KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
PROGRAMŲ INŽINERIJOS KATEDRA

Ramūnas Burneckis

Objektinių aparatūros projektavimo metodų tyrimas

Magistro darbas

Vadovas:
dr. R. Damaševičius

Kaunas, 2007

KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
PROGRAMŲ INŽINERIJOS KATEDRA

Objektinių aparatūros projektavimo metodų tyrimas

Magistro darbas

Vadovas:
dr. R. Damaševičius
2007-05

Recenzentas:
doc. dr. G.Ziberkas
2007-05

Atliko:
IFM-1/5 gr. stud.
Ramūnas Burneckis
2007-05-25

Kaunas, 2007

Santrauka

Tyrinėjant objektinius aparatūros projektavimo metodus, siekiama rasti sąsajų su jau žinomais ir plačiai naudojamais programinėje įrangoje metodais. Tai leistų sumažinti aparatūros projektavimo sudėtingumą, kuris nuolat didėja. Šiame darbe ir siekiama rasti bei išanalizuoti galimus objektinio projektavimo metodus. Pagrindinės objektinio projektavimo priemonės yra šios: abstrakcija, skaidymas, kompozicija bei apibendrinimas. Aptariamas žinomų iš programinės įrangos šablonų taikymas aparatūros aprašymo kalboje. Naudosime aukšto abstrakcijos lygio sistemų projektavimo kalbą UML ir automatizuoto projektavimo priemones aparatūrai aprašyti. Išnagrinėjęs aparatūros projektavimo kalbų galimybes, pasirinkau SystemC kalbą, nes ji kilus iš C++ programavimo kalbos ir turi daugiausia objektinio aparatūros projektavimo galimybių.

SUMMARY

Researching object-oriented (OO) design techniques, we seek to develop links with already known widely used software design methods. This would let to decrease hardware design complexity, as it constantly increasing. In addition, this paper seeks to find and analyze possible object-oriented design methods. The main OO design techniques are as follows: abstraction, separation of concerns, composition and generalization. Considered-about templates, known for software, use in hardware language. We will use UML – the standard specification language of high-level systems, and automatic design techniques for a hardware describing. After exploring possibilities of hardware design languages, I pick SystemC, because it has derived from C++ language and contain the most OO design possibilities from hardware languages.

Turinys

1. Įvadas 8
1.1. Darbo aktualumas 8
1.2. Mokslinis naujumas 8
1.3. Darbo tikslas ir uždaviniai 9
1.4. Apžvalga 9
2. Probleminės srities analizė 10
2.1. Objektinių aparatūros modelių analizė 10
2.2. Objektinių specifikavimo metodų analizė 12
2.3. Aparatūros projektavimo šablonų analizė 14
2.4. Objektinių platformų analizė 20
2.5. Objektiškai orientuotos technikos 23
2.6. Objektinis programavimas SystemC kalboje 24
2.6.1. Paveldimumas SystemC kalboje 24
2.6.2. Hierarchinio modulio konstrukcija SystemC aparatūrinėms bibliotekoms 25
2.6.3. Daugialypės architektūros 26
2.6.4. Struktūros ir jų konfigūracija 27
2.6.5. C++ polimorfizmas SystemC bobliotekoje 28
2.6.6. Persidengimo mechanizmo naudojimas elgsenos pakeitimui 29
2.6.7. Persidengimo naudojimas modeliavimo pagreitinimui arba apbrėžiant tam tikra elgseną 29
2.6.8. Projektavimo šablonai 30
2.7. Išvados 31
3. UML susiejimas su SystemC 34
3.1. UML klasių diagramos aprašančios procesorių 36
3.2. UML būsenų diagramos aprašančios procesorių 38
4. UML kodo generatoriaus modifikacijos ir rezultatai 41
Išvados 43
Literatūra 44
Priedai 46

Paveikslų sąrašas

1 pav. Objekto elgsenos modelis 10
2 pav. Objekto struktūrinis modelis 11
3 pav. Objekto FSM-based modelis 12
4 pav. Sąjungos projektavimo šablonas 15
5 pav.Būsenų projektavimo šablonas 16
6 pav. Deimanto projektavimo šablonas 17
7 pav. Aplanko projektavimo šablonas 19
8 pav. Kompozitoriaus projektavimo šablonas 20
9 pav. OO-ASIP platformos architektūra 21
10 pav. Enodia platformos architektūra 22
11 pav. Modulio specializacija 25
12 pav. Elgsenos hierarchija 26
13 pav. Elgsenos subtilybė 27
14 pav. Dalinė SystemC architektūra 34
15 pav. UML klasės susiejimo su SystemC kalba metamodelis (ryšiai) 35
16 pav. Procesoriaus klasių diagrama 36
17 pav. Pipelined_DLX_Processor klasių diagrama 37
18 pav. Pipelined_DLX_Processor įgyvendinimas 38
19 pav. Read_write klasės būsenų diagrama 39
20 pav. Fetch_execute klasės būsenų diagrama su pertraukimais 40
21 pav. Read_write_execute klasės būsenų diagrama 40
22 pav. Kodo generatoriaus struktūra 41
23 pav. Read_write_execute klasės modeliavimo rezultatai 42

Lentelių sąrašas

1 lentelė. UML diagramų tipai..14
2 lentelė. Šablonų apibendrinimas..32
3 lentelė. Kodo generatoriaus architektūra...46

1. Įvadas

1.1. Darbo aktualumas

Aparatūros projektavimo sudėtingumas nuolat auga. Tyrinėtojai bando atrasti naujus, kuo abstraktesnius ir produktyvesnius projektavimo metodus, arba pritaikyti jau žinomus iš kitų sričių, tokiu kaip programinės įrangos projektavime.

Bendrai paėmus aparatūros sritis yra sudėtinga. Modernūs lustai gali turėti virš dvidešimt penkių milijonų trigerių ir gali reikalauti šešis ar aštuonis mėnesius projektavimo darbo. Ištisos lustų sistemos su daugialypiais mikroprocesoriais, atminties elementai su integruota programine įranga ir specifinės taikomosios schemos gali būti įdiegtos į viena lustinį elementą. Žinoma, kad puslaidininkių technologija dabar leidžia daug sudėtingesnes sistemas kurti, nei aparatūros projektuotojai gali suprojektuoti. Loginių tranzistorių, kurie gali būti integruoti lustų sistemoje, lygmenyje sudėtingumas auga apie 58 procentus kiekvienais metais (pagal Mūro dėsnį). Tuo tarpu projektavimo produktyvumas išauga apie 21 procentą per metus (SEMATECH, 2001).

1.2. Mokslinis naujumas

Objektinių priemonių taikymas įvairiose srityse žinomas jau gana seniai. Ypač išpopuliarėjo programinėje įrangoje, nes šioje srityje labai pasiteisino ir buvo tarsi perversmas nuo iki tol egzistavusių projektavimo metodų. Iš karto buvo kuriamos naujos, arba plečiamos senos programavimo kalbos, palaikančios objektinį projektavimą, kurios populiarios iki šių dienų.

Didėjant aparatūrinių sistemų sudėtingumui, projektuotojai vis ieško naujų kelių, kaip sumažinti vis didėjančius laiko ir pinigų kaštus. Naujus metodus atrasti sunkiau, nei pritaikyti jau žinomus. Todėl palyginti neseniai mokslininkams ir projektuotojams kilo mintis pritaikyti programinėje įrangoje naudojamus objektinio projektavimo metodus, šablonus ir aparatūros projektavime. Taip pat vis dažniau naudojama pakartotinio panaudojimo technologija, leidžianti padidinti projektavimo efektyvumą.

Pagrindiniai projektavimo sudėtingumo valdymo tipiški minėtini terminai yra hierarchija bei abstrakcija. Projektavimo problemos hierarchinė, struktūrinė dekompozicija sumažina sudėtingumą izoliuojant mažesnės problemas, taigi daro bendrą, pagrindinę problemą prieinamą sprendimui.

Abstrakciją reikėtų žiūrėti dviem požiūriais. Pirmasis leidžia apimti egzistuojančius komponentus, kurie gali būti aprašyti abstrakčiu modeliu, turinčiu savyje tik tą informaciją, kurios reikalauja aukštesnio lygio abstrakcija. Kitu atveju, abstrakcija gali būti naudojama mažinant projektavimo pastangas, jeigu projektavimo detalės automatiškai priskiriamos mažiau detalizuotom projektavimo specifikacijom naudojant sintezės įrankius.

1.3. Darbo tikslas ir uždaviniai

Magistro darbo tikslas - ištirti objektinio projektavimo taikymo galimybes aparatūros projektavime. Šiam tikslui pasiekti, suformuluoti tokie uždaviniai:

- Apžvelgti objektinius aparatūros modelius, specifikavimo metodus, platformas, šablonus. Kaip jie gali būti pritaikyti elektroninių sistemų projektavime.
- Išnagrinėti SystemC kalboje galimas objektinio programavimo galimybes.
- UML klasių ir būsenų diagramų pagalba aprašyti procesorių.
- Patobulinti turimą kodo generatorių, kuris transformuoja UML klasių ir būsenų diagramų aprašus į SystemC kalbą, naudojant pasiūlytą metamodelį. Patobulintas kodo generatorius turi palaikyti paveldimumo funkciją. Realizuoti apibendrinimo bei agregacijos ryšius, kurių dėka bus galima generuoti kodą pagal daugumos šablonų sukurtas diagramas.

1.4. Apžvalga

Darbo sandara, be aprašyto įvado yra tokia: antram skyriuje aprašoma probleminės srities analizė, trečiame skyriuje pateikiama UML klasių ir būsenų diagramos, aprašančios procesorių, ketvirtame skyriuje pateikiama kodo generatoriaus architektūra, naudotos priemonės, kalba, patobulinimai, naudoti metaprogramavimo metodai. Darbas baigiamas išvadomis, pateikiamas literatūros sąrašas.

2. Probleminės srities analizė

2.1. Objektinių aparatūros modelių analizė

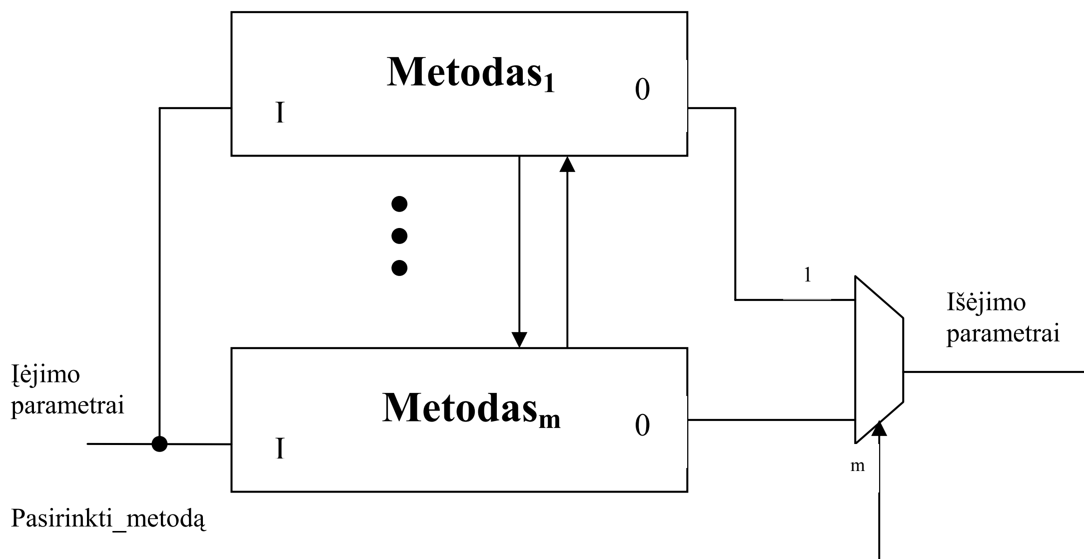
Kiekviena aukšto aprašymo lygio aparatūros specifikacija galiausiai turi būti realizuota iki žemiausio lygio (RTL) aprašo (ventilių ir laidų). Problema tame, kaip tai aprašyti ir įgyvendinti. Iš esmės gali būti trys modeliai, nusakantys tokios aparatūros architektūros įgyvendinimą:

a) Elgsenos modelis. Turimas objektas, kuris apibrėžiamas kaip duomenų paketas, kai duomenys keliauja laidais. Į objektus nežiūrima kaip į komponentus, bet greičiau kaip į ryšius tarp komponentų, kurie patys yra kaip metodai. Metodas atitinka aparatūros komponentą, kuris apdoroja duomenų paketus ir grąžina rezultatą. Objekto būsena (Q, Q^+) taip pat bendrauja (gauna) su kito metodo parametrais, tokiais kaip aparatūros komponento įėjimai (I) ir išėjimai (O). Grąžinamoji metodo reikšmė tampa komponento išėjimo parametru. Šis metodas palaiko duomenų baigtinumą ir atskiriamumą nuo funkcionalumo.

1 pav. Objekto elgsenos modelis

b) Struktūrinis modelis atstovauja fizinius komponentus, tokius kaip VHDL objektus (entities). Objektas yra aparatūros komponentas, kuris, laidų pagalba, gauna informaciją apie metodą. Objektai bendrauja tarpusavyje per metodų iškvietimus (calls), tokiu būdu metodai interpretuojami kaip jungtys/portai. Paveldėjimas galimas sukuriant objektą vaiką iš tėvo objekto, pridedant naujų metodų. Abstraktumas (encaptulation) galimas sukuriant papildomą sąsają, leidžiančią pažadinti (aktyvuoti) metodus. Polimorfizmas (polymorphism) galimas dinamiškai aktyvuojant modelių objektus (entities) esamuoju laiku.

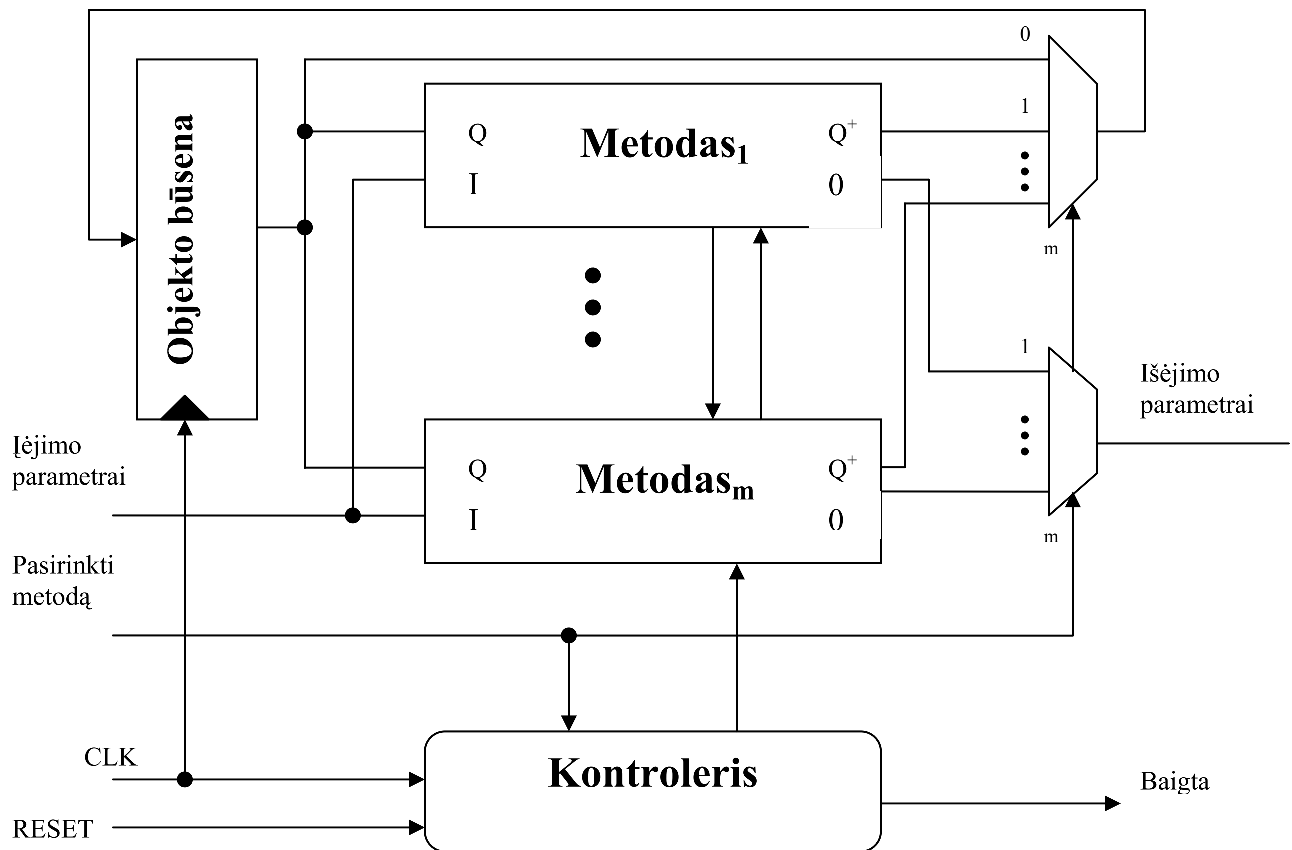
Čia „Pasirinkti metodą“ yra metodo identifikavimas, „Įėjimo parametrai“ ir „Išėjimo parametrai“ yra objekto metodų įėjimai ir išėjimai.



