scenarijų lygmuo,

4. įrankio lygmuo apimantis ir programos generatorių.

Vieningos prieigos prie komponentų lygmuo (angl. *uniform component access layer*) realizuotas per konkrečių komponento modelių pakiklius (angl. *wrapers*). Kiekvienas pakiklis atlieka keturias užduotis [132]:

- *Egzempliorių kūrimas.* Tarpininkas automatiškai inicializuoja komponentą, o jei tokio komponento egzempliorius jau yra jį susieja su sistemos kreipiniu.
- *Prieigos prie komponento savybių užtikrinimas.* Komponentinės programos architektui pateikia informaciją apie komponento savybes.
- *Vaizdavimas grafiškai.* Sukuria komponento vizualų atitikmenį CWB įrankio Systemos kūrimo lange.
- *Konfigūravimas.* Sudaro galimybes parametrizuoti komponentą.

Komponentų aibė, kurią naudoja CWB sistema, aprašyta XML kalba, kiekvienam komponentui nurodant jo modelį, dislokacijos vietą ir pan.

Adaptavimo lygmuo sudaro sąlygas komponentus jungti įvairiais būdais: komponavimo, agregavimo ir t.t. CWB naudojami adapteriai geba reaguoti į įvykius ir jos inicijuoti. Be to, yra sudarytos galimybės, šių adapterių programinį kodą redaguoti.

Scenarijų lygmenyje aprašoma komponentinės programų sistemos architektūra ir tokios komponentų savybės, kurios negali būti išsaugotos juose pačiuose, pavyzdžiui: pakeistas komponento pavadinimas ar sutartinis grafinis vaizdas.

CWB įrankyje numatyta galimybė vieną programos generatorių pakeisti kitu. Praktiniai bandymai atlikti naudojant Java ir XML kalbas. Pastaruoju atveju XML faile aprašyta komponentinės programos architektūra, kurią realizuoti reikėtų jau su kitu įrankiu [132].

Lyginant SoCoSyS ir *Component WorkBeanch* sistemas pastebimi šie skirtumai:

- CWB sistema leidžia kurti heterogenines komponentines sistemas, tačiau panaudojant tik tų modelių komponentus, kurie buvo numatyti ją kuriant, t.y. DCOM, EJB ir CCM. Tuo tarpu SoCoSyS sistemą, dėl naudojamo abstraktaus PKM modelio, galima taikyti platesniam komponento modelių ratui, bet negalima kurti heterogeninių sistemų.
- Skiriasi generavimo metodų taikymo tikslas ir taikymo metas. Automatiniu būdu CWB sistemoje generuojami tik pakikliai, o pačią komponentinių programų sistemą sukuria CWB naudojantis asmuo. Tuo tarpu SoCoSyS sistemoje generavimo metodas taikomas būtent galutinio rezultato gavimui.
- SoCoSyS sistemos rezultatas yra visiškai atitinkantis specifikaciją, tuo tarpu CWB sistema to negali garantuoti.

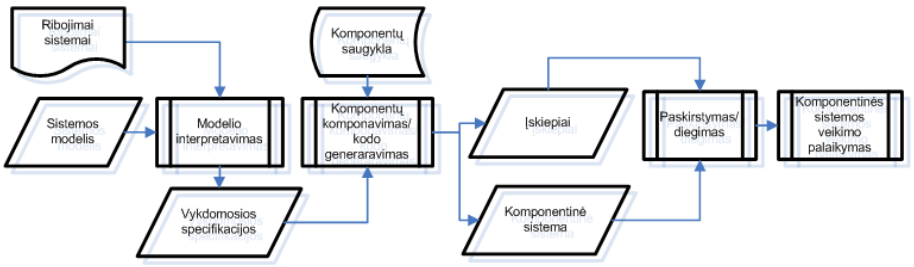
5.3.4. CoSMIC sistema

CoSMIC (angl. *Component Synthesis using Model Integrated Computing*) sistema skirta automatizuotu būdu atrinkti, sujungti ir paskirstyti išskirstytosios relaus laiko įterptinės sistemos komponentus [150].
Uždaviniai, kuriuos atlieka *CoSMIC* sistema, yra šie:

- programų sistemos teikiamų paslaugų konfigūravimas ir paskirstymas,
- komponentų komponavimas į sudėtinius programų serveriui skirtus komponentus ir jų realizacijų generavimas,
- komponentų konteinerių konfigūravimas,
- paslaugų kokybės užtikrinimo priemonių generavimas,
- tarpinės programinės įrangos konfigūracijų generavimas,
- tarpinės programinės įrangos ir jos įskiepių generavimas.

CoSMIC sistemoje realizuotos 4 pagrindinės dalys (5.8 pav.): modelio interpretatorius, kodo generatorius, paskirstymo (diegimo) įrankis ir programų serveris (angl. *application server*). Lyginant SoCoSyS ir *CoSMIC* sistemas matyti ypač dideli skirtumai:

- Skirtingai nei SoCoSyS, *CoSMIC* sistema yra specializuota, skirta kurti įterptinėms sistemoms.
- *CoSMIC* sistema operuoja visuose komponento abstrakcijos lygmenyse įskaitant ir *komponentinio objekto lygmenį*. *CoSMIC* yra integruota su programų serveriu aptarnaujančiu komponentinius objektus veikimo metu. SoCoSyS sistemoje operuojama tik komponento specifikacijos ir komponento realizacijos lygmenyse.



5.8 pav. Komponentinių programų sistemų generavimo sistema CoSMIC.

- *CoSMIC* sistema yra jautresnė komponentinių programų sistemos raidai, nes yra tampriai susieta su vienu (CMM pagrindu sukurtu CIAO) komponento modeliui. SoCoSyS sistema sukurta abstrahuojantis nuo konkretaus komponento modelio savybių, todėl yra lengviau adaptuojama.

5.4. Skyriaus išvados

1. Komponentinių programų sistemų generavimo metodo eksperimentinė realizacija SoCoSyS sistemoje parodė, kad metodas tinka tokioms programų sistemoms kurti.

2. SoCoSyS sistemą palyginus su kitomis sistemomis nustatyta:

   2.1. Skirtingai nei *NORA/HAMMR*, SoCoSyS sistema yra specializuota būtent komponentinių programų sistemų generavimui;

   2.2. Generavimo sistemos sukurtos siekiant skirtingų tikslų, todėl generavimo metodai, kuriuos jos realizuoja taip pat skiriasi. Iš analizuotų sistemų, tik SoCoSyS sistema realizuoja metodą, turintį kelių metodų elementų.

