KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
MULTIMEDIJOS INŽINERIJOS KATEDRA

Andrius Aklys

UML aprašų transformacijos į srities kalbą (VHDL, SystemC)

Magistro darbas

Darbo vadovas
Dr. Robertas Damaševičius

Kaunas, 2006

KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
MULTIMEDIJOS INŽINERIJOS KATEDRA

Andrius Aklys

UML aprašų transformacijos į srities kalbą (VHDL, SystemC)

Magistro darbas

Kalbos konsultantė
Lietuvių k. katedros lekt.
I. Mickienė
2006-05

Vadovas
dr. R. Damaševičius
2006-05

Recenzentas
doc. dr. G. Ziberkas
2006-05

Atliko
IFM-0/1 gr. stud.
Andrius Aklys
2006-05-29

Kaunas, 2006

Santrauka

Elektroninės įrangos projektavimo produktyvumui padidinti siūlome naudoti aukšto lygmens sistemų projektavimo kalbą UML. Aukštesnis abstrakcijos lygmuo ir automatizuoto projektavimo metodai leistų sumažintį aparatūrinės įrangos projektavimo našumo atsilikimą. Elektroninių sistemų struktūros projektavimui siūlome naudoti UML klasių diagramas, o elgsenos specifikavimui – UML būsenų diagramas. Pristatydami metamodelius, kurie leidžia susieti UML klasių ir būsenų diagramas su aparatūros aprašymo kalbomis VHDL ir SystemC, siekiame parodyti vieną iš galimų būdų pasiūlytoms idėjoms realizuoti. Taip pat pateikiame sukurtą kodo generatorių, kuris automatiškai transformuoja UML klasių ir būsenų diagramų aprašus į VHDL ir SystemC kalbas.

Abstract

Transformation of UML notations to domain language (VHDL,SystemC) it is work, how to increase the productivity of electronic systems design. We offer to use UML – the standard specification language of high level systems. The higher level of abstraction and automatic design methods could decrease a gap of hardware design. We offer to use UML class diagrams for the specification of electronic systems structure and UML state diagrams to specify the behavior of electronic systems. We introduce *metamodels* which describe mapping between UML class and state diagrams and hardware description languages (VHDL, SystemC), as the possible realization of ideas we introduced earlier. Also we provide code generator which translates notations of UML class and state diagrams to VHDL and SystemC languages.

Turinys

1. Įvadas ... *7*

1.1. UML naudojimas aparatūros projektavime ... 7
1.2. Projektavimo privalumai naudojant UML kalbą ... 8
1.3. Magistro darbo tikslai ... 8
1.4. Apžvalga ... 9

2. UML kalba ir programų generatoriai ... *9*

2.1. UML panaudojimo galimybės ... 9
2.2. UML įrankių apžvalga ... 10
2.3. Programų generatoriai ... 11
2.4. Programų generatorių tipai ... 12
2.4. Kodo generatorių privalumai ir trūkumai ... 15

3. Analizė ... *16*

3.1 Srities inžinerija ... 16
3.2 Aparatūros aprašymo kalbos ... 17
3.2.1 VHDL apžvalga ... *17*
3.2.2 SystemC apžvalga ... *19*
3.3. Kitų autorių darbų apžvalga ... 21
3.4. Išvados ... 22

4. UML susiejimas su VHDL ir SystemC ... *23*

4.1. UML klasių diagrama ir aparatūros aprašymo kalbos ... 23
4.1.1 UML klasių diagrama ir VHDL ... *23*
4.1.2 UML klasių diagrama ir SystemC ... *26*
4.2 UML būsenų diagrama ir aparatūros aprašymo kalbos ... 28
4.2.1 UML būsenų diagrama ir VHDL ... *30*
4.2.2 UML būsenų diagrama ir SystemC ... *33*

5. Kodo generatoriaus testavimas ... *36*

5.1. Kodo generatoriaus apžvalga ... 36
5.2. VHDL kodo generatoriaus tyrimas ... 37
5.3. SystemC kodo generatoriaus tyrimas ... 39
5.4. Atliktų rezultatų suvestinė ... 42
5.5. Išvados ir įvertinimas ... 42

6. Bendrosios išvados ir tolesni darbai ... *44*

6.1 Išvados ... 44
6.2 Tolesni darbai ... 44

Literatūra ... *46*

Priedai ... *48*

Lentelių sąrašas

1. Pagrindinės VHDL sąvokos.. 18
2. SystemC sąvokos 20
3. VHDL generavimas iš UML klasių diagramos………………………26
4. SystemC generavimas iš UML klasių diagramos…………………….28
5. Modelių ir programų tekstų charakteristikos..............……………...... 43

Paveikslų sąrašas

1 pav. Generatoriaus sandara..13
2 pav. Sistemos bazinė architektūra 14
3 pav. UML klasių diagramos susiejimo su VHDL kalba metamodelis (ryšiai)..…...24
4 pav. UML klasių diagramos susiejimo su VHDL kalba metamodelis (bruožai)….24
5 pav. UML klasių diagramos susiejimo su SystemC kalba metamodelis (ryšiai).…26
6 pav. UML klasių diagramos susiejimo su SystemC kalba metamodelis (bruožai)...27
7 pav. Mili(a) ir Muro(b) automatai aprašyti UML būsenų diagrama……………….29
8 pav. Automato struktūra..30
9 pav. UML būsenų diagramos susiejimo su VHDL metamodelis.............................31
10 pav. Muro automato būsena ir VHDL kodas...32
11 pav. Mili automato būsena ir VHDL kodas...32
12 pav. UML būsenų diagramos susiejimo su SystemC metamodelis........................34
13 pav. Muro automato būsena ir SystemC kodas..35
14 pav. Mili automato būsena ir SystemC kodas..35
15 pav. Generatoriaus struktūra…..36
16 pav. Kodo generatoriaus naudojimas..37
17 pav. Handshake klasių diagrama (VHDL)…...38
18 pav. Handshake būsenų diagrama (VHDL)…...38
19 pav. Handshake įrenginio laikinė diagrama (VHDL)..39
20 pav. Handshake įrenginio skaitinės reikšmės (VHDL)..39
21 pav. Handshake klasių diagrama (SystemC)...40
22 pav. Handshake būsenų diagrama (SystemC)...40
23 pav. Handshake įrenginio laikinė diagrama (SystemC)...41
24 pav. Handshake įrenginio skaitinės reikšmės (SystemC).......................................41

1. Įvadas

1.1. UML naudojimas aparatūros projektavime

Šiandien elektroninių sistemų sudėtingumas nuolat auga, kadangi rinka reikalauja vis greitesnių, pigesnių, mažesnių ir „protingesnių“ elektroninių produktų. Projektavimo produktyvumo atsilikimas tapo didžiausia problema elektroninių sistemų pramonėje. Programinės įrangos inžinerija taip pat susiduria su panašiomis problemomis, tačiau pastebima, kad susidoroti su sunkumais padeda abstraktesnių metodų taikymas projektavime. UML kalba tapo standartu programinių sistemų modeliavime ir paskatino naujų projektavimo metodų atsiradimą. UML naudą pastebi ir kitų sričių atstovai. Ji jau naudojama verslo procesų modeliavime, fizikoje. Manoma, kad ir aparatūrinės įrangos projektavime gali būti pritaikyta UML modeliavimo kalba (Schattkowsky, 2005).