2 pav. Objekto struktūrinis modelis

c) FSM-based modelis atstovauja kiekvieną objektą kaip FSM (angl. Finite state mashine – baigtinis būsenų automatas). Pagrindinė architektūra turi atminties elementą, kuriame saugoma būsena, būsenos perėjimas ir loginis išėjimas, bei sekančios būsenos atsakomoji reakcija į būsenos atmintį. Metodas atitinka aparatūros komponentą, kuris įgyvendina metodų aprašytą funkcionalumą. Objekto būsena ir metodo parametrai yra kartu ir komponento įėjimai, ir išėjimai nuo to laiko, kai juos galima būtų perskaityti ir koreguoti. Keletas objekto atminties bitų gali būti panaudoti klasių bendravimui parodyti. Pridedant į objektą naudų metodų ir praplečiant objekto būseną atsiranda paveldimumas. Polimorfizmas (polymorphism) pasiekiamas naudojant perjungimo (switch-case) konstrukciją (įgyvendinama kaip multiplekseris) tam, kad patikrinti objekto tipą realiu laiku ir iškviesti atitinkamą metodą. Pasikeitimas žinutėmis tarp objektų taip pat galimas per kanalų tinklą.

Čia „Pasirinkti metodą“ yra metodo identifikavimas, „Įėjimo parametrai“ ir „Išėjimo parametrai“ yra objekto metodų įėjimai ir išėjimai, „Baigta“ signalas parodo metodo vykdymo pabaigą, atminties elementas išsaugo objekto būseną, CLK yra sinchroninis signalas ir RESET yra anuliavimo signalas.



3 pav. Objekto FSM-based modelis

Pabandysim apibendrinti šiuos tris objektiškai orientuotus aparatūros modelius. Elgsenos modelis yra paprasčiausias: jis leidžia tik abstraktumą (encapsulation). Struktūrinis modelis leidžia neturintiems būsenos statiniams objektams turėti paveldimumą. FSM-based modelis apibrėžia statinius objektus su paveldimumo palaikymu, žinučių apsikeitimu ir polimorfizmu. Dinaminiai objektai negali būti tiesiogiai įgyvendinti aparatūroje, nepaisant to, jie gali būti pamėgdžioti, kai aparatūros objektai yra derinami su programinės įrangos objektais dalinėje atmintyje. Šis atvejis naudojamas aparatūros platformose.

2.2. Objektinių specifikavimo metodų analizė

Objektiškai orientuoto projektavimo principai gali būti įvesti aparatūrinę sritį naudojant Petri tinklų formalius užrašymus. Petri tinklai suteikia galimybę modeliuoti elgesį, sutampanti kartu su duomenimis, kuriais manipuliuojama. Aukšto lygio Petri tinkluose duomenys modeliuojami kaip simboliai, esantys tam

tikrose vietose. Čia gali būti keletas bandymų kombinuoti struktūrines objektinio projektavimo priemones su Petri tinklais. Šie būdai pristato klases, kurios apima simbolinius duomenis ir funkcijas, kurios vykdomos perėjimų viduje. Šiuo metu dauguma egzistuojančiu objektinių Petri tinklų būdų (traktavimų) tik atsižvelgia į tai, kad simboliniai duomenys būtų objektiškai orientuoti. Vis dėlto LOOPN++ (nauja kalba, skirta objektiškai orientuotiems Petri tinklams) skirta ne tik simboliniams objektiškai orientuotiems duomenims, bet taip pat patiems Petri tinklams. Kiekvienas Petri tinklo objektiškai orientuotas elementas yra objektas, kuris taip pat gali turėti savyje Petri tinklą. Tai suteikia galimybę modeliuoti labai sudėtingas daugelio lygių sistemas.

Objektiškai orientuoto projektavimo idėja į Petri tinklų ženklų sistemą (notaciją) palaiko augančius (besivystančius) komponentus, kurie apima elgesį ir būseną, bei palaiko abstrakciją, tobulinimą, sutraukimą ir pakartotinį panaudojimą. Aparatūriniai komponentai aprašomi naudojant klases, kurios iš anksto paruošia konfigūracijos bei pakartotinio panaudojimo mechanizmus. Sistema yra kiekvieno abstrakcijos lygio bendradarbiaujančių komponentų kompozicija (suma). Sistemos projektavimo laikotarpiu, klasės yra apibrėžiamos, tobulinamos, pakartotinai naudojamos bei konfigūruojamos. Abstrakcija ir tobulinimas, keičiant kuo mažiau tobulinamus komponentus komponentais, turinčiais padidėjusi detalių kiekį, leidžia kompleksines sistemas modeliuoti ir sėkmingai tobulinti link įdiegimo.

Kita sistemų projektavimo ir modeliavimo kalba, plačiai naudojama įvairiose srityse ir tapusi standartu projektavimo sistemų modeliavime, yra UML (angl. Unifidied Modeling Language). Ji naudojama ir fizikoje, verslo procesų modeliavime ir t. t. Buvo prieita prie minties, kad ir aparatūrinės įrangos projektavime galima pasinaudoti šios kalbos galimybėmis (Schattkowsky, 2005). Labiausiai UML paplito programinės įrangos projektavimo srityje, sėkmingai ten taikoma, taigi kilo mintis, kad galima šią universalią modeliavimo kalbą pritaikyti aparatūros projektavime.

UML yra vizualinė, grafinė kalba, skirta programinės įrangos architektūros dokumentavimui, apibendrinimui, projektavimo sprendimų aptarimui. Šioje kalboje informacija gali būti pateikiama pasitelkiant įvairias struktūrines ir elgsenos diagramas.

UML pateikia dvylika diagramų tipų suskirstytų tris klases:

1 lentelė. UML diagramų tipai

Struktūrinės diagramos	Klasių (class), objektų (object), komponentų (component) bei deployment diagramos
Elgsenos diagramos	Panaudos atveju (use case), sekų (sequence), veiksmų (activity), koloboravimo (coboloration), būsenų (statechart) diagramos
Modelio valdymo diagramos	Paketų (packets), posistemių (subsystems) bei modelių (models) diagramos

UML galima pavaizduoti tik bendro pobūdžio notacijas, kurios turi būti pritaikytos proceso vystymo metu. Skirtingos diagramos naudojamos skirtingiems tikslams apibrėžti.

Klasių diagramų pagalba, apibrėžiama sistemos sudedamosios dalys, struktūra. Klasės gali turėti atributus ir metodus. Jos jungiamos įvairios paskirties ryšiais: bendrinimo (generalize), agregacijos ir pan. Taip sukuriamos objektiškai orientuotos hierarchijos. Būsenų diagramos aprašo veiksmų seka, esant tam tikroj, konkrečioj būsenoj, bei perėjimus iš vienos būsenos į kitą. Aukščiausiame abstrakcijos lygyje sistemos modeliavimas vyksta naudojant panaudos atvejų diagramas. Jose aprašomi galimi veiksmai bei aktoriai, susiję su aprašomąja sistema.

Taip pat ir kitos diagramos aprašo sistemoje vykstančius aspektus. Jų pagalba galima gana išsamiai specifikuoti nagrinėjamą sistemą įvairiais abstrakcijos lygiais. Šiame darbe naudosimės UML klasių ir būsenų diagramomis.

2.3. Aparatūros projektavimo šablonų analizė

Yra du siūlymai kaip projektavimo šablonus realizuoti aparatūros lygyje. Pirmas būdas reikalauja jau žinomus programinės įrangos projektavimo šablonus pritaikyti aparatūros projektavimui. Antras būdas siūlo ieškoti ir atradinėti naujus specifinius aparatūros projektavimo šablonus. Toliau aprašysim keletą programinės įrangos projektavimo šablonų ir jų taikymą aparatūriniam projektavime.

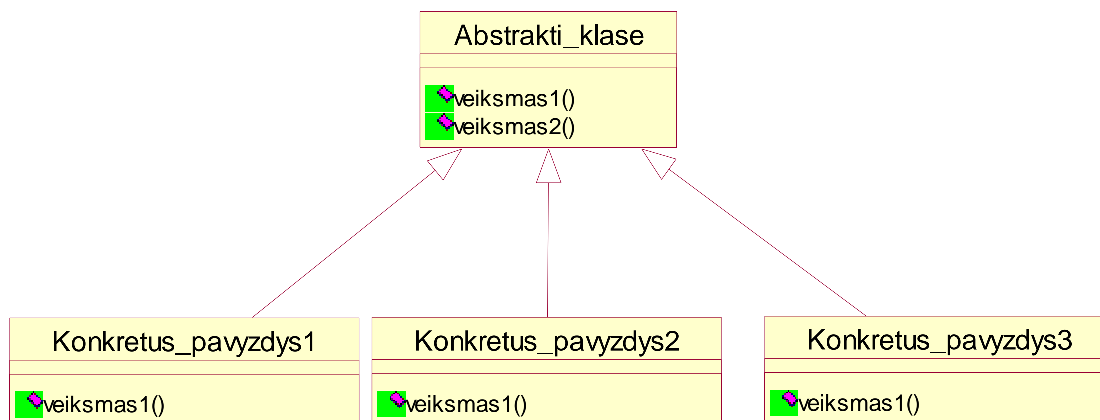
a) „Sąjungos projektavimo šablonas“ (union design pattern) naudojamas abstrakcijos sluoksnių apibrėžimui projektavime. Jis atstovauja keletą sistemos (komponento) konkrečių vaizdų apibendrinimą. Abstrakti superklasė („tėvo“ klasė) „AbstractRepresentation“ vaizduoja aukštesnio

lygio abstrakciją nei konkrečios „vaikų" klasės. „Sąjungos šablonas" tokiu būdu apibrėžia du skirtingus abstrakcijos lygius. Bet kuriuo duotu momentu operacija gali būti atlikta ir žemesniame, konkretesniame abstrakcijos sluoksnyje, ir aukštesniame, abstraktesniame sluoksnyje.

4 pav. pavaizduota „Sąjungos projektavimo šablono" UML klasių diagrama. Abstrakti klasė „Abstrakti_klase" apibrėžia abstraktų (t.y. dar neaprašytą) metodą *veiksmas1()* ir įdiegia metodą *veiksmas2()*. „Konkretus_pavyzdys" klasės paveldi metodo *veiksmas2()* įgyvendinimą iš „Abstrakti_klase" klasės bei parūpina metodo *veiksmas1()* įgyvendinimą.

Šio šablono panaudojimas leidžia visiškai atskirti pastovius (nesikeičiančius) projektavimo problemų aspektus ir kintančius. Abstrakti „tėvo" klasė yra esmės vaizdas, kuris yra bendras visoms galimoms konkrečioms „vaiko" klasėms. Ji tokiu būdu atstovauja pastovius visų klasių sistemoje aspektus. Konkrečios „vaiko" klasės yra vienodos aukštesniame abstraktesniame lygyje, ir skiriasi viena nuo kitos žemesniame abstrakcijos lygyje. Tokiu būdu jos atstovauja įvairius projektavimo problemų aspektus. Tikrasis sistemos elgesys yra nekintamo „tėvo" klasės elgesio bei įvairių konkrečių klasių elgesių kompozicija.

„Sąjungos projektavimo šablonas" gali būti panaudotas aparatūrinio projektavimo plotmėje, kad aprašyti konfigūruojamas aparatūrines sistemas. „Configuration" sakinys VHDL reiškia, kad leidžiama apibrėžti konkretaus projektavimo objekto sistemoje konkretų įgyvendinimą. Pavyzdžiui, ALĮ (aritmetinis loginis įrenginys) gali būti įdiegtas kaip dekodavimo operacijos (pastovūs projektavimo aspektai) bei abstraktaus projektavimo objektai, kurie atstovauja ALĮ operacijas (įvairūs projektavimo aspektai), kompozicija. Tam tikri ALĮ operacijų įgyvendinimai gali būti apibrėžti nepriklausomai, taigi duodami daugiau lankstumo projektuotojui pasirinkti optimalų ALĮ įgyvendinimą, atsižvelgiant į reikalaujamas projektavimo charakteristikas (mikroschemos plotas, greitis ir t.t.).



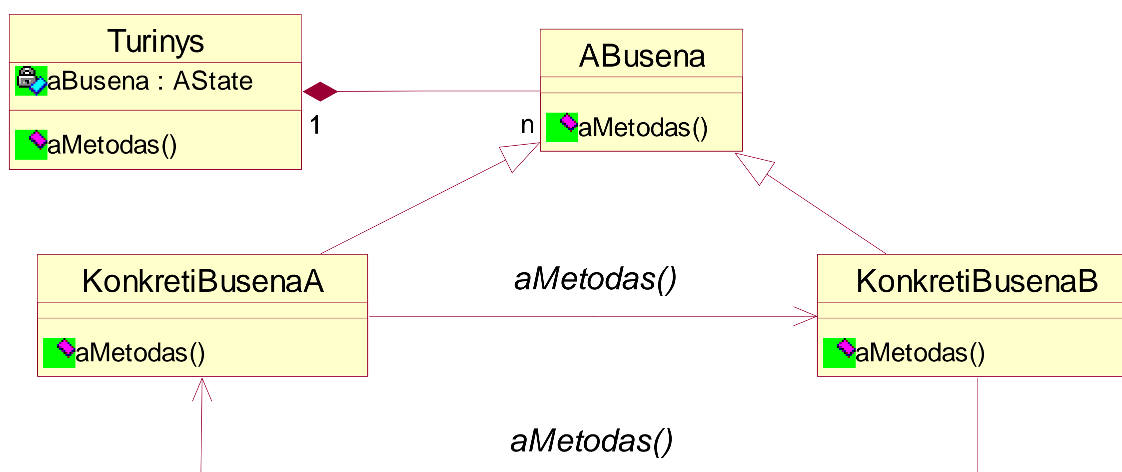
4 pav. Sąjungos projektavimo šablonas

b) „Būsenų projektavimo šablonas“ (state design pattern) naudojamas sistemos atskyrimui nuo jos būsenos bei funkcionalumo susijusio su šia būsena apibendrinimui skirtingose klasėse. Būsenos yra atskirtos ir įdėtos į skirtingas klases. Privalumas tai, kad galimybė valdyti sistemą yra daug didesnė, nei elgsena ar algoritmas, kuris atliktas, kai sistema pereina į tam tikrą būseną, yra įdedama į savo nuosavą klasę, geriau nei paslėpta kažkur kažkokiame metode.

5 pav. pavaizduota UML klasių diagrama. Klasė „Turinys“ turi savyje būseną. Abstrakti klasė „ABusena“ aprašo būsenos funkcionalumą. „KonkretiBusena“ klasės aprūpina tam tikrų klasių įdiegimą.

Šis šablonas gali būti panaudotas aparatūrinėje plotmėje projektuojant FSM (angl. Finite State Mashine). Kiekviena „KonkretiBusena“ klasė pasirūpina atskiros FSM būsenos įdiegimu bei atitinkamu funkcionalumu, kuris atliekamas, kai FSM įžengia į šią būseną. Šis sprendimas leidžia projektuoti FSM, be to gali būti lengvai pakartotinai panaudojama ir praplečiama naujomis būsenomis, reikalingomis naujoms realizacijoms.

Šio šablono trūkumas tai, kad tai gali įtakoti pernelyg didelį projektuojamos sistemos sudėtingumą nuo to laiko, kai kiekvienai būsenai atskiras objektas yra nustatytas vėlesniam įdiegimui kaip atskiras komponentas. Taigi, šio šablono realizacija veda prie blogų projektuojamos sistemos vykdymo charakteristikų. Kompromisas šiuo atveju tarp greitesnio sistemos įdiegimo ir blogų charakteristikų.

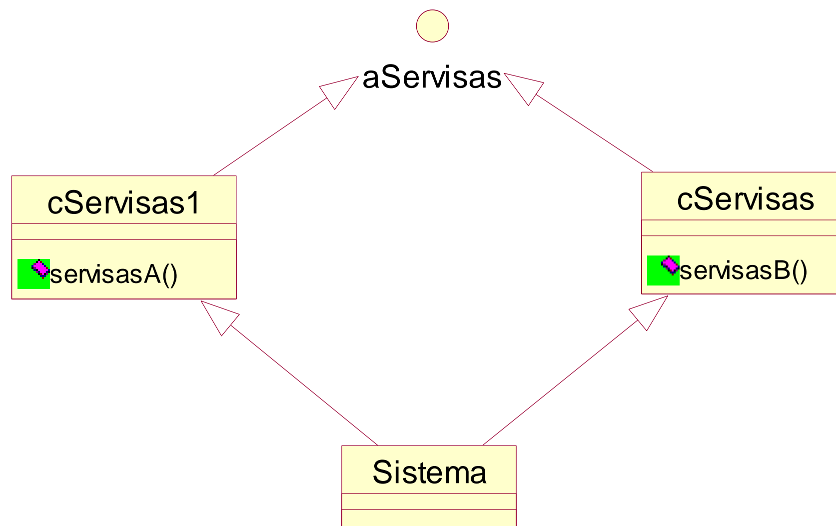


5 pav.Būsenų projektavimo šablonas

c) „Deimanto projektavimo šablonas“ (Diamond design pattern) naudojamas atvaizduoti projektavimo elgsenos sudėtingumui, kuris gaunamas naudojant keletą servisų (aptarnavimų) šioje aplinkoje. Serviso klasių kompozicijai pasiekti naudojamas paveldimumas. Jis leidžia gautai klasei įgyti dviejų ar daugiau „tėvo“ klasių savybes.