   2.3. SoCoSyS sistemoje komponentinių programų kūrimo proceso automatizavimo lygis yra aukštesnis, nei *QUASAR* ir *CWB* sistemose;

5.1 lentelė Komponentinių programų generavimo sistemos

	NORA/ HAMMR	QUASAR	CWB	CoSMIC	SoCoSyS
Naudojamas komponento modelis	–	pilkosios dėžės abstrakcijos	CCM, EJB, DCOM	CCM (CIAO)	PKM
komponento abstrakcijos lygmuo (-enys)	–	komponento specifikacijos (KS)	komponento realizacijos (KR), įdiegto komponento (ĮK), komponentinio objekto (KO)	KS, ĮK, KO	KS ir KR
Generavimo metodas	deduktyvioji sintezė	transformacinė sintezė	transformacinė sintezė	transformacinė sintezė	deduktyvioji sintezė pagal *Curry Howard* protokolą, transformacinė sintezė
Sintezės tikslas	atrinkti tinkamus komponentus.	susieti konkrečius reikalavimus su produktų šeimos architektūriniais šablonais.	sukurti pakiklius sudarant sąlygas projektuoti ir kūrti heterogeninių komponentų sistemas.	sukurti išskirstytąsias realaus laiko įterptines sistemas.	sukurti pasirinkto lygmens komponentinę sistemą.

Išvados

1. Komponento modelių įvairovė yra veiksnys, apsunkinantis automatizuotą komponentinių programų sistemų kūrimo procesą:

   1.1. Apžvelgus komponento modelius, reprezentuojančius žinomas modelių klases, nustatyta, kad jiems aprašyti iš viso naudojamos 28 sąvokos. Kiekvienas modelis aprašomas naudojant nuo 3 iki 12 sąvokų.

   1.2. Komponentinės programos gaunamos naudojant skirtingas komponavimo formas. Kurios būtent formos yra naudojamos, priklauso nuo komponento modelio ir jo abstrakcijos lygmens.

   1.3. Skiriami keturi komponento abstrakcijos lygmenys. Disertacijoje nagrinėjami *specifikacijos* ir *realizacijos* lygmenų komponentai. Įdiegto komponento ir komponentinio objekto lygmenys šioje disertacijoje nenagrinėjami, nes *įdiegto komponento* ir *komponentinio objekto* lygmenys glaudžiai susiję su sistemos veikimo etapu ir kūrimo procesui yra mažiau aktualūs.

2. Išanalizavus programinių komponentų savybes ir komponentinių programų kūrimo proceso ypatumus nustatyta:

   2.1. Pagrindiniai komponentų kūrimo proceso ypatumai yra šie: komponentai kuriami tam, kad būtų panaudoti daugiau nei vieną kartą; jie kuriami nežinant, kokie reikalavimai jiems bus keliami konkrečiose sistemose; komponentų specifikacija ir dokumentacija turi būti tiksli ir išsami.

   2.2. Komponentinių programų sistemų kūrimo procesas, kuriame akcentuojamas ne suprojektuotos sistemos ir jos dalių realizavimas, bet jau sukurtų komponentų pakartotinis panaudojimas turi trūkumų: neišspręstas detalios komponentų paieškos uždavinys; komponentų adaptavimui reikalingos papildomos sąnaudos.

3. Konkrečioms komponentinių programų sistemų surinkimo proceso automatizavimo problemoms spręsti gali būti naudojami struktūrinės sintezės, induktyviosios sintezės ir transformacinės sintezės metodai:

    3.1. Deduktyvieji metodai palaiko „juodosios dežės“ abstrakcijas, be to, taikant deduktyvųjį struktūrinės sintezės metodą užtikrinama rezultatų (taikomųjų programų) atitiktis specifikacijai.

    3.2. Induktyvusis metodas gali padėti spręsti šias deduktyviuoju metodu neišsprendžiamas problemas: specifikacijos neišsamumo problemą, neapibrėžtųjų komponentų problemą ir nefunkcinių reikalavimų problemą.

    3.3. Transformacinė sintezė įgalina mažinti atotrūkį tarp dalykinės srities, kuriai kuriama programinė įranga, ir realizacinės srities (konkrečių komponento modelių, karkasų, operacinių sistemų ir t. t.) konceptų.

4. Programų sintezės uždavinys ir jo sprendimo būdas gali būti taip suformuluotas apibendrinto *programinio komponento modelio* terminais, kad gautas sprendinys gali būti pritaikomas kiekvienam iš apibendrintų komponentų modelių panaudojant konkretizavimo, patikslinimo ir papildymo operacijas.

5. Programų sintezės metodas, komponentinėje paradigmoje realizuojantis *Curry-Howard* protokolą, įgalina automatizuotu būdu kurti komponentines programas ir užtikrina šių programų atitiktį specifikacijai.

6. Komponentinių programų sistemų generavimo metodo eksperimentinė realizacija SoCoSyS sistemoje parodė, kad metodas tinka tokioms programų sistemoms kurti.

Literatūra

[1] Abmann, U. (2003). *Invasive Software Composition*. Springer.

[2] Addibpour, M., Tyugu, E. (1996). Structural Synthesis of Programs from Refined User Requirements. *Formal methods for industrial applications*. (Abrial, J-R et al. eds.) LNCS 1165, Springer Verlag, p. 13–34.

[3] Addibpour, M. (1995). *Control Structures for Parallel Computing in NUT*. Technical Report TRITAIT R 95:18, Stockholm: Royal Institute of Technology (KTH).

[4] Aguirre, N., Maibaum, T. (2007). Temporal Specifications of Component Based Systems with Polimorphic Dynamic Reconfiguration. *Mathematical Frameworks for Component Software: Models for Analysis and Synthesis.* (Liu, Zh., Jifeng, H. eds.) World Scientific Publishing Company.

[5] Alti, A. et al. (2007). Integrating Software Architecture Concepts Into the MDA Platform with UML Profile. *Journal Of Computer Science*, vol. 3(10), p. 793–802.

[6] Amphion projektas. [interaktyvus]. [žiūrėta 2003 m. lapkričio 9 d.]. Prieiga per internetą: http://ti.arc.nasa.gov/ic/projects/amphion/

[7] Apperly, H. (2005). The Component Industry Metaphor. *Component Based Software Engineering: Putting the Pieces Together.* (Heineman, G. T., Councill, W. T. eds.) Addison-Wesley Professional. p. 21–32.

[8] The Archive of Formal Proofs. [interaktyvus]. [žiūrėta 2009 m. lapkričio 5 d.]. Prieiga per internetą: http://afp.sourceforge.net/

[9] Aspinall, D. (2000). Proof General: A Generic Tool for Proof Development. *Tools and Algorithms for the Construction and Analysis of Systems, Proc TACAS*, LNCS 1785, p. 38–43.

[10] Bachmann, F. et al. (2000). Volume II: Technical Concepts of Component-Based Software Engineering, 2nd Edition. Technical

Report CMU/SEI-2000-TR-008. Software Engineering Institute. [interaktyvus]. [žiūrėta 2010 m. sausio 14 d.]. Prieiga per internetą: http://www.sei.cmu.edu/publications/documents/00.reports/00tr008.html

[11] Backhouse, R. (2003). *Program construction: calculating implementatins from specifications*. Wiley.