Aukštesnio lygio kalbų taikymo galimybėmis elektroninių sistemų projektavime susidomėjo mokslininkai ir tyrinėtojai. Didelis pasisekimas ir sukaupta patirtis programinės įrangos projektavimo srityje, leido UML kalbai įžengti ir į aparatūrinės įrangos projektavimą. Pateikiama vis daugiau metodikų ir modelių, kaip šią modeliavimo kalbą pritaikyti elektroninės pramonės srityje (McUmber ir kt., 1999; Sinha ir kt., 2000; Damaševičius ir Štuikys, 2004; Xi ir kt., 2004).

Magistro darbo tikslas - ištirti UML modeliavimo kalbos naudojimo galimybes elektroninių sistemų projektavime. UML kalba yra gana patraukli projektuojant aparatūrinę įranga, tačiau ji pagal idealogiją yra bendros paskirties modeliavimo kalba, todėl negalima iškart jos taikyti projektuojant elektronines sistemas. Reikalingi metodai, kurie leistų susieti UML notacijas ir aparatūrinės įrangos aprašymo kalbas. Darbe aptariami kitų autorių pasiūlyti metodai, leidžiantys UML naudoti aparatūros projektavimui, siūlomi nauji jų patobulinimai. Taip pat aptartas *Rational Rose* modeliavimo įrankiui sukurtas kodo generatorius, kuris iš UML diagramų generuoja aparatūros aprašymo programavimo kalbos kodą (VHDL ir SystemC).

1.2. Projektavimo privalumai naudojant UML kalbą

Aparatūrinės ir programinės įrangos projektavimas turi daug panašumų, pvz. abiems atvejais galima išskirti struktūrinius ir elgsenos modelius. Be to ir aparatūrinėje įrangoje vis didėja programinės įrangos kiekis. Naudojant UML elektroninių sistemų projektavime, tikimasi panašių laimėjimų kaip ir programinės inžinerijos srityje.

UML pateikia net 12 diagramų (Booch, 1999). Tai leidžia aprašyti projektuojamą sistemą įvairiais detalumo lygiais ir gali suteikti šiuos privalumus:

- Apibrėžiant aparatūrinę įrangos sistemą abstrakčiu ir nuo realizacijos nepriklausomu būdu gali labai pakelti abstrakcijos lygį.
- Yra galimybė naudoti objektinį projektavimą ir pritaikyti projektavimo šablonus (*design patterns*).
- Naudojant standartines UML diagramas palengvinamas bendravimas tarp skirtingų projektuotojų komandų.
- Projektuojamos aparatūrinės įrangos kokybę užtikrina objektiškai orientuoto projektavimo ir testavimo metodų taikymas.
- Galimas automatinis UML aprašų transformavimas į srities kalbą.

1.3. Magistro darbo tikslai

Kaip jau minėta UML yra bendro pobudžio vizuali modeliavimo kalba skirta specifikuoti, projektuoti ir dokumentuoti sistemos artifaktus. Norint naudoti UML kalbą aparatūros projektavimui, reikia sukurti *metamodelį* (McUmber, 1999), kuris susietų UML elementus su aparatūros aprašymo kalbų konstrukcijomis. Šiame darbe bus tiriama galimybė specifikuoti elektroninės sistemos struktūrą ir elgseną UML kalba. Tam bus pasitelktos UML klasių (*class*) ir būsenų (*state*) diagramos. Užsibrėžtam tikslui pasiekti iškėlėme šiuos uždavinius:

1. Ištirti UML kalbos teikiamas galimybes sistemų specifikavimui ir jų pritaikymą elektroninės įrangos projektavimo srityje. Susipažinti su aparatūros aprašymo kalbomis.
2. Išnagrinėti kitų autorių siūlomus *metamodelius* UML diagramoms susieti su aparatūros aprašymo kalbomis.
3. Pasiūlyti UML klasių ir būsenų diagramų susiejimo su VHDL ir SystemC kalbomis metamodelius.
4. Sukurti kodo generatorių, kuris transformuotų UML klasių ir būsenų diagramų aprašus į VHDL ir SystemC kalbas, naudojant pasiūlytus metamodelius. Patikrinti generuojamo kodo teisingumą.

1.4. Apžvalga

Likusi darbo sandara yra tokia. Antrajame skyriuje aptariami UML kalbos galimybės ir programų generatoriai, trečiajame atliekama aparatūros aprašymo kalbų analizė ir susijusių darbu apžvalga. Ketvirtajame skyriuje aptariamas UML diagramų susiejimas su VHDL ir SystemC kalbomis. Penktajame skyriuje aptariami eksperimentų rezultatai, tiriant sukurtą generatorių. Darbas baigiamas išvadomis ir tolesnių darbų apžvalga šeštajame skyriuje.

2. UML kalba ir programų generatoriai

2.1. UML panaudojimo galimybės

UML (angl. Unified Modeling Language) modeliavimo kalba sparčiai populiarėja visame pasaulyje ir yra naudojama daugelio IT specialistų, kurie projektuoją programinę įrangą. UML yra vizuali kalba, apibrėžianti grafinę notaciją, skirtą įvairių programinės įrangos architektūros aspektų modeliavimui. UML modeliai leidžia greičiau ir lengviau suprasti programinės įrangos struktūrą ir veikimo principus, todėl yra efektyviai naudojami programinės įrangos architektūros dokumentavimui bei projektavimo sprendimų

aptarimui. UML gali pateikti daug projektuojamos sistemos vaizdų, pasitelkdama įvairias struktūrines ir elgsenos diagramas (Wiley, 2003).

UML pateikia dvyliką diagramų tipų , kurios suskirstytos į tris klases :

I. *Struktūrines diagramos* : klasių (*class*), objektų (*object*), komponentų (*component*), išdėstymo (*deployment*).
II. *Elgsenos diagramos* : panaudos atvejų (*use case*), sekų (*sequence*), veiksmų (*activity*), kolaboravimo (*collaboration*), būsenų (*statechart*).
III. *Modelio valdymo diagramos*: Paketų (*packets*) , Posistemių (*subsystems*) ir modelių (*models*).

Projektavimo metu, skirtingos UML diagramos naudojamos skirtingies tikslams. Labai svarbu žinoti, kad UML pateikia tik bendro pobudžio notacijas, kurios turi būti pritaikytos proceso vystymo metu.