6 pav. pavaizduota klasių diagrama. Abstrakti klasė „aServisas“ aprašo sąsają, kuri yra bendra visiems sistemos aprūpinamiems servisams. „cServisas“ klasės paveldi „aServisas“ klasės savybes ir aprūpina konkretaus serviso implementaciją. Klasė „Sistema“ paveldi „cServisas“ klasių servisų realizacijas ir aprašo multiserviso sistemą.

„Deimanto šablonas“ gali būti plačiai panaudotas aparatūroje bei įmontuotų (embedded) sistemų projektavimo plotmėje. Pavyzdžiui, sudaromoms naujoms sistemoms iš galimų aparatūrinių ir/arba programinės įrangos komponentų. Šablono trūkumas tai, kad jis gali padaryti įžangą žymiam darbo perkrovimui, nes ne visi klasių servisai suprojektuoti naudojant paveldimumą, kuris gali būti reikalaujamas projektuojamai sistemai.



6 pav. Deimanto projektavimo šablonas

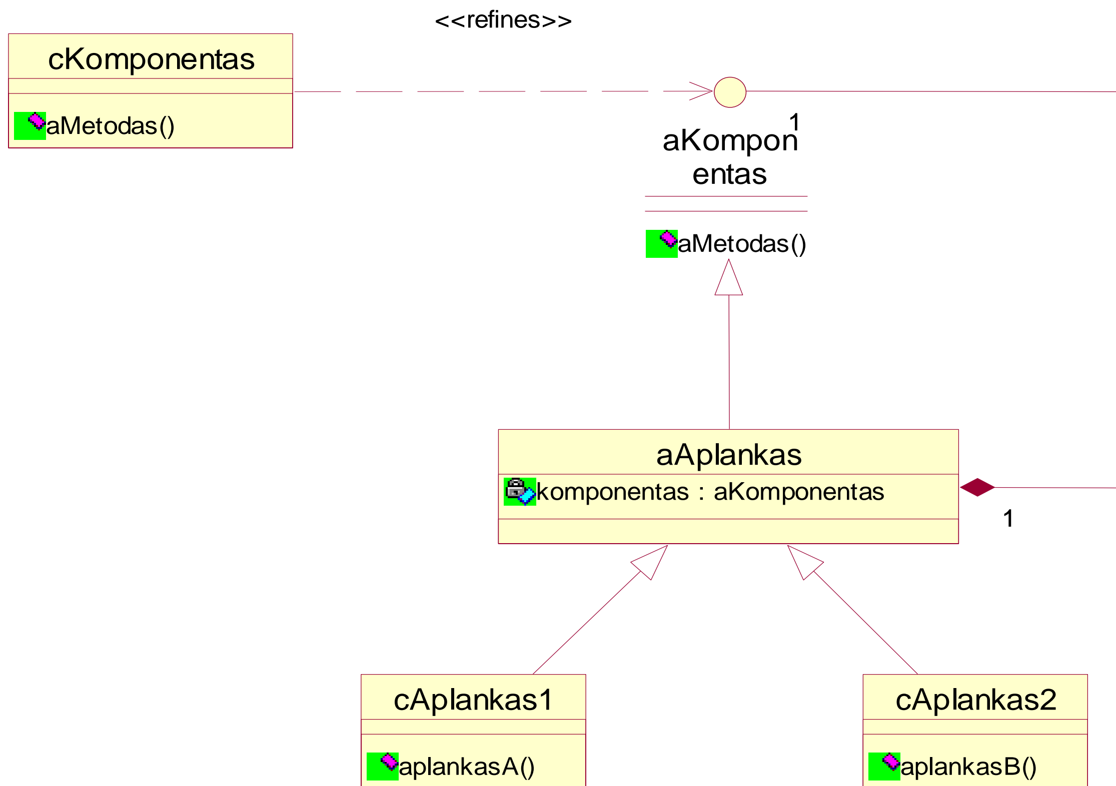
d) „Aplanko/dekoratoriaus projektavimo šablonas“ (Wrapper/decorator design pattern) naudojamas pridedant papildomą funkcionalumą tam tikram komponentui. Aplankas (arba dekoratorius) yra komponentas, turintis keletą sąsajos dalių, identiškų turimo komponento sąsajai. Bet koks iškvietimas, kuri gauna aplankas, jis siunčia komponentui, kurį jis turi, bei prideda savo nuosavą funkcionalumą ir prieš, ir po iškvietimo. Tai įgalina didelį lankstumą ir pakartotinį

panaudojimą tol, kol projektuotojas gali pakeisti aplanką nekeisdamas jame esančių (apgaubtų) komponentų. Kadangi projektuotojas gali rekursiškai įdėti aplankalus vieną i kitą, šablonas leidžia beveik begalinį pritaikomumą.

7 pav. pavaizduota UML klasių diagrama. Abstrakti klasė „aKomponentas“ aprašo sąsają, kuri yra bendra visiems komponentams ir jų aplankams. Klasė „cKomponentas“ rūpinasi klasės „aKomponentas“ įdiegimu. Abstrakti klasė „aAplankas“ aprašo sąsają ir turi savyje „aKomponentas“ klasės pavyzdį. „cAplankas“ klasės rūpinasi skirtingomis aplanko realizacijomis. Aplanko šablonas gali būti plačiai naudojamas aparatūriniam projektavime, ypač dėl:

1. bendradarbiavimo kontrolės, rūpinantis bendradarbiavimo protokolo realizacija, tam, kad aparatūros komponentas bendrautų su savo aplinka;
2. trūkumų (klaidų) tolerancijos realizacijų, kad įdiegti trūkumų vengimo bei trūkumų atradimo technikas tokias kaip TMR (Triple-Modular Redundancy) ir BIST (Built-In Self Test);
3. fizikinės atminties sąsajos pritaikymo bendradarbiavimo tinklui, kad galima būtų turėti skirtingą skaičių priėjimų prie portų;
4. sluoksniuotų duomenų paketo vykdymo tinklų realizacijose.

Šio šablono trūkumas tai, kad gali įvykti žymi perkrova nuo to laiko, kai užbaigta aplanko komponento sąsaja turi būti valdoma aplanko, įtraukiant tuos sąsajos elementus, kurie nepritaikyti. Taigi aplanko šablono realizacija gali vesti į pernelyg didelį kodo pritaikomumo bei atlikimo perkrovos reikšmingumą.



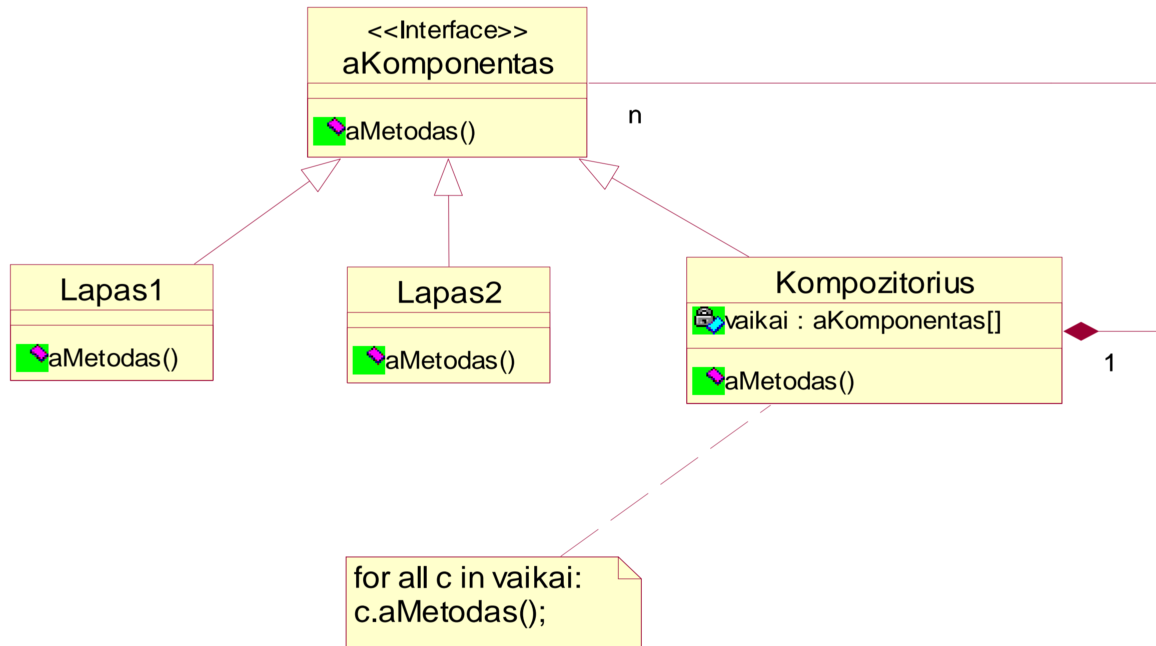
7 pav. Aplanko projektavimo šablonas

e) „Kompozicinis projektavimo šablonas“ (Composite design pattern) leidžia identiškai nagrinėti ir atskirus komponentus, ir komponentų rinkinius. Jis leidžia kurti kompleksinius komponentus rekursiškai komponuojant panašius komponentus, naudojantis hierarchine medžio tipo struktūra. Šis šablonas taip pat leidžia valdyti komponentus nuosekliu būdų, pareikalavus, kad visi komponentai turėtų bendrą tėvą. Šis šablonas dažnai naudojamas rekursinių sistemų architektūrom bei jų elgsenom atvaizduoti.

8 pav. pavaizduota UML klasių diagrama. Abstrakti klasė „aKomponentas“ aprašo sąsają, kuri yra bendra visiems tos klasės įgyvendinimams. „Lapas“ klasės turi skirtingas „aKomponentas“ realizacijas. „Kompozitorius“ klasė yra rekursiškai sudaryta ir „aKomponentas“ klasės tipo masyvo komponentų (kurie gali būti „Lapo“ arba „Kompozitorius“ tipo) ir vykdo rekursinius jų operacijų iškvietimus.

„Kompozicinis projektavimo šablonas“ gali būti plačiai naudojamas sparčiai realizuojamų sistemų aparatūros projektavimui su pasikartojančiom architektūrom, tokiom kaip multiplekserių, registrų masyvai, sumavimo įrenginiai ir t.t. Šio šablono naudojimo privalumas yra pakopinis

realizavimas, tai reiškia dalis sistemos gali elgtis taip pat, kaip ir visa sistema. O trūkumas tas kaip ir daugumoje projektavimo šablonų, kad žymaus persidengimo atlikimo galimybė pateikta kaip rekursinis komponentų komponavimas.



8 pav. Kompozitoriaus projektavimo šablonas

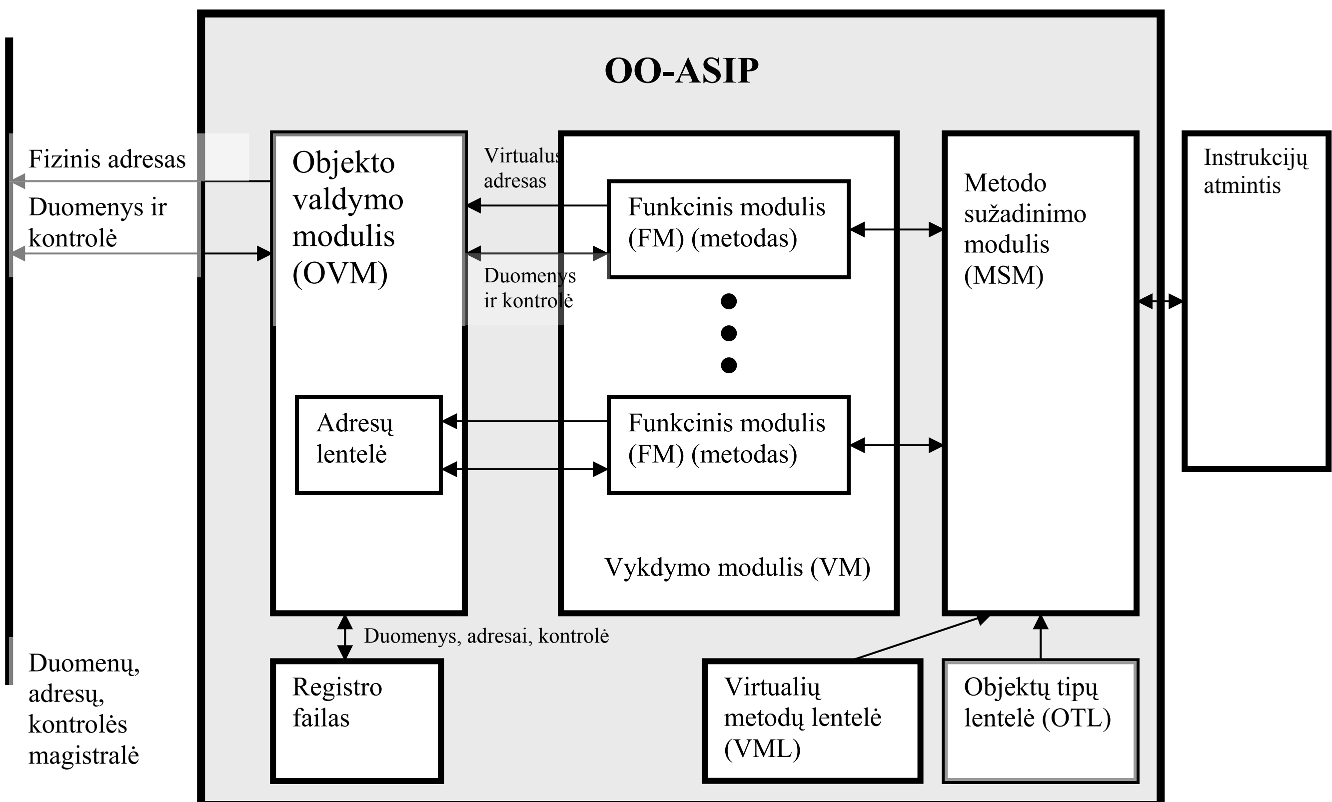
2.4. Objektinių platformų analizė

Objektinio aparatūros projektavimo principai taip pat gali būti pritaikomi platformomis paremtu integruotų sistemų projektavimui. Platformos yra sumodeliuotos naudojant klases ir ryšius tarp jų. Paanalizuosime OO-ASIP ir Enodia platformas.

a) OO-ASIP platformoje objekto metodų sužadinimas naudojamas kaip instrukcija, kuri bus vykdoma procesoriuje apibrėžiant objektą ir papildomus argumentus kaip instrukcijos operandus. Klasių bibliotekos identifikuojami „public“ metodai apibrėžia atitinkamo procesoriaus instrukcijų aibę. OO-ASIP yra procesorius su savo instrukcijų aibe, apibrėžta pasirinktais metodais iš „aparatūros klasių bibliotekos“ ir su aparatūros bei programinės įrangos virtualių metodų dinamiško siuntimo palaikymu. Integruotas pritaikymas sumodeliuotas naudojant „aparatūros klasių biblioteką“ ir sukompiliuotas į atitinkamo OO-ASIP instrukcijas. „Aparatūros klasių biblioteka“ gali būti papildyta pridedant naujų klasių, o tai užtikrina tolesnį pakartotinį naudojimą.

Tokio tipo procesorius pajėgus įvykdyti bet kokia programą, susidedančią iš tos objektų klasės hierarchijos, pvz., visi uždaviniai gali būti sumodeliuoti su „aparatūros klasių biblioteka“.