[12] Barber, G. (2007) The Business Value Proposition of SCA. [interaktyvus]. [žiūrėta 2010 m. kovo 1 d.]. Prieiga per internetą: http://www.osoa.org/display/Main/The+Business+Value+Proposition+of+SCA

[13] Barnett, M., Schulte, W. (2002). *Contracts, components ir their runtime verificationon the .NET platform*. Tech. Report. Microsft Research.

[14] Barzdin', Y. M., Brazma, A. N., Kinber, E. B. (1987). Inductive synthesis of programs: State of the art, problems, prospects, *Cybernetics and Systems Analysis*, 23(6), p. 818-826. [interaktyvus]. [žiūrėta 2005 m. birželio 10 d.]. Prieiga per internetą: http://dx.doi.org/10.1007/BF01070244.

[15] Bass, L. et al. (2001). Market Assessment of Component-Based Software Engineering. [interaktyvus]. [žiūrėta 2009 m. gruodžio 20 d.]. Prieiga per internetą: http://www.sei.cmu.edu/library/abstracts/reports/01tn007.cfm

[16] Basili, V. R., Boehm, B. (2001). COST-Based Systems Top 10 List. *IEEE Computer* vol. 34(6), p. 91-95.

[17] Bazler, R. (1985). A 15 Year Perspective on Automatic Programming. *IEEE Transactions On Software Engineering* vol. SE-11 No. 11 , p. 1257-1268.

[18] Beisiegel, M. et. al. (2007). SCA Service Component Architecture: Assembly Model Specification. [interaktyvus]. [žiūrėta 2010 m. sausio 10 d.]. Prieiga per internetą: http://www.osoa.org/download/attachments/35/SCA_AssemblyModel_V100.pdf?version=1

[19] Berger, K. et al. (2000). *A Formal Model for Componentware. Foundations of Component-Based Systems*. Cambridge University Press.

[20] Bertot, Y. et al. (2004) *Interactive Theorem Proving and Program Development*. Springer.

[21] Binkley, D. at al. (2005). Automated Refactoring of Object Oriented Code into Aspects. *In Proceedings of 21st IEEE International Conference on Software Maintenance ICSM'05*, p. 27–36.

[22] Boyer, R. S., Moore, J. S. (1990). A theorem prover for a computational logic. *Proceedings of the Tenth international Conference on Automated Deduction* Springer-Verlag, New York, p. 1-15.

[23] Bruin, H., Vliet H. (2002). The Future of Component-Based Development is Generation, not Retrieval. *Proceedings ECBS'02 Workshop on CBSE – Composing Systems from Components (Crnkovic, I., Larsson, S., Stafford, J. eds.)*, Lund. [interaktyvus]. [žiūrėta 2004 m. liepos 8 d.]. Prieiga per internetą: http://www.cs.vu.nl/∼hans/publications/y2002/gc.pdf

[24] Bruneton, E. et al. (2006). The Fractal Component Model and Its Support in Java. *Software Practice and Experience, special issue on Experiences with Auto-adaptive and Reconfigurable Systems. 36(11-12)*, p. 1257-1284.

[25] Boyle J.M., Harmer T.J., Winter V.W. (1996). The TAMPR Program Transformation System Proceedings of the Durham Transformation Workshop.

[26] Bundy, A. et al. (2005). *Rippling: Meta-Level Guidance for Mathematical Reasoning*. Cambridge University Press.

[27] Cai, J., Paige, R. (1990). *The RAPTS Transformation System*. New York:New York University.

[28] Cervantes, H., Hall, R.S. (2004). A Framework for Constructing Adaptive Component-based Applications: Concepts and Experiences. In *7th Symposium on Component-Based Software Engineering - CBSE*, LNCS vol. 3054, Edinburgh:Springer, p. 130-137.

[29] Chan, K., Poernomo, I. (2007). QoS-aware model driven architecture through the UML and CIM. *Frontiers of Information Systems. vol.9*, p. 209-224.

[30] Charpentier, M. (2002). An Approach to Composition Motivated by *wp*. In *5th International Conference on Fundamental Approaches to Software Engineering (FASE2002)*, LNCS 2306, Springer-Verlag, p. 1–14.

[31] Charpentier, M. (2006). Composing Invariants. *Science of Computer Programming.*, 60, p. 221–243.

[32] Charpentier, M., Chandy, K. M. (2000). Theorems about composition. *International Conference on Mathematics of Program Construction (MPC2000)*, LNCS 1837, Springer-Verlag, p. 167–186.

[33] Charpentier, M., Chandy, K. M. (2004). Specification transformers: a predicate transformer approach to composition. *Acta Informatica.* vol. 40(4), p. 265–301

[34] Cheesman, J., Daniels, J. (2001). *UML Components.* Addison-Wesley.

[35] Chen, X. et. al. (2006). A Model of Component-Based Programming. Technical Report 350, Macau:UNU/IIST.

[36] Chen, Z. et. al. (2007). Refinement and Verification in Component-Based Model Driven Design. Technical Report 388, Macau:UNU/IIST.

[37] Christiansson, B., Jakobsson, L., Crnkovic, I. (2003) CBD Process. *Building Realiable Component-Based Software Systems.* (Crnkovic, I., Larsson, M. eds.) p. 89–113.

[38] Chung, L., Nixon, B. A., Yu, E., Mylopoulos, J. (2000). *Non-Functional Requirements in Software Engineering.* Kluwer Academic Publishers.

[39] Cleaveland, J. C. (2001). *Program Generators with Java and XML.* Prentice Hall PTR.

[40] Communicating Sequential Processes. The First 25 Years. (2005) Abdallah, Ali E.; Jones, Cliff B.; Sanders, Jeff W. (Eds.) Springer.

[41] Cox, P. T., Song, B. (2001). A formal model for component-based software. *Proc. of the IEEE Symposium on Visual/Multimedia Approaches to Programming ir Software Engineering*, Stresa, Italy, p. 304–311.

[42] Crnkovic, I. et. al. (2005). Automated Component-Based Software Engineering. *The Journal of Systems and Software* 74, p. 1–3.

[43] Czarnecki, K. (2000). *Generative programming - Methods, Tools, and Applications.* Addison-Wesley.

[44] Denney, E., Fischer, B. (2005). *Selected References for SBMF Mini-course on Automated Code Generation.* Porto Alegre, Rio Grande do Sul, Brasil.

[45] Deville, Y., Lau K.-K. (1993). Logic program synthesis. *The Journal of logic programming*, vol. 12, p. 1–199.

[46] D'Souza, D. F., Wills, A. C. (1999). *Objects, Components and Frameworks with UML: The Catalysis Approach.* Addison Wesley Longman. [interaktyvus]. [žiūrėta 2007 m. balandžio 27 d.]. Prieiga per internetą: http://www.catalysis.org/books/ocf/index.htm

[47] Fischer, B. (2001). *Deduction-Based Software Component Retrieval.* PhD-thesis, Universität Passau.