Klasių diagrama ko gero labiausiai žinoma UML diagrama. Pasinaudojant klasėmis ir ryšiais tarp jų, nurodami kuriamos sistemos struktūriniai aspektai. Klasės gali turėti atributus ir operacijas. Taip pat sąsajos bei apibendrinimo ryšiai leidžia sukurti objektiškai orientuotas hierarchijas. Paketų diagramos dažniausiai naudojamos paketų aprašymui, leidžia nurodyti paketų struktūrą ir sudėtį. Sistemos elgsenos modeliavimas aukščiausiu abstrakcijos lygiu, prasideda nuo panaudos atvejų ir aktorių, susijusių su sistema, identifikavimo. Tai atliekama panaudos atvejų diagramų pagalba. Tuo tarpu detalesnei elgsenos specifikacijai naudojamos būsenų ir veiksmų diagramos. Sąveikų diagramomis specifikuojamas bendravimas žinutėmis tarp sistemos komponentų. Visos šios diagramos kartu gali pateikti detalią kuriamos sistemos specifikaciją įvairiu abstrakcijos lygiu (Coyle, 2005).

2.2. UML įrankių apžvalga

Šiandien rinkoje galima rasti daug įvairių UML modeliavimo įrankių, pvz.: *IBM Rational*, *UML Studio*, *MagicDraw* ir kitų. Skirtingi įrankiai gali lengvai keistis modeliais. Pasikeitimą užtikrina XMI standartas ir XML paremta aprašymo kalba, kuri

leidžia išsaugoti UML modelių diagramų duomenis portatyviu ir lengvai nuskaitomu formatu.

Trumpai apžvelgsime kelis UML įrankius :

- *Rational Rose* – tai galingas komercinis UML įrankis programinės įrangos projektavimui, sukurtas IBM kompanijos (Rational Software, 2006). Leidžia specifikuoti kuriamą sistemą įvairiais abstrakcijos lygiais, galima naudoti duomenų bazių projektavimui, palaiko automatinį UML aprašų transformavimą į populiauriausias programavimo kalbas (C++, Java, Visual Basic, Ada ir kitas).
- *MagicDraw* – programinės įrangos modeliavimo priemonė, veikianti *Windows*, *Linux*, *Mac* ir kitose labiausiai paplitusiose aplinkose. Šio produkto kūrėjai stengiasi šį įrankį padaryti kuo universalesniu, kad jį naudotų ne tik IT specialistai, bet ir eiliniai vartotojai, įmonių ir projektų vadovai (MagicDraw, 2006).
- *Visual Paradigm* – UML modeliavimo įrankis, sukurtas plačiam vartotojų ratui: programinės įrangos inžinieriams, sistemų analitikams, biznio procesų analitikams, sistemų architektams. Šis įrankis palaiko naujausius UML ir Java kalbų standartus ir integruojamas į populiariausius Java IDE (Visual Paradigm International, 2006).

Magistro darbo tyrimams buvo pasirinktas *Rational Rose* modeliavimo įrankis. Tokį pasirinkimą lėmė tai, kad jis yra plačiai naudojamas, turi daug galimybių ir funkcijų. Rational Rose paketas pateikia scenarijų kalbą *RoseScript*, kurios pagalba galima dar labiau praplėsti šio įrankio galimybes. *RoseScript* kalba bus panaudota VHDL ir SystemC kodo generatoriui kurti.

2.3. Programų generatoriai

Daugelis UML įrankių turi kodo generavimo galimybę. Programos, kurios sukuria programinę sistemą ar didelę jos dalį ir atlieka automatinį specifikacijos transformavimą, yra vadinamos *programų generatoriais* (Thibault *ir kt.,*1997). Technine prasme programų

generatoriai yra į sritį orientuotų kalbų kompiliatoriai. Nors kompiliatoriai taip pat gali būti nagrinėjami kaip generatoriai, generatorių tyrinėtojai paprastai susitelkia prie programų transformavimo srities. Generatoriai gali būti nagrinėjami kaip kompaktiškas didžiulės programų bibliotekos vaizdavimas, jeigu bibliotekoje būtų saugomi visi galimi kodo variantai. Kadangi generatorius sukuria kodą sistemiškai, norimos kodo konfigūracijos gali būti generuojamos automatiškai. Todėl dėl praktinių priežasčių generatoriai dažnai pakeičia dideles panašių programų šeimas. Kita priežastis, dėl kurios naudojami generatoriai, yra ta, jog specifikavimo kalbos, kurias realizuoja generatoriai, leidžia daug glausčiau ir patogiau aprašyti srities problemas negu tikslo kalbos. Transliavimas iš specifikacijų į tikslo kodą yra atliekamas greitai ir korektiškai, taip padidinanamas programuotojo darbo produktyvumas. Be to, generatoriai gali taikyti įvairias specifines optimizacijas ir patikrinti klaidas (Damaševičius, 2001).

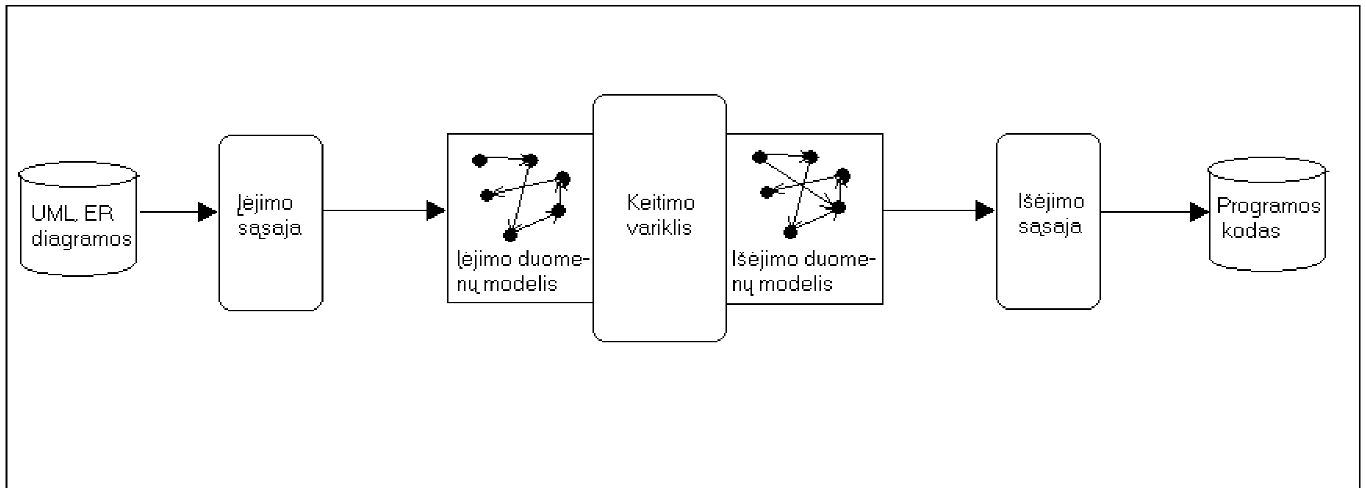
Kadangi generatoriai yra programos, kuriančios kitų programų kodą, mes susiduriame su *metaprogramavimu* (Barlet, 2005). Metaprogramavimas – tai programavimo technologija kai manipuliuojama kitomis programomis, o programa, kuri manipuliuoja kitomis programomis vadinama *metaprograma*. Metaprogramavimo metu galima išskirti du abstrakcijos lygius: žemesnio lygio *tikslo kalbą* ir aukštesnio lygio *metakalbą*. Generavimo metaprogramos – tai programos, kurios atlieka manipuliacijas ir generuoja tikslo programas (Damaševičius, 2001).