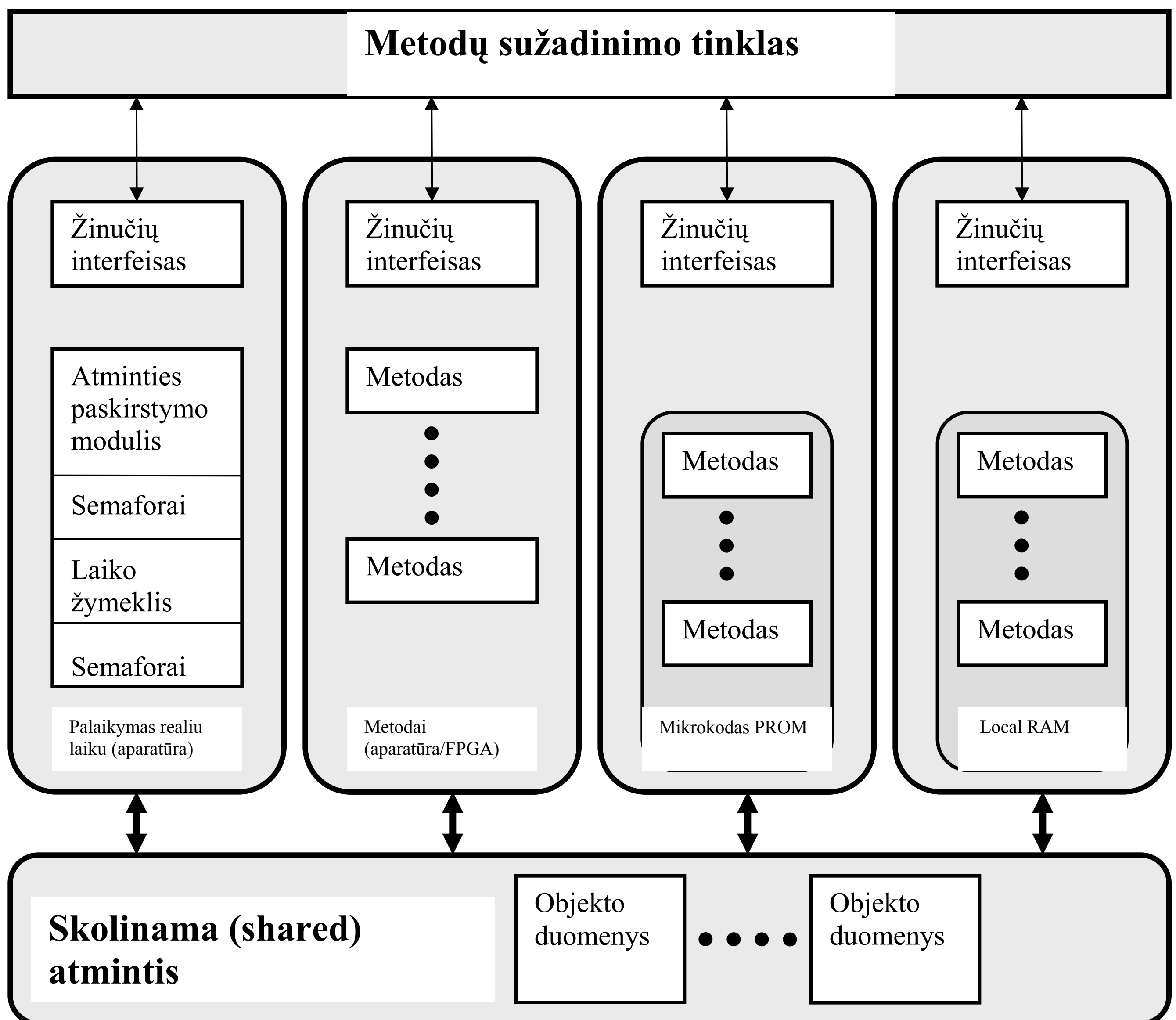
Sistema, suprojektuota ant OO-ASIP platformos veikia taip – „sužadinimo metodų skyrius“ (MIU) perskaito metodo sužadinimo komandas iš instrukcijos atminties ir nurodo metodą bei iškviestą objektą. Pagrįstas iškviesto objekto klasės draugyste, „metodo sužadinimo modulis“ (MSM) sužadina atitinkamą „Funkcinį modulį“ (FM), kuris įtraukia aparatūrinį metodą, arba siunčia metodo iškvietimą programinės įrangos metodui. Kad prieiti prie objekto duomenų, FM bendradarbiauja su „Objektų valdymo moduliui“ (OVM), kuris atlieka operacijas su atmintimi.



9 pav. OO-ASIP platformos architektūra

b) Enodia yra objektiškai orientuoto multiprocesoriaus platforma, kuri naudoja tinklo struktūrą tam, kad siųstų žinutes tarp vykdomų elementų. Enodia platforma naudoja mikroprogramą, kad dinamiškai išskaidyti virtualius metodus atitinkamai vykdomiems elementams, ir aprūpina pastovią aplinką susiejant kombinuota aparatūrinę ir programinės įrangos objektiškai orientuotą sistemą.

Dabar aprašysim esmines Enodia platformos savybes. Objektai turi būseną, kuri saugoma ir skolinamoje (dalinėje), ir privačioje atmintyje, bei gali būti pakeista tik sužadinus atitinkamą objekto metodą. Objekto būsenai skolinamoje (dalinėje) atmintyje, visi metodai, susiję su nagrinėjamo objekto klase, turi nekintamą objekto būsenos vaizdą dėka įdiegimo. Tokiu būdu metodai gali būti perskirstyti tarp programinės įrangos bei aparatūros resursų. Metodai gali būti fiziškai paskirstyti sistemoje; todėl bet koks metodas gali būti įdiegtas programinėje įrangoje, aparatūroje arba FPGA. Platforma naudoja atskirą tinklą metodų iškvietimams ir gražina rezultatą. Taip pat ji gali palaikyti polimorfizmą dinamiškai pažymint duoto objekto iškvietimo žinutės buvimo vietą, esančią kviesto objekto „vaiko“ klasėje.



10 pav. Enodia platformos architektūra

Abi analizuotos platformos realizuoja pagrindinius objektiškai orientuoto projektavimo aspektus, tokius kaip sutraukimas, paveldimumas, polimorfizmas bei žinučių praėjimas. Skirtumai tarp OO-ASIP ir Enodia platformų yra šie:

a) OO-ASIP platforma realizuoja objektus kaip instrukcijų aibės procesorius, kai tuo tarpu Enodia platforma gali realizuoti objektus ir aparatūrinėje, ir programinėje įrangoje.

b) OO-ASIP platforma pateikia metodo sužadinimą objektui kaip instrukcijų vykdymą procesoriuje, kai tuo tarpu Enodia platforma naudoja tinklo struktūrą žinučių siuntimui tarp vykdomų elementų.

c) OO-ASIP platforma realizuoja tik „aparatūrinius objektus“, kai tuo tarpu Enodia platforma pripažįsta ir aparatūrinius objektus, ir programinės įrangos objektus, kurie maitinami lygiai ir sukeičiami nepaisant jų realizavimo.

2.5. Objektiškai orientuotos technikos

Modeliavimo analizės rezultatų kaštai siūlo patikrinti, kokios apimties modeliavimo kaštai gali būti sumažinti pritaikant aparatūros projektavime objektiškai orientuotus metodus.

Esminis dalykas objektiškai orientuotame projektavime yra sistema su bendraujančiais objektais. Kiekvienas objektas yra charakterizuotas atributais, kurių reikšmes apibrėžia objekto būsena. Atributų reikšmės gali būti keičiamos priklausomai nuo metodų repertuaro. Bendravimas tarp objektų realizuotas sužadinant atitinkamus to objekto metodus. Objektiškai orientuoto programavimo metodikos paprastai yra charakterizuojamos specialiais terminais: klasė, paveldimumas, objektas, polimorfizmas, metodas, sutraukimas bei žinučių perdavimas.

Objektai yra klasės atskiri atvejai. Paveldimumas leidžia sukurti klasių hierarchija, kurioje keleto atskirų klasių savybių yra aprašyta tėvo klasėj. Šita klasių hierarchija ir paveldimumo koncepcija leidžia apriboti bei apibendrinti objektus.

Polimorfizmas yra objektiškai orientuotos kalbos galimybė valdyti keletą metodo versijų, t.y. procesai (veiksmai) skirtingose klasių hierarchijos klasėse su tuo pačiu pavadinimu.

Yra mažiausiai du skirtingi aparatūroje objektiškai orientuoto projektavimo pavyzdžiai. Evoliucinis projektavimas yra paremtas egzistuojančiais fiziniais projektavimo objektais ir siūlo sutraukimu paremtą projektavimą. Pasikeitimai pavyzdžiuose iš projektuotojo reikalauja nuosaikumo, kadangi jis/ji gali likti su tradiciniu, einančiu nuo smulkmenų iki bendrųjų principų, projektavimo būdu. Jei vis dėlto yra

numatytas realus einantis nuo bendrų dalykų prie atskirų projektavimas, ir kuris nepanaudoja egzistuojančių fizinių ar sintezuojamų aparatūros komponentų, pavyzdžio pokyčiai lemia didesnį abstraktaus modeliavimo galingumą.

2.6. Objektinis programavimas SystemC kalboje

SystemC 2.0 kalboje, sistemos lygyje, bendrumas išreikštas abstrakčiomis klasėmis, pavyzdžiui interfeisais, tada pokyčiai yra sutraukti iš interfeisų, turinčių daugialypio paveldimumo galimybę. Bendrumas taip pat gali būti išreikštas projektavimo šablonų panaudojime ir kitur.

2.6.1. Paveldimumas SystemC kalboje

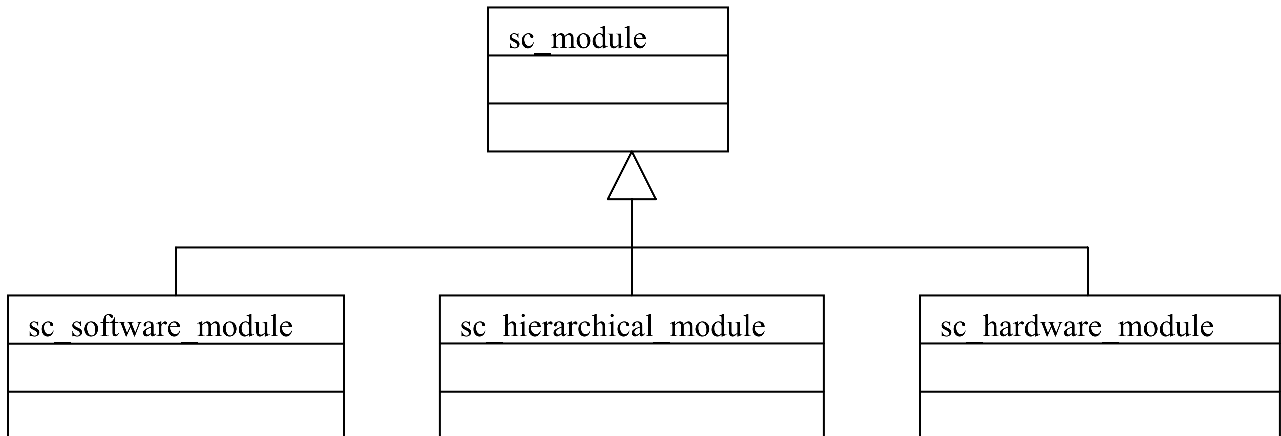
Projektuojant elementų bibliotekas, mes gali apibrėžti visas savybes, susijusias su elementais modulyje *Gate* iš kurio *And, Or, Xor* ir t.t. nustatysime vėliau. *Gate* modulio tikslas yra veikimas, kaip interfeiso. Visos trigerių savybės ir metodai bus aptinkami panašiame modulyje. Pateikiamas paprasto paveldimumo modulio pavyzdys:

```
class Gate : public sc_module {
   public :
   Gate (const sc_module_name& name) : sc_module (name) { (…) }
};

class And: public Gate {
   public :
   SC_HAS_PROCESS (And);
   And (const sc_module_name& name) : Gate (name) { (…) }
};
```

Moderniose, į programinę įrangą orientuotose, metodologijose rekomendacijos pateikiamos atsižvelgiant į bendrą esmę visiems objektais programinės įrangos projektavime. sc_object sudaro pagrindinę SystemC bibliotekos esmę. Modeliavimo tikslui, komponentų ir IP biblioteka, sc_module gali

būti statybinis modelių blokas. Iš to mes galime išvesti kitus specializuotus modulius, kaip parodyta 11 pav.:



11 pav. Modulio specializacija

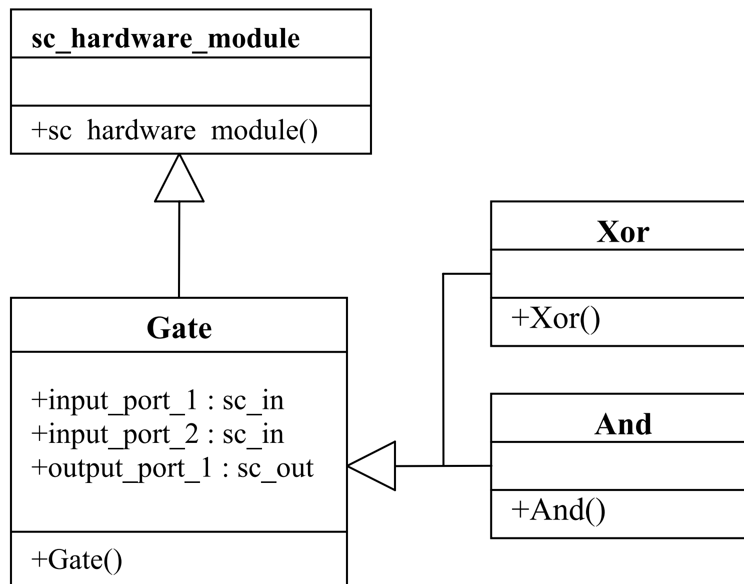
Remiantis šia schema, galima naudoti šiuos esminius statybinius blokus mūsų bibliotekų konstravimui. Su tokio tipo konstrukcija, mes galime apibrėžti, kaip modulis turėtų atrodyti. Kadangi visi trys moduliai paveldi iš to pačio sc_module bloko, jie turi tą pačią elgseną (nebent mes pasirenkame kitaip). Projektuotojas gali nuspręsti, kad konkretus programinės įrangos modulis gali būti pastebėtas sintezės analizatorių, be SystemC sintaksės ir elgsenos pakeitimų darymo, bet tai yra dalis statinės konfigūracijos.

2.6.2. Hierarchinio modulio konstrukcija SystemC aparatūrinėms bibliotekoms

Klasių bibliotekos gali labai pagelbėti tada, kai projektuojamos aparatūros bibliotekos. Štai keletas įrodytų privalumu:

a) Struktūros moduliariškumas: aparatūros savybės gali būti pakartotinai panaudotos išvestinėse klasėse.
b) Kodo vieta: kai reikia ką nors keisti, žinoma, kad tai yra klasėje arba jos prototipe.
c) Pakartotinis kodo panaudojimas: aparatūriniai įgyvendinimai bazinėse klasėse pakartotinai panaudojami išvestinėse klasėse.

12 pav. matome elgsenos hierarchiją:



12 pav. Elgsenos hierarchija

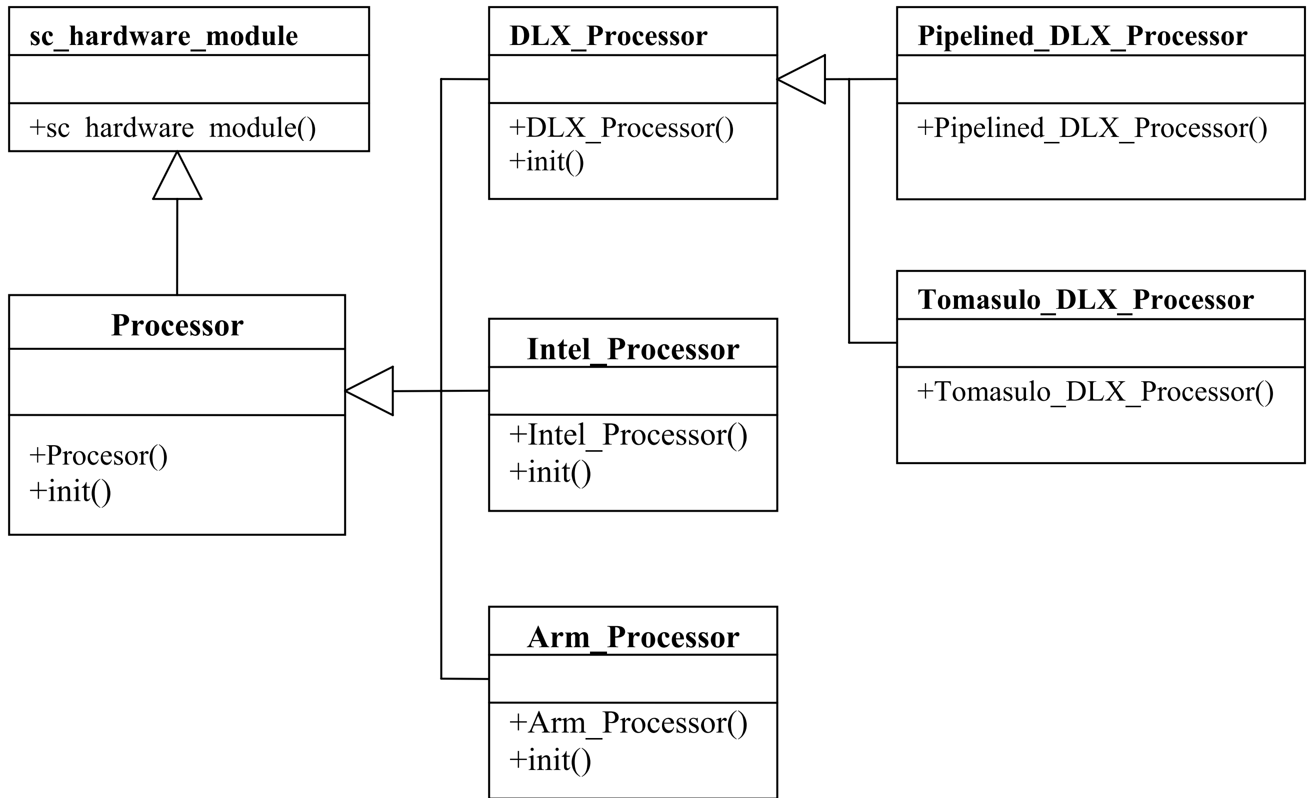
„Vaiko“ klasėje visai nereikia vėl paskelbti porto (jungties) informacijos, viskas, ką reikia padaryti, tai realizuoti metodą toms jungtims ir kiekviename konstruktoriuje metodo jautrumo sąraše įrašyti tas jungtis.