[48] Dijkstra, E. W. (1976). *Discipline of Programming.* Prentice Hall.

[49] Dijkstra, E. W., Scholten, C. S. (1990). Predicate calculus and program semantics. Springer-Verlag.

[50] Endres, A., Rombach, D. (2003). *A Handbook of Software and Systems Engineering: Empirical Observations, Laws and Theories.* Addison Wesley.

[51] Fagorzi, S., Zucca, E. (2007). A Calculus of Components with Dynamic Type-Checking. Proceedings of FACS 2006 (3rd International Workshop on Formal Aspects of Component Software). Electronic Notes in Theoretical Computer Science. Vol. 182, p. 73–90.

[52] Fabresse, L. et al. (2008). Foundations of a simple and unified component-oriented language. *Computer Languages, Systems & Structures*, vol. 34, Iss. 2-3, p. 130–149.

[53] Feather, M. S. (1987). A Survey and Classification of some Program Transformation Approaches and Techniques. *Program Specification and Transformation.* Elsevier.

[54] Ferilli, S., Esposito, F., Basile, T.M.A., Di Mauro, N. (2004). In *Automatic Induction of First-Order Logic Descriptors Type Domains from Observations.* LNCS 3194, Springer, p. 116–131.

[55] Flener, P. (2002). Achievements and prospects of program Synthesis. *Computational Logic: Logic Programming and Beyond: Essays in Honour of Robert A. Kowalski*, LNAI 2407., Springer-Verlag, p. 310–346.

[56] Fractal Component Model Specifications. [interaktyvus]. [žiūrėta 2009 m. lapkričio 9 d.]. Prieiga per internetą: http://fractal.objectweb.org/

[57] Gallier, J. H. (2003). Foundations of Automatic Theorem Proving. Wiley.

[58] Gao, J. Z., Tsao H.-S. J., Wu, Y. (2003). Testing and Quality Assurance for Component-based Software. Artech House.

[59] Generative Modeling Technologies (GMT) project. [interaktyvus]. [žiūrėta 2008 m. vasario 3 d.]. Prieiga per internetą: http://www.eclipse.org/gmt/

[60] Generative Tools. [interaktyvus]. [žiūrėta 2008 m. spalio 3 d.]. Prieiga per internetą: http://www.program-transformation.org/Transform/GenerativeTools

[61] Genssler, T. et al. (2002). *PECOS in a Nutshell.* [interaktyvus]. [žiūrėta 2003 m. vasario 19 d.]. Prieiga per internetą: http://www.pecos-project.org/public_documents/pecosHandbook.pdf

[62] Giedrimas, V. (2003). komponento modelis struktūrinės programų sintezės kontekste. *Informacijos mokslai* 26: p. 246–250.

[63] Giedrimas, V., Lupeikienė, A. (2004). komponento specifikacijos formalizavimas. *Lietuvos Matematikos Rinkinys. Spec. Nr.* 44 p. 276–280.

[64] Giedrimas, V., Lupeikienė, A. (2005). komponentinių programų struktūrinės sintezės teorinės problemos. *Lietuvos Matematikos Rinkinys. Spec. Nr.* 45, p. 139–143.

[65] Giedrimas, V. (2006). Architectures of Component-based Structural Synthesis Systems. *Proceedings of Baltic DBIS06*, p. 311–315.

[66] Giedrimas, V. (2007). Generavimo metodų panaudojimas kuriant .NET komponentines programų sistemas. *Informacijos mokslai*, T. 42–43, p. 246–250.

[67] Gobel, S. (2005). An Mda Approach For Adaptable Components. In *Model Driven Architecture - Foundations And Applications: First European Conference, Ecmda-Fa 2005 Proceedings*, p. 74–87.

[68] Gordon, M. (2008). Twenty Years of Theorem Proving for HOLs Past, Present and Future. *Theorem Proving in Higher Order Logics. Proceedings of 21st International Conference.* Springer, p. 1–5.

[69] Grigorenko, P., Saabas, A., Tyugu, E. (2005). Visual Tool for Generative Programming. *Proc. of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ACM Press.

[70] Goulao, M.C.P.A. (2008). Component-Based Software Engineering: a Quantitative Approach. PhD thesis. Universidade Nova de Lisboa.

[71] Guessoum, A., Komorowski. J. (1995). Induction, Deduction and Abduction for Program Design and Maintenance.

[72] Gupta, N. S. (2004). *An Exploratory Analysis of the .NET Component Model and UniFrame paradigm using a collaborative approach.* M. S. Thesis, Indiana University Purdue University.

[73] Gupta, N. S. et al. (2003). Analyzing the Web Services and UniFrame Paradigms. *The Proceedings of SESEC 2003, the Southeastern Software Engineering Conference*, Huntsville.

[74] Girard, J.Y. (1989). *Proofs and types.* Cambridge University Press.

[75] Harf, M. et al. (2001). Automated Program Synthesis for Java Programming Language. *Ershov Memorial Conference.* p. 157–164.

[76] Harrop, R. (1960). Concerning Formulas of the Types $A \to B \vee C$, $A \to (Ex)B(x)$ in Intuitionistic Formal Systems. *Journal of Symbolic Logic 25*, no. 1, p. 27–32.

[77] He, J., Li, X., Liu, Z. (2006). rCOS: A Refinement Calculus of Object systems. *Theretical Computer Science 365* p. 109–142.

[78] He, J., Liu, Z., Li, X. (2003). *Component Calculus*. Technical Report 285, Macau:UNU/IIST.

[79] He, J., Li, X., Liu, Z. (2005). *Component-Based Software Engineering – the Need to Link Methods and their Theories*. Technical Report 330, Macau:UNU/IIST.

[80] Hehner, E. (2006). Retrospective and prospetive for Unifying Theories of programming. *Unifying Theories of Programming, First International Symposium, UTP 2006, Revised Selected Papers*. LNCS 4010, Springer, p. 1–17.

[81] Herrington, J. (2003). *Code Generation in Action*. Manning.

[82] Ho, S-M., Lau K-K. (2007). Characterizing object-based frameworks in First-order predicate logic. *Mathematical Frameworks for Component Software: Models for Analysis and Synthesis*. (Liu, Zh., Jifeng, H. eds.) World Scientific Publishing Company

[83] Hoare, C.A.R, He, J. (1998). *Unifying Theories of Programming. Prentice Hall Series in Computer Science*. Prentice Hall.

[84] Howard, W.A. (1980) The formulae-as-types notion of constructions. *To H.B.Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic press, p. 479–490.

[85] Huet, G., Hullot, J. M. (1980). Proofs by Induction in Equational Theories with Constructors. *Foundations of Computer Science*, p. 96–107.

[86] Huet, G. P. (1986). Theorem Proving Systems of the Formel Project. *Proceedings of the 8th international Conference on Automated Deduction*. LNCS 230, Springer-Verlag, p. 687–688.