2.4. Programų generatorių tipai

Išskiriami trys programų generatorių tipai (Sommerville, 2000) :

1. Taikomųjų programų generatoriai verslo duomenų apdorojimui (pvz . įėjimas srities kalba; išėjimas: SQL arba COBOL).
2. Teksto nagrinėjimo programų ir leksinių analizatorių generatoriai kalbos apdorojimui.
3. Kodo generatoriai CASE priemonėms.

Generatoriai yra labai panašūs į kompiliatorius , tik šiek tiek paprastesni, nes generatoriai retai atlieka vykdomojo kodo optimizaciją (Gabor Karsai, 1998). Pačiu bendriausiu atveju programų generatoriai sudaryti iš trijų dalių: įėjimo sąsajos, keitimo variklio ir išėjimo sąsajos (žr. 1 pav.).



1 pav. Generatoriaus sandara

Generatoriai, skirtingai negu kompiliatoriai, dažniausiai naudoja grafines duomenų struktūras (kompiliatoriai dažniausiai naudoja tekstinius duomenis kaip įėjimą). Todėl generatorių įėjimo sąsaja nuskaito UML, ER diagramas (pvz. UML klasių arba būsenų diagramas). *Keitimo variklis* sukuria išėjimo duomenų struktūrą, pagal kurią išėjimo sąsaja sugeneruoja programos kodą.

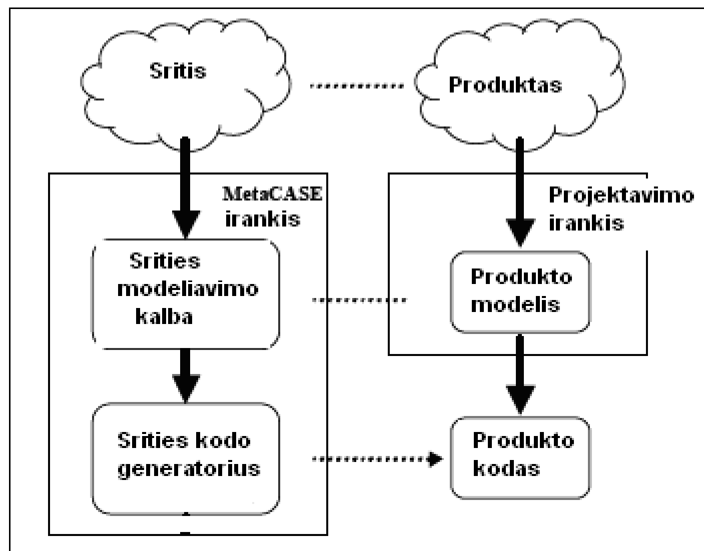
Pagrindinis generatoriaus komponentas yra *keitimo variklis*. Šis komponentas transformuoja diagramas į tikslo kalbos kodą. Galimos mažiausiai dvi strategijos keitimo variklio realizacijai:

1) apibrėžti keitimo veiksmus kaip „įėjimo šablonas -> išėjimo šablonas“ ryšį. Šis būdas lengvas vartotojui, kuris nori specifikuoti keitimo taisykles, tačiau veikimo atžvilgiu tai nėra efektyvu.
2) Kita strategija paremta tuo, kad išanalizavus diagramos tam tikrą mazgą (klasę, būseną) dalis išėjimo duomenų yra suformuojama ir prijungiama prie

kitų išėjimo duomenų dalių. Šis būdas reikalauja žemesnio lygio programavimo nei ankstesnis, tačiau vartotojas turi visą kontrolę ir kodas generuojamas efektyviai.

Generatoriai kuriami tik siauroms, gerai pažintoms sritims. Taikymo sritis turi būti gerai išanalizuota, klasifikuota.

Sistemų šeimynos automatiniam programų variantų generavimui reikalinga turėti: į sritį orientuotą modeliavimo kalbą ir sritį orientuoto kodo generatorių (Pohjonen ir kt., 2004). Bazinė tokios sistemos architektūra pavaizduota 2 pav.



2 pav. Sistemos bazinė architektūra

Šios schemos kairioje pusėje vaizduojami objektai, kuriuos naudoja srities inžinieriai. Šių darbuotojų tikslas sukurti įrankį programų generavimui. Dešinėje pusėje parodytas įrankio naudojimas, generuojant konkrečius programų kodo variantus.

Labai svarbią vietą čia užima srities modeliavimo kalba. Ji turi pateikti srities koncepciją ir semantiką, taip pat vaizduoti produktų šeimynos statinį ir dinaminį variantiškumą.

Taip pat dažnai generatorių programinės įrangos gamintojai pateikia nedidelę, į sritį orientuotą programavimo kalbą *keitimo variklio* programavimui. Tokia programavimo kalba yra specialiai skirta generatorių kūrimo sričiai ir palaiko spartų keitimo komponento prototipo sukūrimą ir modifikavimą.

Rational Rose UML modeliavimo įrankis turi *Rose Scripting* kalbą, kuri skirta generatoriams kurti. Rational Rose scenarijų rašymo kalba sukurta Basic kalbos pagrindu ir yra praplėsta *Summit BasicScriptlanguage* kalbos versija. Praplėtimai leidžia automatizuoti Rational Rose paketo specifines funkcijas, o kai kuriais atvejais leidžia atlikti ir tokius veiksmus, kurie nepasiekiami vartotojui per Rational Rose paketo sąsają. Jos privalumai:

- Greita, efektyvi scenarijų rašymo kalba.
- Sintaksė suderinama su Microsoft VBA kalba.
- Palaikoma įvairiose kompiuterių platformose.
- Gali kontroliuoti Microsoft Ofisse ir kitų programų objektus naudojant OLE.
- Leidžiama praplėsti kalbą, pridedant savo sukurtus objektus, metodus, savybes ir reikšminius žodžius.
- Pateikiamas scenarijų rašymo redaktorius, integruota derinimo programa ir vartotojo sąsajos kurimo įrankis.