Pavyzdys:

```
And::And(...) : public Gate(...) {
  SC_METHOD (main_process)
  sensitive << input_port_1
          << input_port_2;
}
```

2.6.3. Daugialypės architektūros

Naudodami VHDL, mes galime turėti daugialypes architektūras, susijusias su ta pačia esybe. Esybę galima laikyti kaip abstrakčią klasę bei skirtingą architektūrą kaip gaunamas klases iš pagrindinės abstrakčios klasės. Reikia pažymėti, kad VHDL kalboj šis gaunamumo procesas yra skirstomas tik i du abstrakcijos lygius. SystemC kalboj šis patobulintas procesas gali pasilikti neribotas, kaip pavaizduota šiame paveiksle:



13 pav. Elgsenos subtilybė

2.6.4. Struktūros ir jų konfigūracija

VHDL kalboj „generate“ sakinys naudojamas taisyklingų struktūrų, sudarytų iš procesų ar komponentų, modeliavimui. SystemC kalboj paprastas komponentų masyvo aprašymas yra pakankamas, kaip pavaizduota žemiau.

```
Processor *processor_array[10];
processor_array[0]=new Arm_Processor();
processor_array[1]=new DLX_Processor();
processor_array[2]=new Intel_Processor();
```

Konfigūracija gali būti įdėta faile ir tada nuskaitoma vykdymo metu. „Switch“ sakinys tada gali būti panaudotas nuskaitant skirtingus procesorius. Tai neturi analogo VHDL kalboj:

```
int i=0;
FILE input = fopen("cpu.cnf", "rt");
while (!eof(input)) {
  char buffer[500];
  buffer = fgets(input);
  switch(atoi(buffer)) {
    case ARM:
  processor_array[i++] = new Arm_Processor();
  break;
    case INTEL:
  processor_array[i++] = new Intel_Processor();
  break;
    (…)
}
```

2.6.5. C++ polimorfizmas SystemC bobliotekoje

Jei mes grįžtume ir pakartotinai panagrinėtume UML elgsenos paveikslą, pavaizduotą daugialypių architektūrų skyriuje, mes pastebėtume, kad abstrakti klasė „Processor“ turi teorinį virtualų metodą init(). Šis metodas yra apibrėžtas kitame abstrakcijos lygmenyje gautose klasėse DLX_Processor, Intel_Processor bei Arm_Processor. Jei paprastai užtenka, šie metodai yra pakartotinai panaudojami trečiajame tobulinimo lygmenyje pakartotinai jų neapibrėžinėjant. Kodo moduliariškumas yra labai svarbus tam, kad pasiekti pakartotinio panaudojimo galimybes. Modulis, kuris naudoja šiuos procesorius, gali būti aprašomas naudojant tik abstrakčios klasės procesorių. Priklausomai nuo reikiamo procesoriaus atvejo, reikalingas init metodas bus sužadinamas polimorfizmo būdu:

```
for (int i = 0; i < 10; i++)
  //initialize all processors
  processor[i]->init();
```

Jei mes labai norėtumėme tokio pačio efekto VHDL kalboje, mus reikėtų specialaus signalo „Reset_Processor“, kuris turėtų maitinti tik procesoriaus komponentus. Tai yra ne taip patrauklu ir prasčiau skaitoma, nei mes apibrėžėme čia. Tai taip pat daugiau laiko reikalaujantis būdas modeliuojant, kadangi signalų įvykiai uždeda sunkia našta VHDL simuliatoriui.

2.6.6. Persidengimo mechanizmo naudojimas elgsenos pakeitimui

VHDL kalboj persidengimas yra ribotas funkcijoms ir procedūroms. SystemC kalboj ne tik metodai, bet taip pat ir klasių konstruktoriai gali būti persidengiantys. Tai leidžia dinamiškesnę fredų bei modulių konfigūraciją. Nustatant teisingo metodo iškvietimą, kompiliatorius žiūri tik į parametrų tipus, kai naudojamas metodo iškvietimas. Pateikiamas persidengiančio konstruktoriaus „Processor“ objektui pavyzdys:

```
    Processor::Processor(const sc_module_name &name, int bus_format) :
sc_hardware_module(name) {
      bus = new sc_in<int>();
      (...)
    }

    Processor::Processor(const sc_module_name &name, char bus_format) :
sc_hardware_module(name) {
    bus = new sc_in<int>();
      (...)
    }
```

2.6.7. Persidengimo naudojimas modeliavimo pagreitinimui arba apbrėžiant tam tikra elgseną

Jei VHDL kalboje mes turime apibrėžti bendrinę esybę su n portų, mes pirmiausia nusakom bendrumą ir tada paskelbiam n portų masyvą. Vis dėlto, jei tam tikram portų skaičiui mes turime konkrečią elgseną arba kokia optimizaciją, vienintelis būdas patikrinti parametrą architektūros viduje,

susijęs su bendrine esybe. SystemC kalboje naudodami persidengimą, mes galime pasiekti tai daug efektyvesniu keliu, kaip parodyta žemiau:

```
void And::compute(void); {
  Output_port = input_port_1 && input_port_2;
}

void And::compute(int n_ports); {
  if (n_ports != 0)
    for (int i = 0; i < n_ports; i++)
      (…)
}
```

2.6.8. Projektavimo šablonai

Projektavimo šablonai skirti nustatyti, dokumentuoti, kataloguoti sėkmingus programinės įrangos sprendimus. Šablonai padeda vystyti pakartotinį panaudojimą programinėj įrangoj, išreiškiant struktūrą ir komponentų bendradarbiavimą sistemos kūrėjui aukštesniu nei kodas lygiu ar nei objektiškai orientuotų projektavimo modelių lygiu, kurie remiasi individualiais objektais ir klasėmis. Šablonų panaudojimas gali būti naudojamas ir aparatūros komponentų modelių vystymui. Kaip paprastą pavyzdį pavaizduosime singletono šablono panaudojimą.

Singletono šablonas yra programinės įrangos sprendimas, kai turima viena ir tik viena atskira klasė (šis šablonas gana lengvai gali būti apibendrintas n klasėms). Šablonas draudžia vartotojui daugiau nei vieną klasę taip pat ir siunčiant įspėjimus arba grąžinant nuorodą į jau sukurtą klasę. Pagrindinė sprendimo idėja yra „paslėpti“ konstruktorių nuo viešo priėjimo, tada suteikti viešą statinį metodą valdyti modeliui.

Vienas iš šio pavyzdžio panaudojimo atvejų, kai derinant konfigūraciją, bazinė klasė turi nuspręsti, kuri iš klasių yra labiausiai naudinga modeliui.

```
Processor *Processor::get_instance(int type) {
 Switch(type) {
       Case ARM :
             Return new Arm_Processor();
             Break;
```

```
        Case INTEL :
              Return new Intel_Processor();
              Break;
  }
}
```

Problema su šio tipo pavyzdžiu yra kaip ir dažniausia būna, „vaiko“ klasė yra „žinanti“, kas darosi pagrindinėje („tėvo“) klasėje, tai tuo tarpu „tėvo“ klasė nieko nežino apie „vaikų“ klases. Šis sprendimas nėra lengvai išplečiamas: kai įdedama „vaiko“ klasė, pagrindinė klasė turi būti modifikuojama.

2.7. Išvados

Aptarėme daugialypių pavyzdžių aspektų panaudojimą SystemC kalboje kad padidinti projektavimo efektyvumo panaudojimo erdvę ir RTL, ir architektūriniame lygmenyse. Aprašytos metodologijos naudojimas leidžia lanksčių ir gerai dokumentuotų modelių vystymą. Ji taip pat padeda efektyvių testų kūrimui.

Iš objektinių aparatūros modelių galima išskirti baigtinio būsenų automato (FSM) modelį. Jis apima aprašytų elgsenos ir struktūrinio modelių savybes. FSM-based modelis apibrėžia statinius objektus su paveldimumo palaikymu, žinučių apsikeitimu ir polimorfizmu.

Iš aparatūros specifikavimo metodų, pasirinkome universalią kalbą, naudojamą įvairiose srityse – UML. Konkrečiai aparatūros aprašymui bus naudojamos klasių ir būsenų diagramos.

Aparatūros projektavimo šablonų privalumai ir trūkumai:

2 lentelė. Šablonų apibendrinimas

Šablonas	Paskirtis	Privalumai	Trūkumai	Galimybės
„Sąjungos projektavimo šablonas“ (union design pattern)	Leidžia apibendrinti keletą sistemos komponentų.	Šio šablono panaudojimas leidžia visiškai atskirti pastovius (nesikeičiančius) projektavimo problemų aspektus ir kintančius.		Tam tikri ALĮ operacijų įgyvendinimai gali būti apibrėžti nepriklausomai, taigi duodami daugiau

				lankstumo projektuotojui pasirinkti optimalų ALĮ įgyvendinimą, atsižvelgiant į reikalaujamas projektavimo charakteristikas (mikroschemos plotas, greitis ir t.t.)
„Būsenų projektavimo šablonas“ (state design pattern)	Naudojamas sistemos atskyrimui nuo jos būsenos bei funkcionalumo susijusio su šia būsena apibendrinimui skirtingose klasėse.	Galimybė valdyti sistemą yra daug didesnė, kai atskira būsena įdedama į nuosavą klasę.	Išauga projektuojamos sistemos sudėtingumas. Šio šablono realizacija veda prie blogų projektuojamos sistemos vykdymo charakteristikų	Leidžia projektuoti FSM (baigtinį būsenų automatą).
„Deimanto projektavimo šablonas“ (Diamond design pattern)	Naudojamas atvaizduoti projektavimo elgsenos sudėtingumui, kuris gaunamas naudojant keletą servisų šioje aplinkoje.	Jis leidžia gautai klasei įgyti dviejų ar daugiau „tėvo“ klasių savybes.	Gali įtakoti darbo perkrovimą, nes ne visi klasių servisai suprojektuoti naudojant paveldimumą, kuris gali būti reikalaujami projektuojamoje sistemoje.	Gali būti plačiai panaudotas aparatūroje bei įmontuotų (embedded) sistemų projektavimo plotmėje.
„Aplanko/dekoratoriaus projektavimo šablonas“ (Wrapper/decorator design pattern)	Naudojamas pridedant papildomą funkcionalumą tam tikram komponentui.	Tai įgalina didelį lankstumą ir pakartotinį panaudojimą tol, kol	Vykdomi ir tie pilnai sąsajoje aprašyti elementai, kurie nebūtinai	Galimas aparatūros komponentų bendravimas su

		projektuotojas gali pakeisti aplanką nekeisdamas jame esančių (apgaubtų) komponentų.	turėtų būti vykdomi.	savo aplinka. Klaidų toleravimo galimybė.
„Kompozicinis projektavimo šablonas“ (Composite design pattern)	Leidžia identiškai nagrinėti ir atskirus komponentus, ir komponentų rinkinius.	Pakopinis realizavimas, tai reiškia dalis sistemos gali elgtis taip pat, kaip ir visa sistema	Persidengimo atlikimo galimybė pateikta kaip rekursinė komponentų kompozicija.	Gali būti plačiai naudojamas sparčiai realizuojamų sistemų aparatūros projektavimui su pasikartojančiom architektūrom, tokiom kaip multiplekserių, registrų masyvai, sumavimo įrenginiai ir t.t.

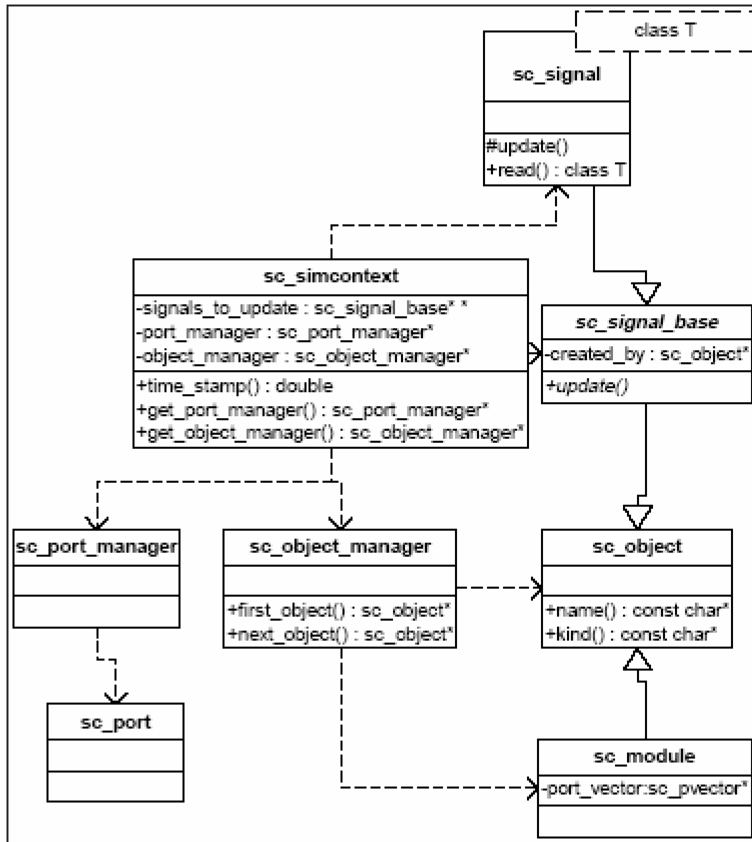
Taip pat išnagrinėjome SystemC kalbos galimybės pritaikant objektinį programavimą.

Tyrinėjant kitų autorių pasiektus rezultatus bei UML panaudojimo galimybes aparatūros projektavime, buvo nutarta:

- Aparatūros projektavimui naudoti UML klasių ir būsenų diagramas;
- Patobulinti turimą kodo generatorių, kad jis palaikytų daugumą šablonų tipų, aprašytų UML;
- Įvesti paveldimumo ir kitas objektinio programavimo savybes.
- UML klasių ir būsenų diagramų pagalba, sumodeliuoti keletą procesoriaus atliekamų veiksmų.

3. UML susiejimas su SystemC

SystemC kalbos architektūrą galima pavaizduoti tokia klasių diagrama:



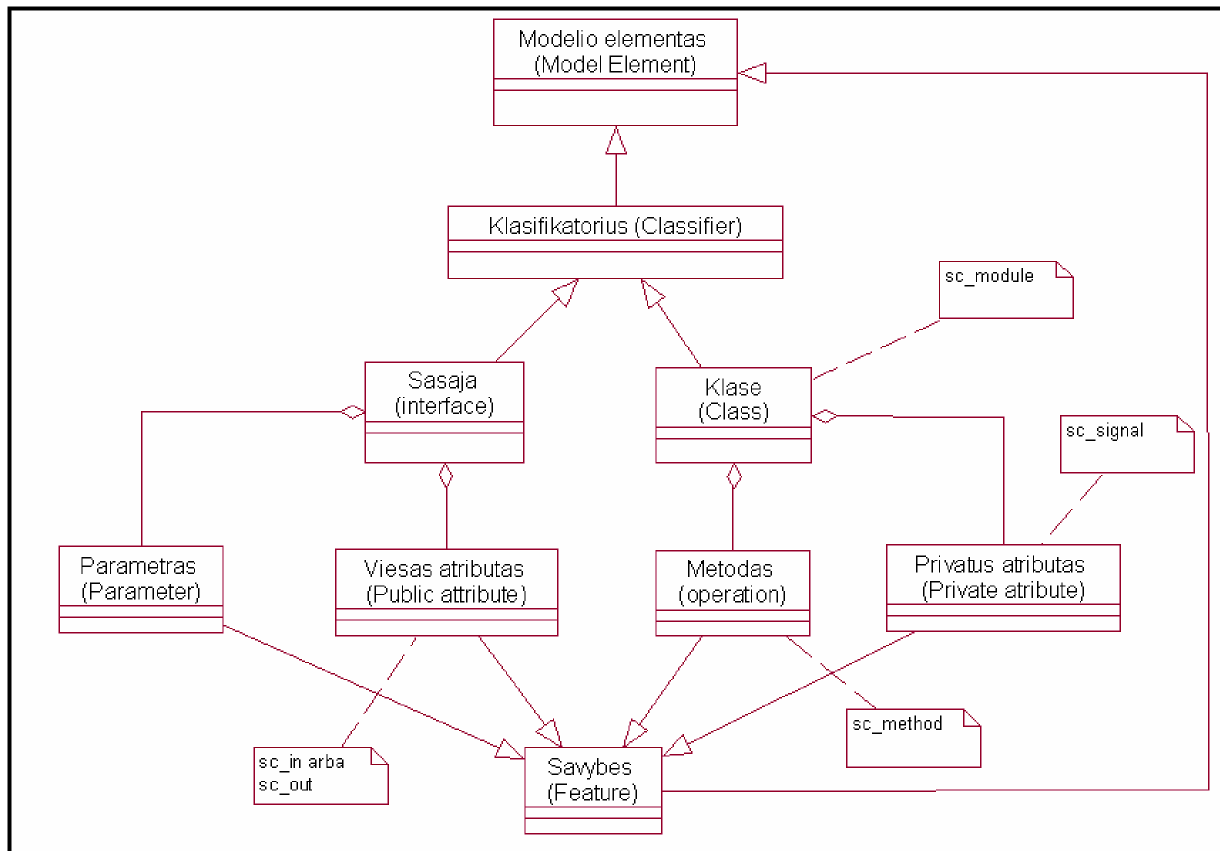
14 pav. Dalinė SystemC architektūra

Pagrindinė klasė signalams, moduliams ir sinchronizacijai yra *sc_object*, kuri turi pagrindinius metodus ir savybes identifikuojant ir klasifikuojant SystemC objektą.

sc_signal_base klasė yra *sc_object* specializacija, kuri turi pagrindinius metodus ir ryšius signalams. Tai yra abstrakti klasė. *sc_signal* paveldi *sc_signal_base* savybes. Tai yra bazinės klasės specializacija („vaikas“), kuri turi šablono tipo duomenis bei metodus, kad galėtų manipuliuoti tais duomenimis.