[87] ISO/IEC TR 9126-3:2003 Software Engineering – Product Quality – Part 3: Internal metrics. (2003)

[88] ISO/IEC 25051:2006 Software Engineering – Software product Quality Requirements and Evaluation (SQuaRE) – Requirements for quality of Commercial Off-The-Shelf (COTS) software product and instructions for testing. (2006)

[89] Institute of Cybernetics. *The NUT language*. [interaktyvus]. [žiūrėta 2002 m. spalio 10 d.]. Prieiga per internetą: http://cs.ioc.ee/~nut/

[90] Jablonskij, S. V. (1986). *Vviedienijie v diskretnuju matematiku*. Moskva:Mir (rusų k.)

[91] Kabir, M. H. (2007). *Automatic inductive theorem proving and program construction methods using program transformation*. PhD. thesis, Dublin City University.

[92] Kaisler, S. H. (2005). *Software paradigms*. Wiley-interscience.

[93] Kant, E., Barstow, D. (1981). The refinement paradigm: the interaction of coding and efficiency knowledge in program synthesis. *IEEE Transactions on Software Engineering*, vol. 7, no. 5 p. 458-471.

[94] Kleppe, A. (2003). *Mda Explained: The Model Driven Architecture: Practice And Promise*. Boston: Addison-Wesley.

[95] Kum, D. K., Kim, S. D. (2006). A Systematic Method To Generate .Net Components From Mda/Psm For Pervasive Service., *Fourth International Conference On Software Engineering Research, Management And Applications*. p. 324–331.

[96] Küster, J., Bowles, F. ir Moschoyiannis, S. (2007). Concurrent Logic and Automata Combined: a Semantics for Components. *In Proc. of CONCUR 2006 - Foundations of Coordination Languages and Software Architectures (FOCLASA'06)*, ENTCS, 175(2): 135–151, Elsevier.

[97] Lau, K.-K., Momigliano, A., Ornaghi, M. (2005). Constructive Specification of Compositional Units. *Proceedings of the Fourteenth International Workshop on Logic-based Program Synthesis and Transformation*. LNCS 3573, Springer-Verlag, p. 198–214.

[98] Lau, K.-K. (2003). Component-based Software Development and Logic Programming. *Proceedings of the Nineteenth International Conference on Logic Programming*. LNCS 2926, Springer-Verlag, p. 103–108.

[99] Lau, K.-K., Ornaghi, M. (2001) A Formal Approach to Software Component Specification. Proceedings of Specification and Verification of Component-based Systems Workshop at OOPSLA2001. [interaktyvus]. [žiūrėta 2006 m. kovo 9 d.]. Prieiga per internetą: http://citeseer.ist.psu.edu/592159.html

[100] Lau, K.-K., Wang, Z. (2005). A Taxonomy of Software Component Models. *Proceedings of the 31st EUROMICRO Conference on Software Engineering and Advanced Applications* p. 88–95.

[101] Lau, K.-K., Wang, Z. (2007). Software Component Models. *IEEE Transactions on Software Engineering, vol. 33 - 10*, p. 709–724

[102] Lammermann, S. (2002). Runtime Service composition via logic-based program synthesis. PhD. thesis, KTH, Stockholm. TRITA-IT AVH 02:03 ISSN 1403-5286

[103] Lammermann, S., Tyugu., E. (2002) Implementing Extended Structural Synthesis of Programs. *In Proc. AAAI 2002 Spring Symposium Series on Logic-Based Program Synthesis: State of the Art and Future Trends.* AAAI Press, p. 63–71.

[104] Leavens, G.T., Dhara, K.K. (2000). Concepts of Behavioral Subtyping & Foundations of Component-Based Systems. Cambridge University Press

[105] van Lint, J.H., Wilson, R.M. (2002). *A Course in Combinatorics.* Cambridge University Press.

[106] López-Sanz, M. et al. (2008). Modelling Of Service-Oriented Architectures With Uml. ENTCS 194(4), p. 23–37.

[107] Luders, F., Lau, K.-K.,Ho S.-M. (2003). Specification of software components. *CBD Process.Building Realiable Component-Based Software Systems.* Eds. Crnkovic.,I., Larsson., M. p. 23–38.

[108] Lupeikiene, A., Giedrimas, V. (2005) Component model and its formalisation for structural synthesis. *Scientific Proc. of Riga Technical University. Computer Science*, vol. 22, p. 169–180.

[109] Lupeikiene, A. (2006) Integrated Enterprise Information System Development through Component Abstraction. *Proceedings of the Seventh International Baltic Conference on Databases and Information Systems*, (Vasilecas, O., Eder, J., Čaplinskas, A. eds.), Institute of Electrical and Electronics Engineers, p. 168–174.

[110] Lupeikienė, A. (2007) *Teoriniai ir technologiniai informacinių sistemų aspektai.* Vilnius.

[111] Manna, Z., Walldinger, R. (1980). A Deductive Aproach to Program Synthesis. *Transactions of programming Languages and Systems* vol. 2, p. 90–121.

[112] Martelli, M., Mascardi, V., Zini F. (1999). A Logic Programming Framework for Component-Based Software Prototyping. *Proceeding of 2nd International Workshop on Component-based Software Development in Computational Logic (COCL' 99).*

[113] Matskin, M., Tyugu, E. (1997). Strategies of Structural Synthesis of Programs. *Automated Software Engineering Conference*. p. 305–306.

[114] Matskin, M., Rao, J. (2002). Value-Added Web Services Composition Using Automatic Program Synthesis. *CAiSE '02/ WES '02: Revised Papers from the International Workshop on Web Services*. Springer-Verlag, p. 213–224.

[115] McIlroy, M. D. (1968). Mass produced software components. *Proc. Nato Software Eng. Conf.*. Garmisch, p. 138-155.

[116] Medvidovic, N., Taylor, R. N. (2000). A Classification and Comparision Framework for Software Architecture Description Languages. In *IEEE Transactions on Software Engineering*. vol. 26, p. 70–93.

[117] Mellor, S.J., Balcer M. J. (2002). Executable UML: A Foundation for Model-Driven Architecture. Addison-Wesley.

[118] Mellor, J. et al. (2004). MDA Distilled: Principles of Model-driven Architecture. Addison-Wesley.

[119] Meng S., Aicherning B.K. (2002). Component-Based Coalgebraic Specification and Verification in RSL. Technical Report 267, Macau:UNU/IIST.

[120] Meng S., Aicherning B.K. (2003). A Coalgebraic Calculus for Component Based Systems. Proceedings of the Workshop on Formal Aspects of Component Software FACS'03, H. H.D. Van , Z. Liu, eds., Pisa, Italy, pages 27–46.

[121] Meyer, B. (2003). The Grand Challenge of Trusted Components. In *ICSE 25 (International Conference on Software Engineering, Portland, Oregon, May*, IEEE Computer Press.

[122] Microsoft. (1996). *The component object model specification.* Report v0.99, Microsoft Standarts, Redmond WA:Microsoft Press.