2.4. Kodo generatorių privalumai ir trūkumai

Kodo generatorių privalumai:

- *Padeda apibrėžti sistemą panaudojant srities abstrakcijas* . Generatorių įėjimą gali sudaryti tiek sistemos savybių sąrašas, tiek pilna specifikavimo kalba, kuri pateikia neapibrėžtą srities koncepcijų kombinaciją. Specifikacija naudoja srities terminus ir nurodo komponentus, kurie bus naudojami.
- *Išvengiama kodo „korozijos“.* Srities analizė yra brangi, jei vykdoma tik tam tikros problemos srityje analizė, siekiant apibrėžti objektus ir operacijas. Tokia analizė yra efektyvi jeigu naudojama projektuojant iš kart daug sistemų. Pakartotinai naudojama informacija turi būti prižiūrima per visą programos gyvavimo ciklą. Programos kodo aprašai yra pasmerkti „korozijai“, kadangi per ilgą laiką nuolat keičiasi programavimo kalbos, kompiliatoriai, operacinės sistemos. (Neighbors, 1984). Generatorių naudojimas leidžia išvengti programos kodo „korozijos“. Aišku, programų generatoriai taip pat turi būti prižiūrimi ir keičiami, tačiau toks prižiūrėjimas yra centralizuotas ir leidžia keisti iškart daug programos kodo.

- *Leidžia pasiekti realizacijos variantiškumą*. Jeigu komponentai abstraktuoti kodo lygyje, tada bibliotekoje turi būti saugomi tiek komponentų, kiek yra galimų jų variantų (Neighbors, 1994). Tuo tarpu generatoriai nusprendžia generavimo metu, kiek komponentų turi būti sugeneruota.
- *Suteikia kodo optimizavimo galimybes*. Generuojamo kodo optimizavimas gali būti atliekamas sistemos specifikacijoje, kuri yra pateikiama generatoriui. Pavyzdžiui veiksmai, kurie iškviečiami pasibaigus taimerio laikui gali būti pašalinami iš būsenos apibrėžimo jeigu bus įrodyta, kad taimeris niekad nebus įjungtas toje būsenoje. Tokias optimizavimo galimybes būtų labai sunku pastebėti programos kode (Neighbors, 1984).

Kodo generatorių trūkumai :

- Taikytini tik tam tikroje srityje.
- Taikymas apribotas tik išeities kodu.
- Negali būti lengvai pritaikymas visose situacijose.
- Dažnai jie arba per daug bendri arba per daug specifiniai.

3. Analizė

3.1 Srities inžinerija

Šio magistro darbo tyrinėjimo sritis – aparatūrinės įrangos ir įterptinių sistemų projektavimas. Vienas iš darbo tikslų yra sukurti aparatūros aprašymo kalbų kodo generatorių, o tam reikia gerai pažinti nagrinėjamą sritį, arba atlikti *srities inžineriją*. „Kai mes žiūrime į produkto kūrimą, visos produktų šeimynos kontekste, mes turime apibrėžti sritį , nurodydami bendrumus ir skirtumus tarp susijusių produktų. Šis procesas vadinamas srities inžinerija“ – taip apibrėžiama srities inžinerija pagal D. Weis‘ą (Weis, 1999). Srities inžinerija apima vienos arba kelių sričių identifikavimą, skirtingumų radimą srities viduje, lengvai pritaikomos architektūros konstravimą, mechanizmų apibrėžimą, transliuojančių reikalavimus į sistemas iš pakartotinai naudojamų

komponentų. Srities inžinerijos rezultatai yra srities modeliai, architektūros modeliai, į sritį orientuotos kalbos, kodo generatoriai ir programos kodas.

Pradinis žingsnis srities pažinimui yra srities analizė, srities koncepcijos identifikavimas ir supratimas. Srities analizė turi tiksliai apibrėžti nagrinėjamą sritį, analizuoti panašumus ir skirtumus srities viduje, paaiškinti sąryšį tarp skirtingų srities elementų, ir viską atvaizduoti priimtina forma.

3.2 Aparatūros aprašymo kalbos

Aparatūros aprašymo kalbos (AAK) – tai kompiuterinių kalbų klasė, naudojama elektroninių sistemų formaliam aprašymui (Coyle, 2005). Pasinaudojant AAK standartinėmis tekstinėmis išraiškomis, galima aprašyti elektroninės sistemos elgseną ir struktūrą. AAK kalbų yra sukurta labai daug. Šiuo metu pagrindinės AAK kalbos yra VHDL ir Verilog, tačiau nuolat kuriamos naujos AAK kalbos arba tobulinamos senosios, pvz.: Handel-C, JHDL, Objective-VHDL, OpenJ, SystemC (Damaševičius, 2005). Šiame darbe tyrimui buvo pasirinktos VHDL ir SystemC: VHDL kalba standartas aparatūrinės įrangos projektavime, o SystemC yra gan nauja ir vis populiarėjanti AAK kalba, sukurta C++ programavimo kalbos pagrindu.

3.2.1 VHDL apžvalga

VHDL yra skaitmeninės elektroninės aparatūros funkcionavimo aprašymo kalba, kuri 1987 m. buvo pripažinta IEEE 1076 standartu (McUmber, 1999). VHDL yra sukurta ADA programavimo kalbos pagrindu, o jai atsirasti padėjo JAV vyriausybės „Very High Integrated Circuits“ programa.

VHDL leidžia aprašyti bet kokį aparatūriškai realizuojamą algoritmą, todėl VHDL kalba labai panaši į programavimo kalbą. Esminis skirtumas yra tas kad prie konstantų ir kintamųjų dar turi signalus, su kuriais siejami laiko parametrai, ir galima nurodyti operatorių vėlinimus. Be to VHDL leidžia aprašyti lygiagrečiai vykdomus procesus. (Jusas ir kt., 1997).

Nors VHDL yra naudojama aprašyti aparatūrinės įrangos struktūrą ir elgseną, tačiau labai svarbi ir naudinga šios kalbos savybė – galimybė vykdyti programos kodą. VHDL kalba aprašytas skaitmeninis įrenginys gali būti modeliuojamas kompiuteryje, o jo veikimas analizuojamas pagal laikines diagramas. Šia kalba aprašytas įrenginys vėliau gali būti sintezuojamas. Automatinės sintezės priemonės užtikrina, kad sintezuota schema veiktų taip pat, kaip ir VHDL aprašytas įrenginys.