Klasė *sc_simcontext* yra simuliacijos branduolys.

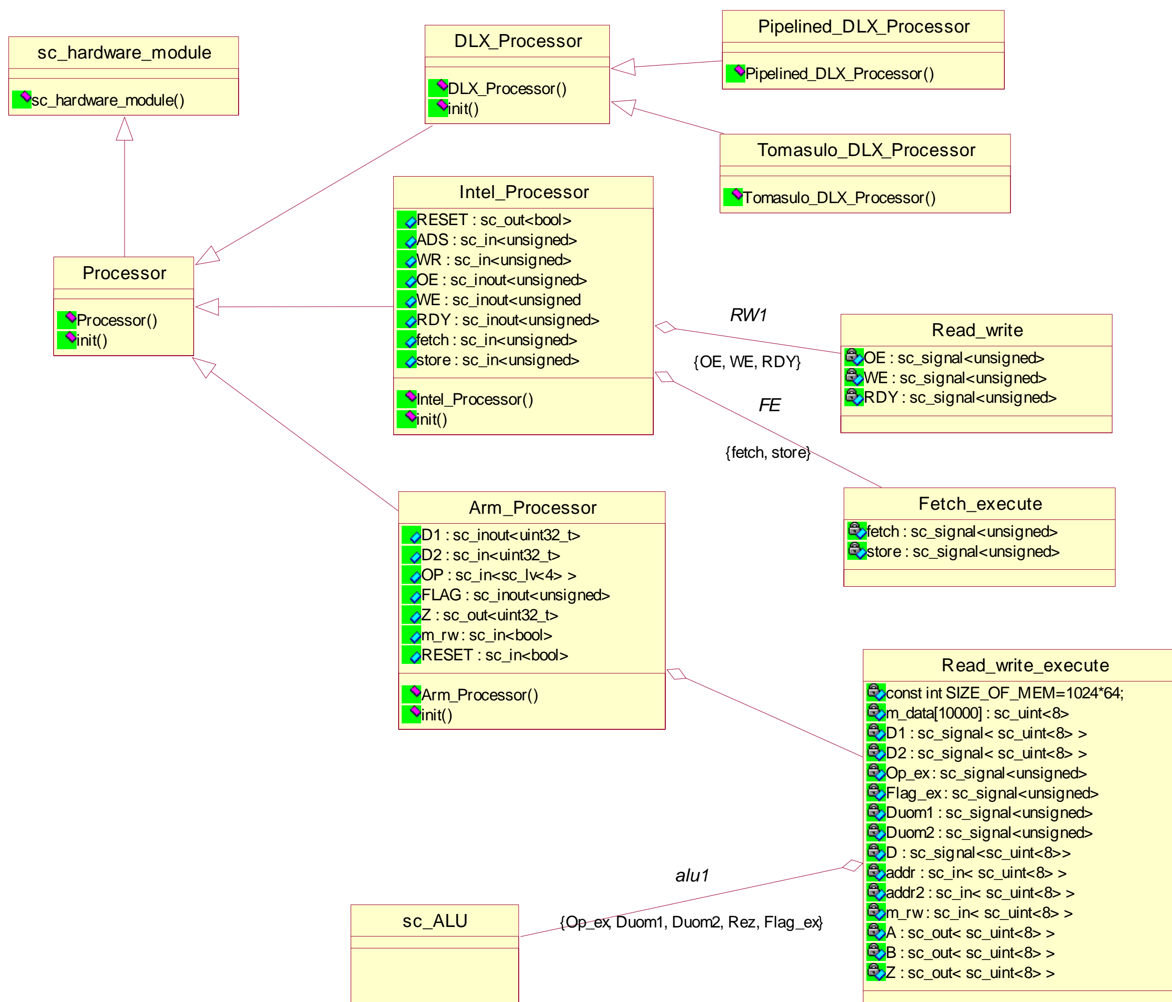
15 pav. pateikiamas UML klasių diagramų susiejimo su SystemC metamodelis:



15 pav. UML klasės susiejimo su SystemC kalba metamodelis (ryšiai)

3.1. UML klasių diagramos aprašančios procesorių

16 pav. pateikiama pagrindinė klasių diagrama, kuria remiantis buvo atliekamas tyrimas.

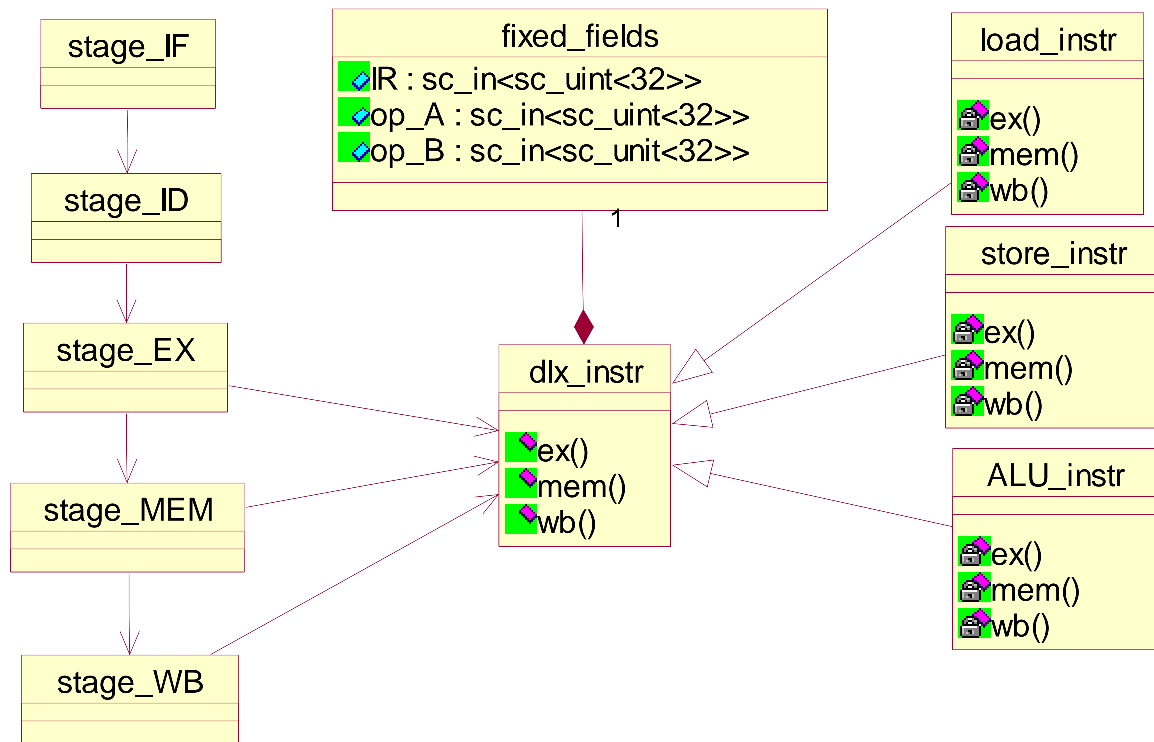


16 pav. Procesoriaus klasių diagrama

Intel_Processor klasei pasirinkau ir išskyriau dvi klases: Read_write ir Fetch_execute. Read_write klasė nuskaito ir įrašo į atmintį duomenis, o Fetch_execute klasė atlieka instrukcijų paėmimą ir atlieka veiksmus.

Kaip matome, aparatūroje galima pritaikyti tuos pačius plačiai programinėje įrangoje naudojamus projektavimo šablonus. Čia galima įžvelgti anksčiau aprašytus „Sąjungos“, „Būsenų“ projektavimo šablonų aspektus.

17 pav. detalizuota Pipelined_DLX_Processor klasių diagrama:

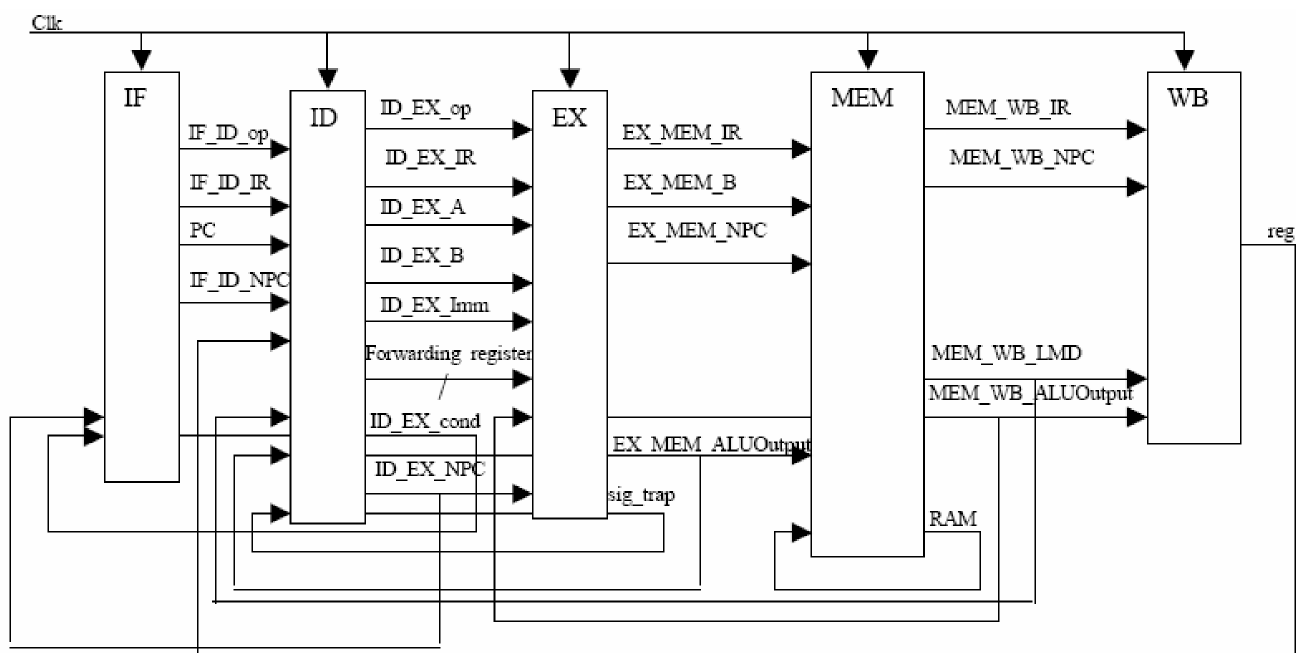


17 pav. Pipelined_DLX_Processor klasių diagrama

Matome penkias kanalų stadijas: sužadinimo (fetch), dešifravimo (decode), vykdymo, informacijos iš atminties gavimo/išrinkimo, perrašymo (write back) registrai yra atitinkamai *stage_IF*, *stage_ID*, *stage_EX*, *stage_MEM* ir *stage_WB* klasės.

Registrai tarp pakopų yra įdiegti per charakterizuotas susijusias klases ir šios klasės remiasi instrukcijom, persiduodančiom per kanalą; pradedant dešifravimo stadija. Abstraktaus pagrindo klasė *dls_instr* apibrėžia visų instrukcijų sąsają. Ji susideda iš „pastovių_laukų“. Šitie „pastovūs_laukai“ (fixed_fields) turi savyje instrukcijų registrą ir operandus. Kiekviena instrukcijų interfeiso specializacija turi ex(), mem() ir wb() metodus, kurie iškviečiami atitinkamai vykdymo, atminties ir perrašymo stadijų. Polimorfizmo mechanizmas įgalina manipuliavimą instrukcijomis su *instruction_c* pagrindine klase.

Ši klasių diagrama turi ir būsenų projektavimo šablono bruožų.



18 pav. Pipelined_DLX_Processor įgyvendinimas

Kiekvienas iš penkių modulių turi atitinkamą procesą. Realizacija tarp modulių realizuota per portus, kurie parodyti 18 pav.

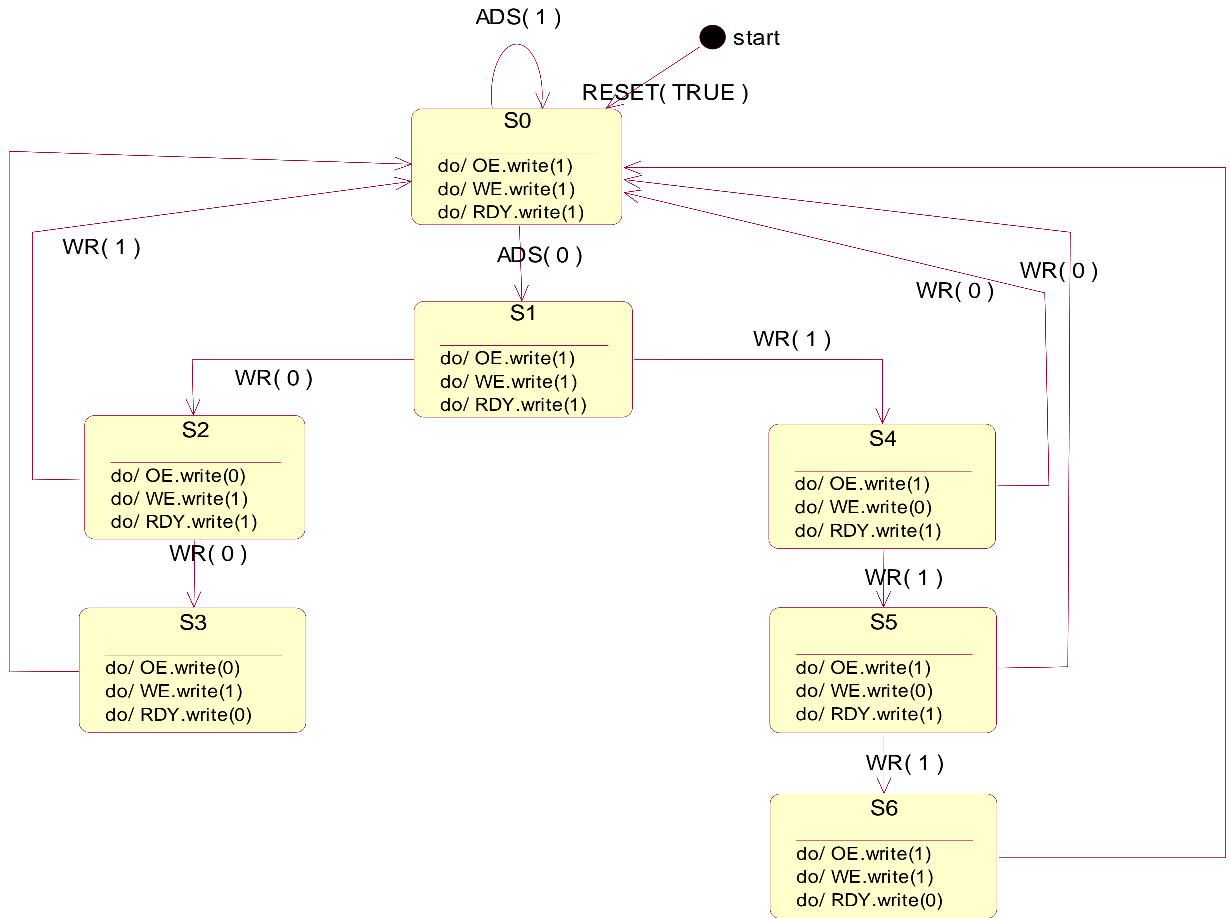
Procesai vykdomi vienu metu. Ciklas, grįžtamasis ryšys i buvusią būseną reiškia mažėjančią riziką duomenų patikimumui.

Struktūrinė rizika mažinama įterpus automatinius užlaikymus.