[123] Moschoyiannis, S., Shields. M. W.(2003). Component-based design: towards guided composition. *Proceedings of Application of Concurrency to System Design (ACSD'03)*, Guimaraes:IEEE Computer Society, p. 122–131.

[124] Muggleton, S., Feng C. (1990). Efficient Induction Of Logic Programs. *Proceedings of the 1st Conference on Algorithmic Learning Theory* p. 368–381.

[125] Muggleton, S.,De Raedt, L. (1994). Inductive logic programming : Theory and methods. *Journal of Logic Programming*, vol. 19-20, p. 629-679.

[126] Muller, P., Poetzsch-Heffer, A. (2000). Modular specification and Verification Techniques for Object-oriented software composition. *Foundations of Component-Based Systems*. Cambridge University Press.

[127] Newborn, M. (2001). *Automated Theorem Proving: Theory and Practice*. Springer.

[128] Nierstrasz, O. (1995). Object-oriented Software Composition. Prentice Hall. Prieiga per internetą: [interaktyvus]. [žiūrėta 2008 m. kovo 9 d.]. http://www.iam.unibe.ch/∼scg/Archive/nie95/PDF/Nier95bnie95book.pdf

[129] Nierstrasz, O. et al. (2002). A Component Model for Field Devices. *Proceedings First International IFIP/ACM Working Conference on Component Deployment*, ACM, Berlin.

[130] Nipkow, T., Paulson, L. C., Wenzel, M. (2002). *Isabelle/HOL - A Proof Assistant for Higher-order Logic*. Springer.

[131] Norgėla, S. (2007). *Logika ir dirbtinis intelektas*. V.:TEV.

[132] Oberleitner, J., Gschwind, T. (2003). Composing distributed components with the Component Workbench. *SEM : Software engineering and middleware*, LNCS 2596. Springer, p. 102–114.

[133] Ohori, A. (1999). *A Curry-Howard Isomorphism for Compilation and Program Execution*. LNCS 1581, Springer.

[134] OMG (1998). CORBA components. Report ORBOS/99-02-01, Object Management Group [interaktyvus]. [žiūrėta 2009 m. gruodžio 1 d.]. Prieiga per internetą: http://www.omg.org

[135] OpenCCM - The Open CORBA Component Model Platform. [interaktyvus]. [žiūrėta 2009 m. gruodžio 1 d.]. Prieiga per internetą: http://openccm.objectweb.org/

[136] OMG. Unified modelling language Specification (Action Semantics). Version 2.0 [interaktyvus]. [žiūrėta 2009 m. balandžio 10 d.]. Prieiga per internetą: http://www.omg.org/cgi-bin/doc.cgi?ptc/02-01-09.pdf, [žiūrėta 2009 m. gegužės 2 d.].

[137] OMG. Meta Object Facility (MOF) Core Specification. Version 2.0 [interaktyvus]. [žiūrėta 2009 m. balandžio 10 d.]. Prieiga per internetą: http://www.omg.org/spec/MOF/2.0

[138] Pastor, O., Molina, J. C. (2007). *Model-Driven Architecture in Practice*. Springer.

[139] Paulson, L. C. (2005). *The Isabelle/Isar Reference Manual.* [interaktyvus]. [žiūrėta 2008 m. rugpjūčio 15 d.]. Prieiga per internetą: http://isabelle.in.tum.de/dist/Isabelle/doc/isar-ref.pdf

[140] Penjam, J. , Tyugu, E. (1993). Constraint Semantics of NUT. Technical Report TRITA-IT-9309 IOC-CS-58/93, CSLab, Dept. of Teleinformatics, Stockholm:Royal Institute of Technology (KTH).

[141] Poernomo, I. H., Reussner, R., Schmidt, H. W. (2002). Architectures of Enterprise Systems: Modelling Transactional Contexts. *Component Deployement* LNSC 2370, p. 233–243

[142] Poernomo, I. H., Crossley, J. N., Wirsing, M. (2005). *Adapting Proofs-as-Programs: The Curry–Howard Protocol.* Springer.

[143] Program Generator Projects and Activities. [interaktyvus]. [žiūrėta 2009 m. balandžio 10 d.]. Prieiga per internetą: http://www.craigc.com/pg/projects.html

[144] Raje, R. (2000). UMM: Unified Meta-Object Model For Open Distributed Systems. *Proceedings of 4th IEEE International Conference on Algorithms and Architecture for Parallel Processing, ICA3PP'2000*, Hongkong, p. 454–465

[145] Rao, J. (2004). *Semantic Web Service Composition via Logic-based Program Synthesis.* PhD Thesis. Department of Computer and Information Science, Norwegian University of Science and Technology.

[146] Reussner, R.H., Schmidt, H.W., Poernomo, I.H. (2003). Reliability prediction for component-based software architectures. *Journal if Systems and Software* vol. 66, 3 p. 241–252.

[147] Sametinger, J. (1997). Software Engineering with Reusable Components. Springer-Verlag.

[148] Salzmann, C. (2002). Invariants of Component Reconfiguration. *Proceedings of Seventh International Workshop on Component-Oriented Programming.* [interaktyvus]. [žiūrėta 2008 m. spalio 20 d.]. Prieiga per internetą: http://research.microsoft.com/∼cszypers/events/WCOP2002/12_Salzmann.pdf

[149] Schneider, J.G. (1999). *Components, Scripts, ir Glue: A conceptual framework for software composition* PhD thesis. [interaktyvus]. [žiūrėta 2003 m. spalio 15 d.]. Prieiga per internetą: http://www.iam.unibe.ch/∼scg/Archive/PhD/schneider-phd.pdf

[150] Schmidt, D.C. et al. (2002). CoSMIC: an MDA Generative tool for Distributed real-time and Embedded Component Middleware and Applications. *Proceedings of the OOPSLA 2002 Workshop on Generative Techniques in the Context of Model Driven Architecture*, ACM.

[151] Schmidt, D.C. (2009). Real-time CCM with CIAO (Component Integrated ACE ORB). [interaktyvus]. [žiūrėta 2009 m. lapkričio 3 d.]. Prieiga per internetą: http://www.cs.wustl.edu/∼schmidt/CIAO.html

[152] Schumann, J.M. (2001). *Automated Theorem Proving in Software Engineering*. Springer

[153] Smith, D. R. (1985). Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence*, vol. 27, no. 1, p. 43 - 96.

[154] Stojanovič, Z. (2005). *A Method For Component-Based And Service-Oriented Software Systems Engineering*. Phd Thesis. Delft University Of Technology.

[155] Sommerville, I. (2006). *Software Engineering. 8th Ed.* Addison Wesley.