1 lentelė. Pagrindinės VHDL sąvokos

Sąvoka	Paaiškinimas
Entity (objektas)	Objektas yra pirminė aparatūros abstrakcija VHDL kalboje, jis taip pat yra pagrindinis sudarantysis projekto blokas. Objektą apibrėžia objekto aprašas ir objekto architektūra.
Architecture (architektūra)	Visi objektai kuriuos galima modeliuoti turi architektūros aprašymą. Architektūra aprašo objekto elgseną. Ji gali aprašyti objekto elgseną funkciniu arba struktūriniu lygmeniu.
Process (procesas)	Procesas yra bazinis vykdomasis vienetas VHDL kalboje. Jis sudaromas iš procedūrinių sakinių , visi sakiniai procese vykdomi nuosekliai. Keli procesai tarpusavyje vykdomi lygiagrečiai
Signal (signalas)	VHDL kalba papildomai prie kintamųjų ir konstantų turi signalus. Signalas, kaip ir kintamasis gali įgyti reikšmę modeliavimo metu. Kintamasis nuo signalo skiriasi tuo, kad signalui priskirta reikšmė tampa einamąja tik po tam tikro laiko momento tuo tarpu kintamajam priskirta reikšmė tampa einamąja iš karto.
Attribute (atributas)	Atributas gali būti reikšmė, funkcija, tipas, diapazonas, signalas ar konstanta. Yra dvi atributų kategorijos: iš anksto apibrėžti ir vartotojo apibrėžti
Port (išvadai)	Išvadai nurodo objekto sąsają su išoriniu pasauliu. Jie būna dviejų tipų : įėjimo (*IN*) ir išėjimo (*OUT*)
Component (komponentė)	Komponentė tai objektas , kurį naudoja kitas objektas. Objektas gali turėti daug komponenčių

VHDL kalboje modelio aprašymas susideda iš dviejų dalių: sąsajos deklaravimo ir elgsenos aprašymo. Pagrindiniai VHDL programos blokai yra objektas (**entity)** ir architektūra (**architecture)**. Objekto aprašymas prasideda deklaruojant objekto vardą ir išvadus (**ports**). Išvadų aprašymai nurodo signalus ateinančius ir išeinančius į objekto išorę. Objekto elgsenos aprašymas deklaruojamas objekto architektūros bloke. Ten taip pat gali būti nurodomi lokalūs kintamieji, signalai (**signals**) ir procesai (**process**). Visi sakiniai proceso viduje vykdomi nuosekliai, o keli procesai tarpusavyje vykdomi lygiagrečiai. Signalas, kaip ir kintamasis, gali įgyti reikšme modeliavimo metu. Kintamasis nuo signalo skiriasi tuo, kad signalui priskirta reikšmė tampa einamąja po tam tikro laiko momento, kai tuo tarpu kintamajam priskirta reikšmė tampa einamąja iš karto.
Pagrindinės VHDL kalbos savokos ir jų paaiškinimai pateikiami 1 lentelėje. Šios sąvokos bus naudojamos susiejant UML diagramas ir VHDL kalbą.

3.2.2 SystemC apžvalga

SystemC yra aparatūros projektavimo kalba paremta programavimo kalba C++. Jei kalbant tiksliau SystemC nėra kalba, o klasių biblioteka, kuri papildo C++ kalbą aparatūrinės įrangos atributais (Black ir kt., 2004).
Pagrindinis SystemC kalbos blokas yra modulis (**sc_module**). Modulis yra funkcinio vieneto abstraktus atvaizdavimas, nenurodant jokių realizacijos detalių. Kiekvienas modulis turi tam tikrą kiekį išvadų (**ports**), kurių pagalba kontaktuojama su išoriniu pasauliu. Atskiri moduliai komunikuoja vieni su kitais signalų (**signals**) pagalba. Signalai sujungia modulių išvadus. Modulių funkcionalumas aprašomas procesuose.
SystemC palaiko visus C++ kalbos duomenų tipus. Aparatūros projektavimui pateikiami papildomi duomenų tipai: bitai, bitų vektoriai, 4 reikšmių loginis tipas ir kiti. Taip pat pateikiami konstrukcijos perteikiantys aparatūrinės įrangos elgseną. Vykdymui sustabdyti naudojami ***wait()*** sakiniai, nusiųsti ir gauti išvadų reikšmes naudojamos funkcijos ***write()*** ir ***read()***.

SystemC kalbos privalumai pagal R. Damaševičių (Damaševičius, 2005):

1. *Vykdomas kodas* – SystemC kalba aprašytas modelis gali būtis sukompiliuotas ir įvykdytas.
2. *Greitesnė simuliacija* – simuliacijos greitis yra dydesnis lyginant su VHDL ar Verilog kalbomis.
3. *Aukštesnis abstrakcijos lygis* – lyginant su įprastinėmis aparatūros aprašymo kalbomis, SystemC leidžia modeliuoti aukštesniomis abstrakcijos sąvokomis glausta ir elegantiška forma.
4. *Nepriklausomumas nuo realizacijos* – SystemC modelis nenurodo tikslaus realizacijos būdo. Tai gali būti atlikta tiek aparatūriškai, tiek programiškai.

2 lentelėje pateikiamos savokos kurios bus naudojamos tolimesniems tyrimams.

2 lentelė. SystemC sąvokos

Terminas	Paaiškinimas
SC_MODULE (modulis)	SystemC kalboje jis taip pat yra pagrindinis sudarantysis projekto blokas. Jis gali turėti kitus modelius, metodus, įėjimus
SC_METHOD (metodas)	*SC_METHOD* yra klasės *SC_MODULE* funkcija, kuri neturi nei argumentų nei gražinamų reikšmių, gali būti iškviečiama daug kartų
SC_EVENT (įvykis)	Įvykis yra iškviečiamas funkcijos *notify*().
SC_PORT (išvadai)	Išvadai nurodo objekto sąsają su išoriniu pasauliu. Jie taip pat gali būti naudojami susieti atskirus modelius

3.3. Kitų autorių darbų apžvalga

Didėjantis elektroninės įrangos sudėtingumas, verčia projektuotojus ieškoti naujų metodų ir būdų, kaip padidinti projektavimo našumą, sutrumpinti galutinio produkto kelią į rinką ir padidinti jo kokybę, bei patikimumą. Sėkmingas UML įsitvirtinimas programinės įrangos inžinerijoje, priverte projektuotojus pagalvoti apie šios modeliavimo kalbos panaudojimą ir aparatūrinės įrangos projektavime. Šia sritimi besidominantys mokslininkai ir tyrinėtojai (Damaševičius ir kt., 2004; McUmber ir kt., 1999; Xi ir kt., 2005; Sinha ir kt., 2000), savo darbuose aprašo UML taikymo galimybes aparatūrinės įrangos projektavime.

W. McUmber su kolegomis savo straipsnyje (McUmber ir kt., 1999) pristato sistemą, kuri generuoja VHDL programos kodą iš UML diagramų. Autoriai taip pat pateikia taisykles, kaip susieti UML diagramas bei VHDL kalbos konstrukcijas. Šiame projekte buvo tiriamos UML klasių ir būsenų diagramos. Klasių diagramomis vaizduojama projektuojamos aparatūrinės įrangos struktūra ir sudėtinės dalys: kiekviena klasė atitinka VHDL kalbos *entity* ir *architecture* porą, o ryšiai tarp klasių – VHDL signalus. Daug didesnį dėmesį autoriai skiria aparatūros elgsenos modeliavimui. Elgsenai modeliuoti buvo pasirinkta UML būsenų diagrama. Panaudojant UML būsenų diagramų komponentus: perėjimus (*transitions*), įvykius (*events*), veiksmus (*actions*) ir įvairiu tipų būsenas (*states*), modeliuojamas aparatūros elgsenos modelis UML kalba, o sistema sugeneruoja pilną VHDL programos kodą.