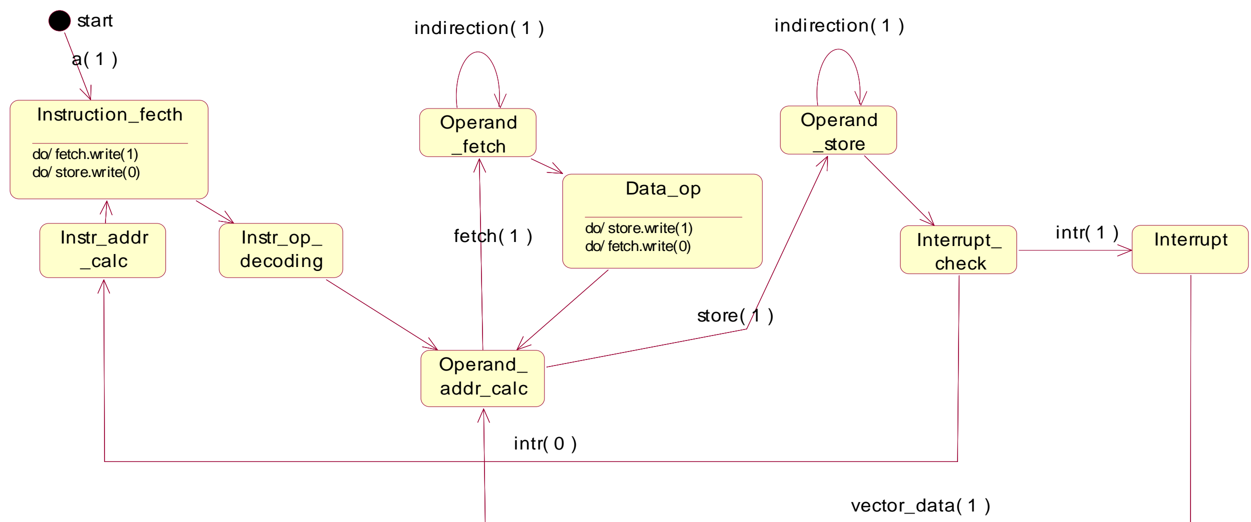
Instrukcijų rinkinį sudaro 54 instrukcijos: sveikų skaičių aritmetika, logika, perstūmimai, palyginimai, peršokimai ir atšakos, užkrovimo ir saugojimo bei kitos specializuotos instrukcijos. Programa, kurią vykdys DLX procesorius, yra faile, kurio vardas perduodamas per parametrus pagrindinei funkcijai. IF modulis naudoja tada failo turinį atminties instrukcijų gavimui. RAM yra dalis modulio MEM.

3.2. UML būsenų diagramos aprašančios procesorių

Būsenų diagramos aprašytos dviems Intel_Processor išskirtoms klasėms: Read_write ir Fetch_execute. Read_write klasės būsenų diagrama aprašo duomenų iš atminties nuskaitymą ir įrašymą; Fetch_execute – procesoriaus instrukcijos paėmimą bei įvykdymą, jei nėra pertraukimų.



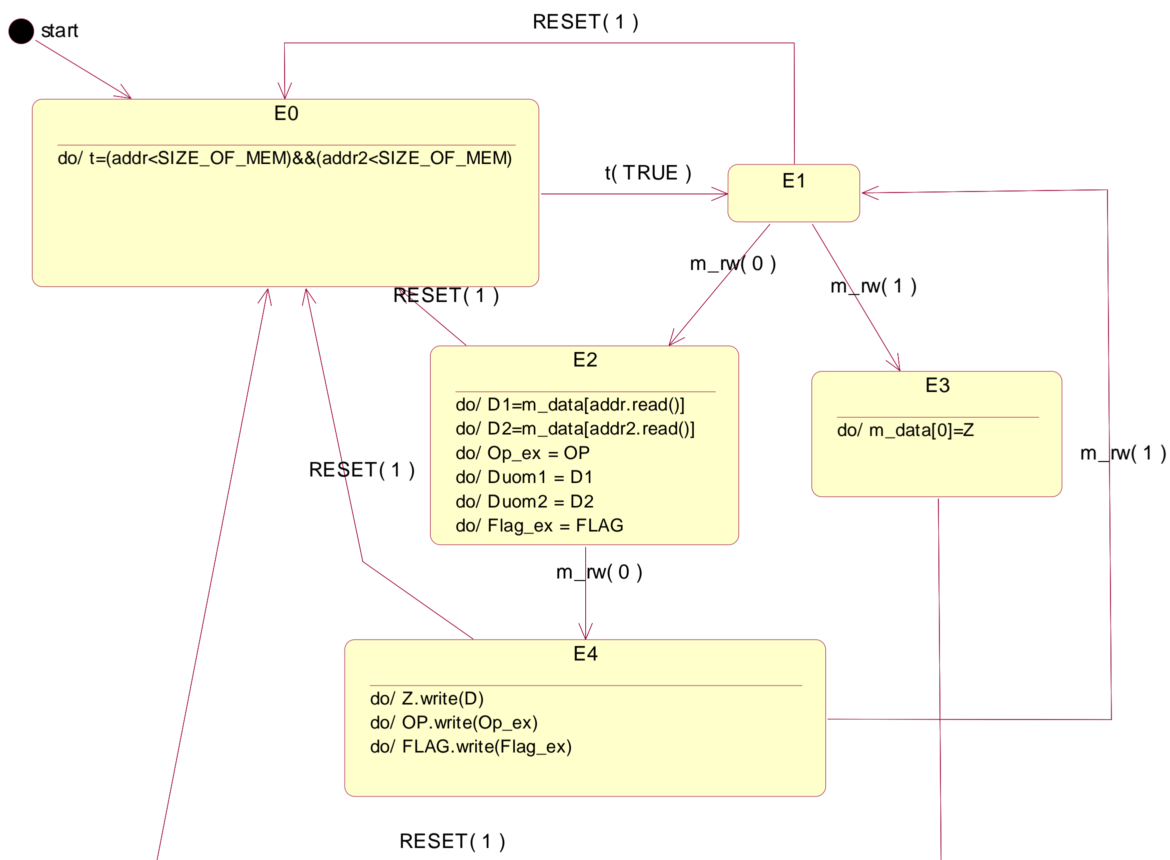
19 pav. Read_write klasės būsenų diagrama



20 pav. Fetch_execute klasės būsenų diagrama su pertraukimais

ARM procesoriaus duomenų nuskaitymas iš atminties, operacijos įvykdymas ir rezultato įrašymas į atmintį – tai Read_write_execute klasės atliekami veiksmai. Juos aprašo 21 pav. pavaizduota būsenų diagrama. Kadangi procesoriaus veiksmus pilnai aprašyti yra gana sudėtinga, pateikiamas bazinis, supaprastintas modelis.

Pagal ARM procesoriaus aprašymą, duomenis į atmintį jis turi nuskaityti iš failo. Šios funkcijos aprašymui UML pagalba priemonių neradau, todėl į pagalbą buvo pasitelktas duomenų masyvas su adresais, į kurį iš anksto surašyti duomenys. Duomenimis pasirinkti sveiki skaičiai, su kuriais buvo atliekama sumavimo operacija, ir rezultatas įrašomas į iki tol laisvą nulinę masyvo skiltį.



21 pav. Read_write_execute klasės būsenų diagrama

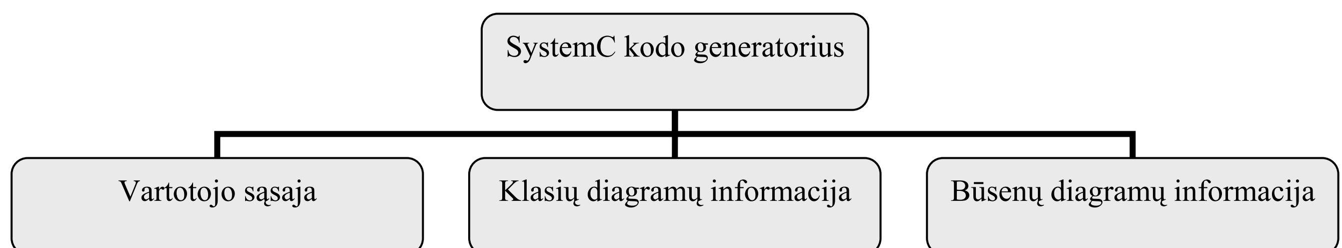
Sekančiame skyriuje, 23 pav. pateikiami modeliavimo rezultatai.

4. UML kodo generatoriaus modifikacijos ir rezultatai

Kodo generatorius sukurtas naudojantis Rational Rose programinės įrangos, kuri palaiko UML, pagalba. Šio įrankio suteikiamomis priemonėmis, galima specifikuoti kuriamą sistemą įvairiais abstrakcijos lygiais, taip pat naudoti duomenų bazių modeliavimui; palaiko automatinį UML aprašų transformavimą į populiariausias programavimo kalbas (C++, Java, Visual Basic, Ada ir kitas).

Rational Rose suteikia galimybę kurti kodą Visual Basic kalbos pagrindu ir tokiu būdu UML diagramas automatiškai transformuoti į norimos kalbos kodą. Pasinaudojus šia galimybe sukurtas automatinis generatorius, iš klasių ir būsenų diagramų generuojantis SystemC kalbos kodą (magistro darbas, Andrius Aklys, 2006).

Pats kodo generatorius, parašytas Visual Basic kalba, susideda iš atskirų funkcijų. Detalesnę informaciją galima rasti priede (priedas A.1.). Pagrindinės generatoriaus dalys yra šios:



22 pav. Kodo generatoriaus struktūra

UML kodo generatoriuje buvo atlikti tokie patobulinimai:

1. padaryta, kad paprasta („vaiko“) klasė matytų „tėvo“ klasės viešus atributus.
2. jei būsenų diagramoj nėra perėjimo, iš vienos būsenos į kitą, sąlygos (t.y. yra besąlyginis pereinantis ryšys), tai SystemC kode taip ir būtų aprašyta.
3. jei yra paveldimumo/bendrinimo (generalize) ryšys, tai duotoj klasėj būtų *include* sakinys, nurodantis „tėvo“ klasę.
4. paveldimumo/bendrinimo (generalize) ryšys realizuotas taip - *Processor.h* faile apibrėžiamas „tėvo“ klasės masyvas. Generuojant *Processor* klasės kodą, gaunam:

```
//Submodels
Processor *Processor_array[ 3 ];
```

```
//Module Constructor
SC_CTOR(Processor)
{
  Processor_array[ 0 ] = new Intel_Processor();
  Processor_array[ 1 ] = new Arm_Processor();
  Processor_array[ 2 ] = new DLX_Processor();
```

5. padaryta, kad generatorius automatiškai į .h failą įrašytų metodus, apibrėžtus UML diagramoje generuojamoje klasėse.
6. padaryta, kad būtų atpažįstamas agregacijos ryšys, kaip ir minėtas *generalize* ryšys.

Pagrindinių klasių sugeneruotus failus galite rasti priede (Priedas A.2.). Sugeneruotas SystemC kalbos kodas daugeliu atvejų negali būti iš karto modeliuojamas, nes procesoriaus struktūra yra gana sudėtinga ir kol kas ne visos funkcijos gali būti specifikuotos klasių ar būsenų diagramomis. Pvz., ARM procesoriaus atveju, pradinis duomenų įrašymas į atmintį vyksta iš failo. Taip pat pats informacijos paėmimas iš atminties ir įrašymas valdomas nemažo kiekio signalų. Kita problema, pvz., Intel procesoriaus atžvilgiu – detalaus veikimo principo aprašymo trūkumas.

Kodo generatorius naudoja tik kelias paprastas SystemC kalbos struktūras, tokias kaip *switch-case, if, paprasto priskyrimo* sakiniai, todėl vien šiais sakiniais aprašyti kokį aparatūrinį komponentą yra sudėtinga.

23 pav. pateikiami ARM procesoriaus duomenų nuskaitymo iš atminties, sumavimo ir rezultato įrašymo į atmintį modeliavimo rezultatai. Čia m_rw yra signalas, kuris nusako duomenų nuskaitymą ir įrašymą į atmintį. Kai *m_rw=0*, duomenys nuskaitomi, kai *m_rw=1*, rezultatas įrašomas į atmintį. *addr* ir *addr2* parodo, iš kurių masyvo elementų nuskaityti duomenys (*A* ir *B*), o Z yra atlikto veiksmo rezultatas.

Sumodeliuota pasitelkus Borland C++ Builder su systemc.h biblioteka.

```
             SystemC 2.0.1 --- Sep  9 2003 10:04:10
        Copyright (c) 1996-2002 by all Contributors
                    ALL RIGHTS RESERVED
m_rw= 0
addr= 3 addr2= 4 A= 5 B= 8 Z= 13
m_rw= 1
addr= 3 addr2= 4 A= 5 B= 8 Z= 13
m_rw= 0
addr= 1 addr2= 2 A= 1 B= 3 Z= 4
m_rw= 1
addr= 1 addr2= 2 A= 1 B= 3 Z= 4
```

23 pav. Read_write_execute klasės modeliavimo rezultatai

Išvados

1. Aparatūros projektavimo sudėtingumas nuolat auga. Kadangi jo mažinimui išrasti naujų priemonių nėra lengva, tai siūloma pasitelkti į pagalbą jau žinomus ir plačiai naudojamus kitose srityse, pvz., programinėje įrangoje, būdus. Pirmiausia objektinio projektavimo naudojimas. Palankiausia objektinio programavimo aparatūros aprašymo kalba yra SystemC. Ji kilus iš C++, todėl turi daug objektinio programavimo savybių. Tada galima pasitelkti į pagalbą žinomus šablonus, didinti abstrakcijos lygmenį, naudotis pakartotinio panaudojimo strategija. Taip pat kurti ir naudoti automatizuoto projektavimo priemones.
2. Aparatūros specifikavimui pasirinkta įvairiose srityse plačiai naudojama kalba UML. Jos pagalba, taikant „sąjungos projektavimo šablono“ (Procesoriaus klasių diagrama, 16 pav.), bei „būsenų projektavimo šablono“ (Pipelined_DLX_Processor klasių diagrama, 17 pav.) savybes, aprašyti kai kurie procesoriaus atliekami veiksmai klasių ir būsenų diagramomis. „Sąjungos šablono dėka“, įgyvendinamas paveldimumas, o „būsenų šablono privalumas“ – atskira sistemos būsena įdedama į atskirą klasę. Galima teigti, kad UML pagalba, pasinaudojant žinomais šablonais, galima gana išsamiai aprašyti procesorius, jų tipus ir veikimo principus.
3. Atlikti patobulinimai turėtam kodo generatoriui, sukurtam Rational Rose įrankio pagalba, naudojant Rose Script kalbą. Aprašytas generalize ryšys, suteikiantis paveldimumo funkciją, pakoreguoti perėjimo tarp būsenų aprašai.
4. Sumodeliuotas ARM procesoriaus duomenų iš atminties nuskaitymas, sumavimo operacijos atlikimas bei rezultato įrašymas į atmintį.

Literatūra

1. R. Damaševičius, V. Štuikys, Application of object-oriented principles for hardware and ambedded system design, Integration, the VLSI journal, 2004.
2. L. Charest and E.M. Aboulhamid, A VHDL/SystemC Comparison in Handling Design Reuse. Universite de Montreal.
3. Open SystemC Initiative (OSCI), Functional Specification for SystemC 2.0, [žiūrėta 2007 03 20], http://www.systemc.org
4. Andrius Aklys, Aprašų transformacijos į srities kalbą (VHDL/SystemC), magistro darbas, KTU Informatikos fakultetas, 2006.
5. H. Chang, L. Cooke, M. Hunt, G. Martin, A. McNelly, L. Todd, Surviving the SOC Revolution: A Guide to Platform-Based Design, Kluwer Academic Publishers, Dordrecht, 1999.
6. SEMATECH, The International Technology Roadmap for semiconductors, 2001.
7. Schattkowsky Tim, (2005) UML 2.0 - Overview and Perspectives in SoC Design, pp. 832-833, Design, Automation and Test in Europe (DATE'05) Volume 2, .
8. F. Vahid, T. Givargis, Embedded System Design: A Unified Hardware/Software Introduction, Wiley, New York, 2002.
9. T. Potok, M. Vouk, Development productivity for commercial software using object-oriented methods, Proceedings of the 1995 Conference of the Centre for Advanced Studies on Collaborative Research, IMB Press, Toronto, Ontario, Canada, 1995, p. 52.
10. M. Radetzki, Synthesis of digital circuits from object-oriented specifications, PH.D. Thesis, Universitat Oldenburg, Germany, 2000.
11. R. Allen, D. Gajski, The case for C/C++ hardware design, EETimes, 2000.
12. C. Sculz-Key, M. Winterholer, T. Schweizer, T. Kuhn, W. Rosenstiel, Object-oriented modeling and synthesis of SystemC specifications, in: Proceedings of the Asia South Pasific Design Automation Conference (ASPDAC'04), Yokohama, Japan, 2004, pp. 238-243.
13. Sangiovanni-Vincentelli, G. Martin, A vision for embedded systems: platform-based design and software methology, IEEE Des. Test Comput. 18 (6) (2001) 23-33.
14. M. Guodarzi, S. Hessabi, A.Mycroft, Object-oriented ASIP design and synthesis, in: Forum on Specification and Design Languages (FDL'03), Frankfurt, Germany, September 23-26, 2003.
15. Silicon Infusion Ltd., An Object Oriented Re-Configurable System on Chip, White paper, 2003.