[156] Stickel, E. (2008) SNARK - SRI's New Automated Reasoning Kit. [interaktyvus]. [žiūrėta 2009 m. gruodžio 11 d.]. Prieiga per internetą: http://www.ai.sri.com/∼stickel/snark.html

[157] Sun Microsystems. JavaBeans 1.01 specifikacija. [interaktyvus]. [žiūrėta 2009 m. sausio 10 d.]. http://java.sun.com/beans

[158] Szyperski, C. (2000). *Component Software and the Way Ahead. Foundations of Component-Based Systems*. 2nd Ed. Cambridge University Press.

[159] Szyperski, C. (2000). *Component Software Beyound Object-Oriented Programming*. 2nd Ed. MA:Addison-Wesley.

[160] Štuikys, V., Damaševičius, R. (2008). *Modelių ir programų abstrakčiosios transformacijos*. Kaunas:Technologija.

[161] Teschke T., Ritter J. (2001). Towards a foundation of component-oriented software reference models. *Lecture Notes in Computer Science 2177*, Springer. p. 70–85.

[162] Tilak, O.J., Raje, R. R. (2007). Temporal Interaction Contracts for Components in a Distributed System. *11th IEEE International Enterprise Distributed Object Computing Conference*, IEEE press, p. 339–349.

[163] Thai, P.H. Hung, D.V. (2007). Towards a Template Language for Component-based Programming. Technical Report 354, Macau:UNU/IIST.

[164] Tourwé, T., Brichau, J., Kellens, A., Gybels K. (2004). Induced intentional software views Computer Languages, *Systems & Structures*, 30(1-2), p. 35–47

[165] Tracz., W. (2005). COTS Myths and Other Lessons Learned in component-based sofware development. *Component Based Software Engineering: Putting the Pieces Together*. (Heineman, G. T., Councill, W. T. eds.) Addison-Wesley Professional.

[166] Tyugu, E., Harf, M. (1985). Algoritmy strukturnovo sinteza program. *Programirovanie*. 4, p. 3–13. (rusų k.).

[167] Tyugu, E. (1988). *Knowleadge based programming*. Turing Institute Press.

[168] Tyugu, E. (1985). *Konceptualnojie programmirovanijie*. Moskva:Mir. (rusų k.).

[169] Tyugu, E. (1975). A programming system with automatic program synthesis. LNCS 47, p. 251–267.

[170] Tyugu, E. (1991). Three new-generation software environments. *Comm. ACM*, vol.34, No.6, p. 46–59.

[171] Tyugu, E. (1994). Using Classes as Specifications for Automatic Construction of Programs in the NUT System. *Journal of Automated Software Engineering*.

[172] Tyugu, E. (1982). Vyčislitielnyje freimy i strukturnyj sintezis programm. *Techničeskaja kibernetika* Nr. 6. (rusų k.)

[173] UniFrame – Unified Meta-component Model for Distributed Systems. [interaktyvus]. [žiūrėta 2007 m. balandžio 8 d.]. Prieiga per internetą: http://www.cs.iupui.edu/uniFrame

[174] United Nations University, International Institute of Software Technology. [interaktyvus]. [žiūrėta 2007 m. balandžio 8 d.]. Prieiga per internetą: http://www.iist.unu.edu/

[175] Uustalu, T. et al. (1994). The NUT Language Report. Technical Report TRITA-IT-R 94:14, Stockholm:Royal Institute of Technology.

[176] Volož, B. et al. (1982). Sistema PRIZ i isčislienijie vyskazyvanii. *Kibernetika* Vol. 6, p. 63–70. (rusų k.)

[177] Volož, B., Kopp, M., Tyugu., E. (1993). *The NUT Graphics*. Technical Report TRITA-IT-R 93:05, Stockholm:Royal Institute of Technology.

[178] (2006). *Knowledge-Based Software Engineering*. E.Tyugu, T.Yamaguchi (Eds.) IOS Press.

[179] Voronkov, A. (2002). First-Order Theorem Provers: the Next Generation. *Proceedings of the 2002 International Workshop on Description Logics (DL2002)*. [interaktyvus]. [žiūrėta 2007 m. spalio 13 d.]. Prieiga per internetą: http://ftp.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-53/voronkov.ps

[180] Walldinger, R., Jarvis, P., Dungan, J. (2004) Program Synthesis for Multi-agent Question Answering. *Verification: Theory and Practice* LNCS 2772 p. 443–458.

[181] Wallnau, K. C., Hissam S. A., Seacord R.C. (2001). *Building System from Commercial Components*. Addison-Wesley.

[182] Watson G.N. (1998). Proof representation in theorem provers. *Technical report No 98-13*. University of Queensland. [interaktyvus]. [žiūrėta 2008 m. liepos 1 d.]. Prieiga per internetą: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.35.9242

[183] Web Services Activity. [interaktyvus]. [žiūrėta 2006 m. balandžio 18 d.]. Prieiga per internetą: http://www.w3.org/2002/ws/

[184] Web Services Description Language (WSDL) Version 1.2 [interaktyvus]. [žiūrėta 2006 m. balandžio 18 d.]. Prieiga per internetą: http://www.w3.org/TR/2003/WD-wsdl12-20030303/wsdl12.pdf

[185] Wenzel, M., Berghofer, S. (2005). The Isabelle System Manual. [interaktyvus]. [žiūrėta 2008 m. rugpjūčio 15 d.]. Prieiga per internetą: http://isabelle.in.tum.de/dist/Isabelle/doc/system.pdf

[186] Weerawarana, S. et al. (2001). Bean Markup Language: A Composition Language for JavaBeans Components. In *Proceedings of the 6th conference on USENIX Conference on Object-Oriented Technologies and Systems vol. 6* p. 13–13.

[187] Weinreich, R., Sametinger, J. (2001). Component Models and Component Services: Concepts and Principles. *Component Based Software Engineering: Putting the Pieces Together*. (Heineman, G. T., Councill, W. T. eds.) Addison-Wesley Professional. p. 21–32.

[188] Whitehead, K. (2002). Component-based Development :Principles and Planing for Business Systems. Addison-Wesley.

[189] Xiao, L. (2009). An adaptive security model using agent-oriented MDA. Inf. Softw. Technol., vol. 51, no. 5, p. 933–955.

[190] Yoshida, K., Honiden S. (2003). Formal specification of components in a component-based framework development method. *Systems and Computers in Japan* vol. 34, Iss. 8, p. 62–76.

A priedas

Publikacijų sąrašas

Straipsniai recenzuojamuose Lietuvos žurnaluose

1. **Giedrimas, V.** (2009). Modelinės architektūros naudojimas kuriant komponentines programų sistemas. *Informacijos mokslai*, nr. 50, p. 168–172.

2. **Giedrimas, V.** (2007). Generavimo metodų panaudojimas kuriant .NET komponentines programų sistemas. *Informacijos mokslai*, nr. 42–43, p. 246–250.

3. **Giedrimas, V.** (2006). Induktyvinis metodas komponentinių programų sintezėje. *Liet. mat. rinkinys*, t. 46, spec. nr., p. 103–106.

4. **Giedrimas, V.**, Lupeikienė, A. (2005). Komponentinių programų struktūrinės sintezės teorinės problemos. *Liet. mat. rinkinys*, t. 45, spec.nr., p. 139–143.