Kitokį UML klasių diagramos ir VHDL kalbos susiejimo būdą siūlo R. Damaševičius ir V. Štuikys (Damaševičius ir Štuikys, 2004). Jų pateiktame modelyje įtakos turi tiek klasės tipas, tiek ir ryšiai, jungiantys klases. Abstrakti (abstract) klasė atitinka VHDL *entity* objektą, o klasė sujungta realizuojančiu (*realize*) ryšiu su abstrakčia klase atvaizduojama, kaip VHDL *architecture* blokas. Jeigu klasė jungiama kompoziciniu (*composition*) ryšiu, ji yra komponento (*component*) atitikmuo VHDL kalboje. Toks UML klasių diagramos ir VHDL kalbos susiejimo būdas leidžia naudoti objektiškai orientuotą projektavimo metodologiją ir įvairius projektavimo šablonus (*design*

patterns). Šis privalumas leidžia padidinti projektavimo produktyvumą, sumažinti klaidų tikimybę.

Eksperimentinis UML-SystemC transliatorius (UMLSC) aprašytas Xi straipsnyje (Xi ir kt., 2005). Šis įrankis yra dvikryptis: sugeneruotas SystemC programos kodas gali būti modifikuotas ir šio įrankio pagalba vėl transformuotas į UML aprašus. Projektuojamos sistemos modeliavimui naudojamos UML klasių ir būsenų diagramos. Klasių diagramose gali būti pavaizduotos trys pagrindinės SystemC kalbos esybės : modulis (*module*), kanalas (*channel)* ir sąsaja (*interface*). SystemC modulį atitinka UML klasė, kurios stereotipas „sc_module“, kanalą – klasė, kurios stereotipas yra „sc_channel“, o klasė, kurios stereotipas „sc_interface“ atitinka SystemC kalbos sąsają. Šiomis klasėmis suprojektuojama aparatūrinės įrangos struktūra: sistemos atskiri moduliai sujungiami kanalais. Dinaminės sistemos savybės modeliuojamos UML būsenų diagramų pagalba, tam naudojamos paprastos ir sudėtinės būsenos.

3.4. Išvados

Išnagrinėjus kitų autorių pasiektus rezultatus, tyrinėjant UML panaudojimo galimybes aparatūros projektavime, buvo nutarta:

- Aparatūrinės įrangos struktūros modeliavimui naudoti UML klasių diagramas.
- Susiejant UML klasių diagramas ir aparatūros aprašymo kalbas, naudoti R. Damaševičiaus ir V. Štuikio pasiulytą metodą (Damaševičius ir Štuikys, 2004).
- Aparatūrinės įrangos elgsenos modeliavimui naudoti UML būsenų diagramas.
- Suformuoti metamodelį, kuris leistų susieti UML būsenų diagramas ir aparatūros aprašymo kalbas.

4. UML susiejimas su VHDL ir SystemC

UML, kaip bendro naudojimo modeliavimo kalba, nėra tiesiogiai pritaikyta aparatūrinės įrangos projektavimui. Pirmiausiai UML kalbos aprašai turi būti susieti su aparatūros aprašymo kalbų konstrukcijomis. Jeigu norime automatiškai generuoti kodą, turime nustatyti taisykles, kurios nusako, kaip UML aprašus keisti į aparatūros aprašymo kalbos sakinius.

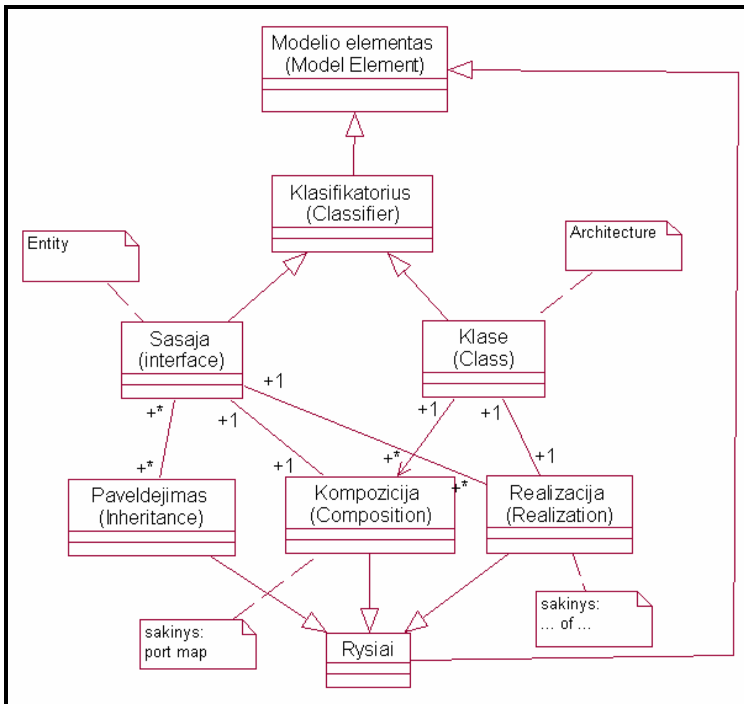
Susiejimas dažniausiai pusiau formaliai apibrėžiamas naudojant UML *metamodelį*. Metamodelis, tai nedidelė aibė UML elementų, kuriais nurodoma UML diagramų sintaksė. Metamodelį sudaro klasių diagrama, kur klasės apibrėžia UML diagramos komponentus (McUmber, 1999). Metamodelio atskiras atvejis yra bet koks UML modelis, kuris sudarytas iš komponentų nurodytų metamodelyje. Metamodelis nusako kokie UML komponentai gali būti naudojami diagramose, kaip jie gali būti jungiami vienas su kitu. Taip pat apibrėžia kaip atskiri komponentai ar jų junginiai atvaizduojami aparatūros aprašymo kalboje.