16. J.P. Peterson, Petri Net Theory and the Modeling of Systems, Prentice-Hall, Englewood cliffs, NJ, 1981.
17. C.A. Lakos, C.D. Keen, LOOPN++: a new language for object-oriented Petri nets, Proceedings of the Modelling and Simulation (European Simulation Multiconference), Barselona, Spain, 1994, pp. 369-374.
18. R.J. Machado, J.M. Fernandes, H.D. Santos, a methodology for complex embedded systems design: petri nets within a UML approach, in: B. Kleinjohann (Ed.), Architecture and Design of Distributed Embedded Systems, Kluwer Academic Publishers, Dordrecht, 2001, pp. 1-10.
19. Liccardi, T. Maier-Komor, J.A. Oswald, M. Elkotob, G. Farber, A meta-modeling concept for embedded RT-systems design, in: 14th Euromicro Conference on Real-Time Systems, Vienna, Austria, 19-21 June 2002.
20. R. Damaševičius, V. Štuikys, Application of UML for hardware design based process model, Proceedings of the Asia South Pacific Design Automation Conference (ASP-DAC 2004), Yokohama, Japan, January 27-30, 2004, pp. 244-249.
21. E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, MA, 1995.
22. R. Damaševičius, G. Majauskas, V. Štuikys, Application of design patterns for hardware design, Proceedings of the 40th Design Automation Conference (DAC 2003), Anaheim, CA, USA, June 2-6, 2003, pp. 48-53.
23. William Stallings, Computer Orgaanization and Architekture, 6 edition, [Žiūrėta 2006 02 15], <www.elet.polimi.it/upload/fornacia/didattica/org_calc0304/**03OrgCalc_Ch_3_Buses**.pdf >
24. Interfacing the Byte-Wide SmartVoltage FlashLive Memory Family to the Intel486 Microprocessor Family, [Žiūrėta 2006 02 17], <ftp://download.intel.com/design/intarch/technote/**29780501**.PDF >
25. Interfacing Memory System to the ARM7TDM Without using AMBA, [Žiūrėta 2007 03 14], <www.arm.com/pdfs/**DAI0029A_arm7tdm_mem_intf**.pdf >
26. Polymorphic C++ Debugging for System Design, [Žiūrėta 2007 04 02], <www.ics.uci.edu/~balboa/pubs/**UCI-CECS-TR00-06**.pdf >

Priedai

A.1. Generatoriaus detalizacija.

3 lentelė. Kodo generatoriaus architektūra

Function ReportDialogLoop(controlname$, action%, suppvalue%) As Integer Function MakeFileName (Path As String, FileName As String) As String Function ChangeFileExtension (FullFileName As String, NewExtension As String) As String Function CreateFilePath (FilePath As String, FileName As String, Extension As String) As String Function GetUserInput (eror As Integer, count As Integer) As String Begin Dialog UserDialog ,,244,128,"SystemC Code Generation",.ReportDialogLoop	Funkcijos, kurios aprašo vartotojo sąsają	Vartotojo sąsaja
Function GetType_States (theClass As Class) As String	Funkcija generuoja enum state_t sąrašą.	
Function StartOfHFile (theName As String) As String	.h failo pradžia	
Function GetInitialState (aClass As Class) As State	Randama pradinė būsena	
Function ExistEventName (events As EventCollection , event As Event) As Boolean	Tikrinama, ar įvykis egzistuojantis	
Function GetFinalState (aClass As Class) As State	Grąžinama galinė būsena	
Function IsInitialState (theState As State) As Boolean	Tikrinama, ar būsena pradinė	
Function IsFinalState (theState As State) As Boolean	Tikrinama, ar būsena galinė	
Function GetAction (aClass As Class, theState As State) As String	Gaunama perėjimo iš vienos būsenos į kitą veiksmo vardas	
Function GeneratesWhenThis (theState As State ,aClass As Class) As String	Generuojamas perėjimas į kitą būseną	
Function NextStateAndOutput (theName As String , aClass As Class) As String	Generuojamas kodas, aprašantis perėjimus tarp būsenų.	

Function GetActionM (aClass As Class, theState As State) As String Function GeneratesWhenAM (theState As State, aClass As Class) As String	Esat būsenoj atliekami veiksmai	Kodo generavimas iš būsenų diagramų
Function RealizationUpdateState (theName As String) As String	Visada pastovus update_state metodas	
Function StartOfCppFile (theName As String) As String Function EndOfHFile (theName As String) As String Function ModuleBegin (theName As String) As String Function InputOutput As String Function StateClassConstructor (theDiagram As StateDiagram) As String	Reikelingo SystemC failų kodo aprašai	
Function IsComponentOf (thisClass As Class, theTargetClassName As String) As Boolean	Tikrinama, ar yra koks ryšys su kitom klasėm	
Function GenerateAttributes (currentClass As Class) As String	Gaunami atributai	
Function OperationsDeclaration (currentClass As Class) As String	Gaunami metodai	
Function GenerateSensitiveBlock (theEvents As EventCollection) As String	Sudaromas jautrumo sąrašas	
Function GetOperationFromDiagram (theStateDiagram As StateDiagram, currentClass As Class) As String	Paskelbiami nekintantys metodai su jautrumo sąrašais: ("SC_METHOD(update_state);" „SC_METHOD(ns_logic);" "SC_METHOD(op_logic);")	
Function GenerateAttributesC (currentClass As Class) As String	Atributų aprašymas	
Function GetIncludeFiles (theClass As Class, theClasses As ClassCollection) As String	Formuojamas include sąrašas	
Function GetAssociationName (component As Class,	Randamas ryšio vardas.	

architecture As Class) As String		Kodo generavimas iš klasių diagramose pateiktos informacijos
Function IsRealizeOf (thisClass As Class, theEntityName As String) As Boolean	Tikrinama, ar theEntityName klasė yra thisClass klasės architektūra	
Function FromRealized (theClass As Class, theClasses As ClassCollection) As String	Gaunami abstrakčios klasės atributai	
Function SubModels (theClass As Class, theClasses As ClassCollection) As String Function SubModelsInit (theClass As Class, theClasses As ClassCollection) As String	Randamos nagrinėjamos klasės „vaikų“ klasės ir paskelbiamos kaip tėvo klasės masyvo elementai. Taip pat aprašomas agregacijos ryšys, jei yra.	
Function GetAssociationConstraint (component As Class, architecture As Class) As String	Gaunami ryšiu perduodami parametrai	
Function GenerateClassConstructor (currentClass As Class, theClasses As ClassCollection) As String	Aprašomas klasės konstruktorius	
Sub GenerateModule(theClass As Class, theClasses As ClassCollection)	Sujungiamos funkcijos.	

A.2. Sugeneruotas kodas SystemC kalba

Processor.h

```
//---- file : Processor.h

#include "systemc.h"

#include "sc_hardware_module.h"



SC_MODULE(Processor)
{
```

```
  //Submodels
  Processor *Processor_array[ 3 ];

  //Module Constructor
  SC_CTOR(Processor)
  {
    Processor_array[ 0 ] = new Intel_Processor();
    Processor_array[ 1 ] = new Arm_Processor();
    Processor_array[ 2 ] = new DLX_Processor();


  }

  void Processor();

  void init();


};//---- end of class Processor
```

DLX_Processor.h

```
//---- file : DLX_Processor.h

#include "systemc.h"

#include "Processor.h"




SC_MODULE(DLX_Processor)
{






  //Submodels
  DLX_Processor *DLX_Processor_array[ 2 ];

  //Module Constructor
  SC_CTOR(DLX_Processor)
  {
    DLX_Processor_array[ 0 ] = new Tomasulo_DLX_Processor();
    DLX_Processor_array[ 1 ] = new Pipelined_DLX_Processor();


   }

  void DLX_Processor();

  void init();


};//---- end of class DLX_Processor
```

Read_write.h

```
//---- file : Read_write.h

#include "systemc.h"
#include "Intel_processor.h"


SC_MODULE(Read_write)
{
  sc_out<bool> RESET;
  sc_in<unsigned> ADS;
  sc_in<unsigned> WR;
  sc_inout<unsigned> OE;
  sc_inout<unsigned WE;
  sc_inout<unsigned> RDY;
  sc_in<unsigned> fetch;
  sc_in<unsigned> store;



  sc_signal<unsigned> OE;
  sc_signal<unsigned> WE;
  sc_signal<unsigned> RDY;




  //defining the states
  enum state_t {start ,S0 ,S1 ,S2 ,S3 ,S4 ,S5 ,S6 };
  sc_signal<state_t> state, next_state;

  //Module Constructor
  SC_CTOR(Read_write)
  {


    SC_METHOD(update_state);
    sensitive_pos << clk;
    SC_METHOD(ns_logic);
    sensitive << state << RESET << ADS << WR;
    SC_METHOD(op_logic);
    sensitive << state << RESET << ADS << WR;

  }

  void update_state();
  void ns_logic();
  void op_logic();

};//---- end of class Read_write
```

Read_write.cpp

```
//---- file : Read_write.cpp
```

```
#include <systemc.h>
#include "Read_write.h"


void Read_write::update_state() {
  state = next_state;
}


void Read_write::ns_logic() {
  switch(state) {
    case S0:

         if( ADS.read() = 1 ) {
             next_state = S0;
         } else if( ADS.read() = 0 ) {
             next_state = S1;
         } else {
             next_state = S0;
         };
         break;
    case S1:

         if( WR.read() = 1 ) {
             next_state = S4;
         } else if( WR.read() = 0 ) {
             next_state = S2;
         } else {
             next_state = S1;
         };
         break;
    case S2:

         if( WR.read() = 0 ) {
             next_state = S3;
         } else if( WR.read() = 1 ) {
             next_state = S0;
         } else {
             next_state = S2;
         };
         break;
    case S3:

             next_state = S0;
         break;
    case S4:

         if( WR.read() = 1 ) {
             next_state = S5;
         } else if( WR.read() = 0 ) {
             next_state = S0;
         } else {
             next_state = S4;
         };
         break;
    case S5:

         if( WR.read() = 1 ) {
```

```
            next_state = S6;
        } else if( WR.read() = 0 ) {
            next_state = S0;
        } else {
            next_state = S5;
        };
        break;
    case S6:

            next_state = S0;
        break;
  }
}

void Read_write::op_logic() {
  switch(state) {


    case S0:
         OE.write(1);
         WE.write(1);
         RDY.write(1);
         break;

    case S1:
         OE.write(1);
         WE.write(1);
         RDY.write(1);
         break;

    case S2:
         OE.write(0);
         WE.write(1);
         RDY.write(1);
         break;

    case S3:
         OE.write(0);
         WE.write(1);
         RDY.write(0);
         break;

    case S4:
         OE.write(1);
         WE.write(0);
         RDY.write(1);
         break;

    case S5:
         OE.write(1);
         WE.write(0);
         RDY.write(1);
         break;

    case S6:
         OE.write(1);
         WE.write(1);
         RDY.write(0);
         break;  }
```

```
}
```

fetch_execute.h

```
//---- file : Fetch_execute.h

#include "systemc.h"
#include "Intel_processor.h"




SC_MODULE(Fetch_execute)
{
  sc_out<bool> RESET;
  sc_in<unsigned> ADS;
  sc_in<unsigned> WR;
  sc_inout<unsigned> OE;
  sc_inout<unsigned WE;
  sc_inout<unsigned> RDY;
  sc_in<unsigned> fetch;
  sc_in<unsigned> store;


  sc_signal<unsigned> fetch;
  sc_signal<unsigned> store;




  //defining the states
  enum state_t {start ,Instruction_fecth ,Instr_addr_calc ,Instr_op_decoding
,Operand_fetch ,Operand_addr_calc ,Data_op ,Operand_store ,Interrupt_check ,Interrupt
};
  sc_signal<state_t> state, next_state;

  //Module Constructor
  SC_CTOR(Fetch_execute)
  {


    SC_METHOD(update_state);
    sensitive_pos << clk;
    SC_METHOD(ns_logic);
    sensitive << state << a << indirection << fetch << store << intr << vector_data;
    SC_METHOD(op_logic);
    sensitive << state << a << indirection << fetch << store << intr << vector_data;

  }

  void update_state();
  void ns_logic();
  void op_logic();

};//---- end of class Fetch_execute
```

Fetch_execute.cpp

```
//---- file : Fetch_execute.cpp

#include <systemc.h>
#include "Fetch_execute.h"


void Fetch_execute::update_state() {
  state = next_state;
}


void Fetch_execute::ns_logic() {
  switch(state) {
    case Instruction_fecth:

            next_state = Instr_op_decoding;
          break;
    case Instr_addr_calc:

            next_state = Instruction_fecth;
          break;
    case Instr_op_decoding:

            next_state = Operand_addr_calc;
          break;
    case Operand_fetch:

            next_state = Data_op;
          } else if( indirection.read() = 1 ) {
            next_state = Operand_fetch;
          } else {
            next_state = Operand_fetch;
          };
          break;
    case Operand_addr_calc:

          if( fetch.read() = 1 ) {
            next_state = Operand_fetch;
          } else if( store.read() = 1 ) {
            next_state = Operand_store;
          } else {
            next_state = Operand_addr_calc;
          };
          break;
    case Data_op:

            next_state = Operand_addr_calc;
          break;
    case Operand_store:

            next_state = Interrupt_check;
          } else if( indirection.read() = 1 ) {
            next_state = Operand_store;
          } else {
            next_state = Operand_store;
          };
          break;
```

```
    case Interrupt_check:

         if( intr.read() = 1 ) {
             next_state = Interrupt;
         } else if( intr.read() = 0 ) {
             next_state = Instr_addr_calc;
         } else {
             next_state = Interrupt_check;
         };
         break;
    case Interrupt:

         if( vector_data.read() = 1 ) {
             next_state = Operand_addr_calc;
         } else {
             next_state = Interrupt;
         };
         break;
  }
}

void Fetch_execute::op_logic() {
  switch(state) {


    case Instruction_fecth:
         fetch.write(1);
         store.write(0);
         break;

    case Instr_addr_calc:
         break;

    case Instr_op_decoding:
         break;

    case Operand_fetch:
         break;

    case Operand_addr_calc:
         break;

    case Data_op:
         store.write(1);
         fetch.write(0);
         break;

    case Operand_store:
         break;

    case Interrupt_check:
         break;

    case Interrupt:
         break;  }
}
```

Read_write_execute.h

```
//---- file : Read_write_execute.h

#include "systemc.h"

#include "Read_write_execute.h"
#include “scALU.h”


const int SIZE_OF_MEM=1024*64;

SC_MODULE(Read_write_execute)
{


  sc_in<sc_lv<4> > OP;
  sc_inout<unsigned> Flag;
  sc_uint<8> m_data[10000];
  sc_signal< sc_uint<8> > D1;
  sc_signal< sc_uint<8> > D2;
  sc_signal<unsigned> Op_ex;
  sc_signal<unsigned> Flag_ex;
  sc_signal<unsigned> Duom1;
  sc_signal<unsigned> Duom2;
  sc_signal<sc_uint<8>> D;
  sc_in< sc_uint<8> > addr;
  sc_in< sc_uint<8> > addr2;
  sc_in< sc_uint<8> > m_rw;
  sc_out< sc_uint<8> > A;
  sc_out< sc_uint<8> > B;
  sc_out< sc_uint<8> > Z;



  scALU *alu1;

  //defining the states
  enum state_t {start ,E0 ,E1 ,E2 ,E3 ,E4 };
  sc_signal<state_t> state, next_state;

  //Module Constructor
  SC_CTOR(Read_write_execute)
  {

    alu1 = new scALU ("alu1");
      (*alu1)(Op_ex, Duom1, Duom2, Rez, Flag_ex);

    SC_METHOD(update_state);
    sensitive_pos << clk;
    SC_METHOD(ns_logic);
    sensitive << state << t << RESET << m_rw;
    SC_METHOD(op_logic);
    sensitive << state << t << RESET << m_rw;

  }

  void update_state();
  void ns_logic();
```

```
  void op_logic();

};//---- end of class Read_write_execute
```

Read_write_execute.cpp

```
//---- file : Read_write_execute.cpp

#include <systemc.h>
#include "Read_write_execute.h"


void Read_write_execute::update_state() {
  state = next_state;
}


void Read_write_execute::ns_logic() {
  switch(state) {
    case E0:

         if( t.read() == TRUE ) {
             next_state = E1;
         } else {
             next_state = E0;
         };
         break;
    case E1:

         if( RESET.read() == 1 ) {
             next_state = E0;
         } else if( m_rw.read() == 0 ) {
             next_state = E2;
         } else if( m_rw.read() == 1 ) {
             next_state = E3;
         } else {
             next_state = E1;
         };
         break;
    case E2:

         if( m_rw.read() == 0 ) {
             next_state = E4;
         } else if( RESET.read() == 1 ) {
             next_state = E0;
         } else {
             next_state = E2;
         };
         break;
    case E3:

         if( RESET.read() == 1 ) {
             next_state = E0;
         } else {
             next_state = E3;
         };
```

```
          break;
     case E4:

          if( RESET.read() == 1 ) {
              next_state = E0;
          } else if( m_rw.read() == 1 ) {
              next_state = E1;
          } else {
              next_state = E4;
          };
          break;
  }
}

void Read_write_execute::op_logic() {
  switch(state) {


     case E0:
          t=(addr<SIZE_OF_MEM)&&(addr2<SIZE_OF_MEM);
          break;

     case E1:
          break;

     case E2:
          D1=m_data[addr.read()];
          D2=m_data[addr2.read()];
          Op_ex = OP;
          Duom1 = D1;
          Duom2 = D2;
          Flag_ex = FLAG;
          break;

     case E3:
          m_data[0]=Z;
          break;

     case E4:
          Z.write(D);
          OP.write(Op_ex);
          FLAG.write(Flag_ex);
          break;
   }
}
```