5. **Giedrimas, V.**, Lupeikienė, A. (2004). Komponento specifikacijos formalizavimas. *Liet. mat. rinkinys, t. 44*, spec. nr., p. 276–280.

6. **Giedrimas, V.** (2003). Programų sistemų automatizuoto surinkimo iš gatavų komponentų uždavinys *Liet. mat. rinkinys, t. 43*, spec. nr., 228–232

7. **Giedrimas, V.** (2003). Komponento modelis struktūrinės programų sintezės kontekste. *Informacijos mokslai*, nr. 26, p. 246–250.

Užsienio žurnaluose ir leidiniuose bei tarptautinių konferencijų darbuose

1. **Giedrimas, V.** (2006). Architectures of Component-Based Structural Synthesis Systems. Databases and Information Systems. *Seventh International Baltic Conference on Databases and Information Systems. Communications.* Vilnius:Technika, ISBN 9955-28-013-1, p. 331–315.

2. **Giedrimas, V.** (2005). Component-based Software Generation: The Structural Synthesis Approach. *Proceedings of 7th GPCE YRW 2005*, Tallinn, ISBN 9985-894-89-8, p. 19–23.

3. Lupeikienė, A., **Giedrimas, V.** (2005). Component model and its formalisation for structural synthesis. *Scientific Proc. of Riga Technical University. Computer Science, vol. 22*, ISSN 1407-7493, p. 169–180.

Kituose Lietuvos leidiniuose

1. **Giedrimas, V.** (2005). .NET komponento specifikacija programų sintezės kontekste. *Konferencijos „Informacinės technologijos 2005“ medžiaga.* Kaunas, p. 382–385.

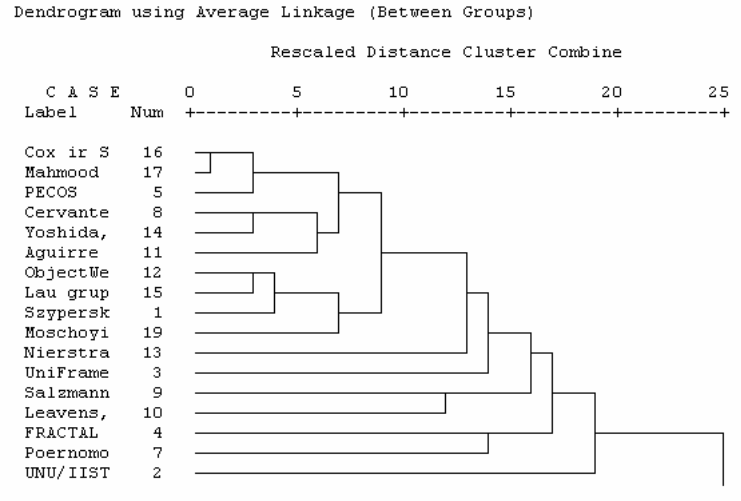
B priedas

Komponento modelių klasterizavimo rezultatai

B.1. Komponento modelių klasteriai

Šiame skyriuje pateikti komponento modelių klasterizavimo rezultatai. Klasterizuota siekiant atskleisti kurie komponento modeliai į kokius klasterius gali būti grupuojami. Dėl metodo specifikos po 25-osios iteracijos visi modeliai priskiriami vienam klasteriui, todėl prasminga nagrinėti tik anstyvosiose iteracijose gautus klasterius.

Pastebima, kad po 5-osios iteracijos 17 komponento modelių sugrupuota į 9 klasterius, o po 15-osios – į 4.

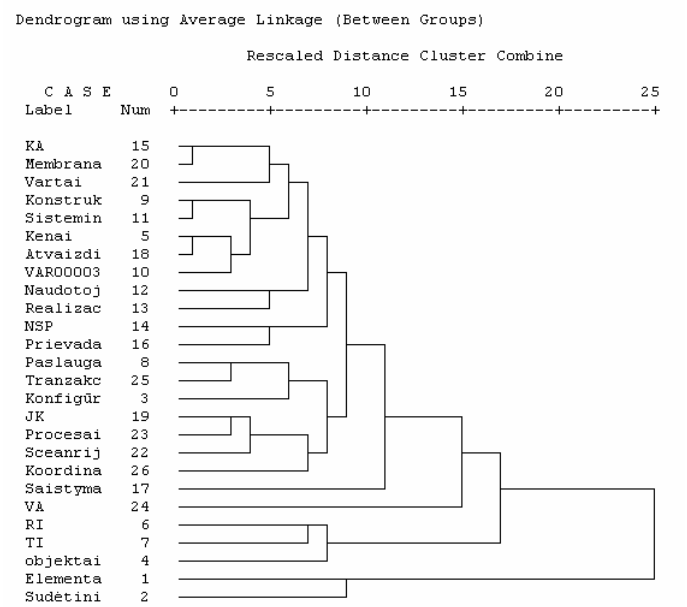


B.1 pav. Komponento modelių klasterių dendograma.

B.2. Komponento modelių savybių klasteriai

Šiame skyriuje pateikti komponento modelių savybių klasterizavimo rezultatai. Klasterizuota siekiant atskleisti kurios savybės galėtų būti grupuojamos. Tikslinga nagrinėti tik pirmąsias šios analizės iteracijas, nes klasterizavimo metodas neleidžia įvertinti atskirų savybių panašumo semantine prasme.

Po pirmosios iteracijos į klasterius suskirstyti šie elementai: valdymo elementai (KA) ir membrana, konstrukcija ir sisteminis interfeisas, kenai ir atvaizdžio tipo jungtis, paslauga ir tranzakcija. Po trečiosios iteracijos – jungiantysis kodas ir procesas, reikalaujami ir teikiami interfeisai. Įdomu tai, kad išskyrus kenų-atvaizdžio ir paslaugos-tranzakcijos poras, grupuojamos sąvokos yra gana artimos ir semantine prasme. Žinoma, procesą tesiant toliau su kiekviena kita iteracija klasterizuojamų objektų semantinis panašumas nyksta.



B.2 pav. Komponento modelių savybių klasterių dendograma.

Vaidas Giedrimas

KOMPONENTINIŲ PROGRAMŲ SISTEMŲ GENERAVIMO METODAS

Daktaro disertacija

Fiziniai mokslai (P 000)
Informatika (09 P)
Informatika, sistemų teorija (P 175)

SL 843.2010-04-02. 14,3 leidyb. apsk. l. Tiražas 20. Užsakymas 27.
Išleido VšĮ Šiaulių universiteto leidykla, Vilniaus g. 88, LT-76285 Šiauliai.
El. p. leidykla@cr.su.lt, tel. (8 ~ 41) 52 09 80, faks. (8 ~ 41) 52 09 80.
Spausdino UAB „Šiaulių knygrišykla“, P. Lukšio 9A, LT-76207, Šiauliai.