4.1. UML klasių diagrama ir aparatūros aprašymo kalbos

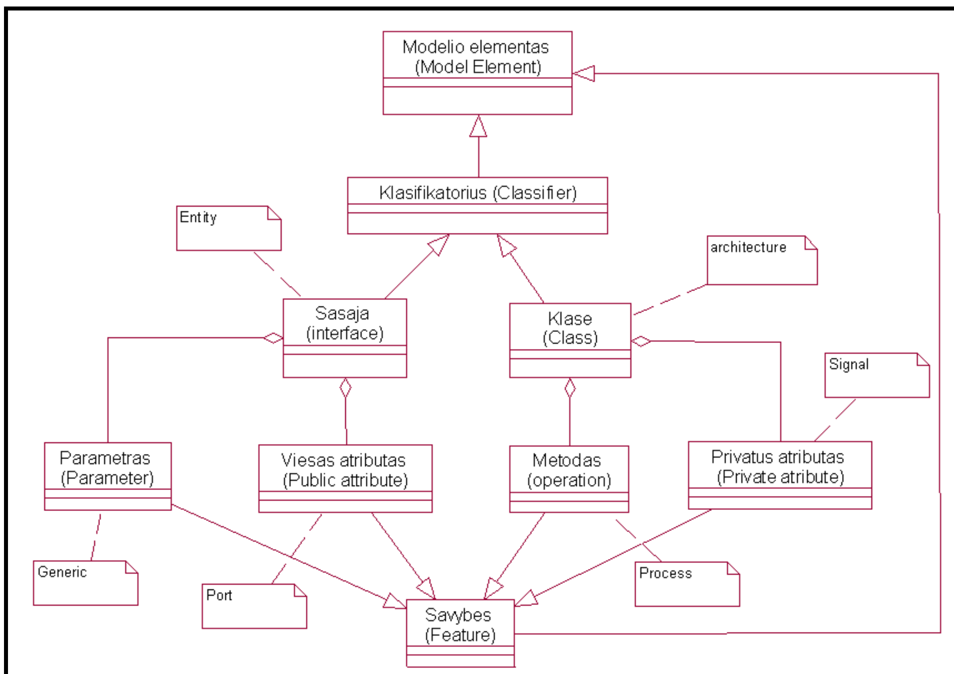
UML klasių diagramos labiausiai tinka projektuojamos aparatūrinės įrangos struktūrai specifikuoti. VHDL ir UML klasių diagramų susiejimui bus panaudotas Damaševičiaus ir Štuikio pasiūlytas metamodelis (Damaševičius ir Štuikys, 2004). Šis metamodelis bus pritaikytas ir SystemC kalbai susieti su klasių diagramomis.

4.1.1 UML klasių diagrama ir VHDL

UML klasių diagramos ir VHDL kalbos struktūrinių abstrakcijų susiejimo metamodelis, susidedantis iš dviejų dalių (ryšių ir bruožų) pavaizduotas 3 ir 4 paveikslėliuose. Metamodelis parodo, kokie klasių diagramos elementai gali būti naudojami aparatūros projektavimui, ir kaip jie transformuojami į VHDL kalbos konstrukcijas.



3 *pav. UML klasių diagramų susiejimo su VHDL kalba metamodelis (ryšiai)*



4 pav. UML klasių diagramų susiejimo su VHDL kalba metamodelis (bruožai)

Šis metamodelis nurodo, kad gali būti naudojamos dviejų tipų klasės : abstrakčios (*interface*) ir paprastos (*class*). Abstrakti klasė, tai projektuojamo aparatūrinio objekto sąsaja su išoriniu pasauliu, todel VHDL kalboje ji atvaizduojama kaip **entity** blokas.

Abstrakti klasė gali būti sujungta su paprasta klase realizuojančiu ryšiu, tokių ryšių ji gali turėti su keliomis klasėmis. Klasė, kuri realizuoja abstrakčią klasę, VHDL kalboje atvaizduojama kaip **architecture** blokas. Abstrakčią ir paprastą klasę dar gali sieti kompozicijos (*composition*) ryšys. Šiuo atveju abstrakti klasė tampa VHDL komponentu (*component*), o susiejimas atvaizduojamas **port map** sakiniu. Paveldėjimo ryšys leidžia klasei paveldėti kitos klasės viešus atributus. UML abstrakčios klasės vieši (*public*) atributai atitinka VHDL kalboje išvadų (*ports*) deklaravimą, o privatūs paprastos klasės atributai – VHDL siganlus (*signals*). Klasės metodai VHDL kalbojei atvaizduojami kaip procesai (*process*), o parametrai leidžia parametrizuoti projektuojamą aparatūros objektą ir VHDL kalboje turi **generic** prasmę.

UML klasių diagramos transformavimui į VHDL kalbą taisyklės:

```
ENTITY <abstrakčios klasės vardas> is
     PORT (
          <viešas atributas  1> <atributo tipas 1>;
          <viešas atributas  2> <atributo tipas 2>
           ……
      );
  END  <abstrakčios klasės vardas>;

 ARCHITECTURE <klasės vardas> OF < abstrakčios klasės vardas> IS
  COMPONENT <klasės sujungtos agregatiniu ryšiu vardas>
   Port(<viešas atributas  1> <atributo tipas 1>;
        <viešas atributas  2> <atributo tipas 2>
  );
  END COMPONENT;
 SIGNAL <privatus  atributas 1> <atributo tipas 1>;
 SIGNAL <privatus atributas  2> <atributo tipas 2>;
 …
 BEGIN
  <operacijos vardas>: PROCESS
  BEGIN

  END PROCESS <operacijos vardas>;

  <agregatinio ryšio vardas>: <klasės sujungtos agregatiniu ryšiu vardas>
    PORT MAP ( <agregatinio ryšio apribojimai> );

 END <klasės vardas>;
```

Testinis pavyzdys pateiktas 3 lentelėje:

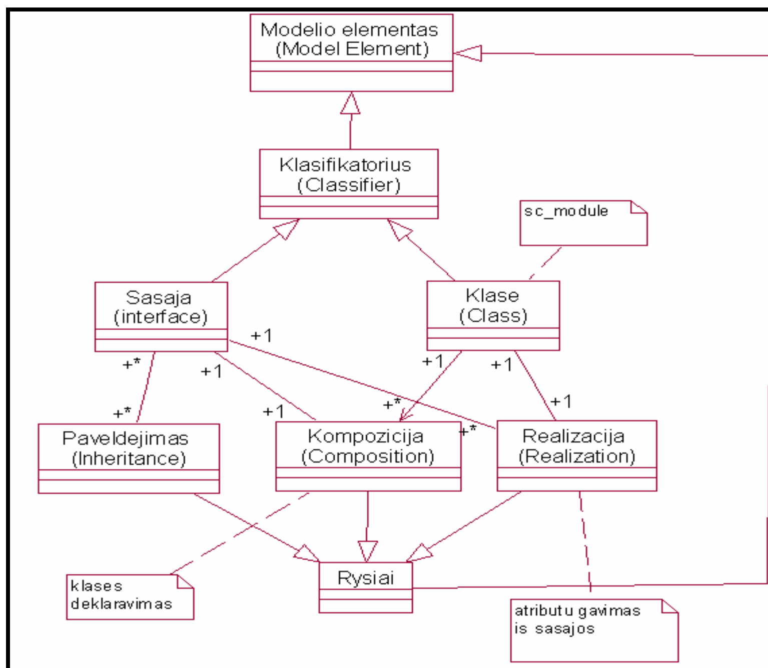
3 lentelė. VHDL generavimas iš UML klasių diagramos

UML klasių diagrama	VHDL programos kodas	

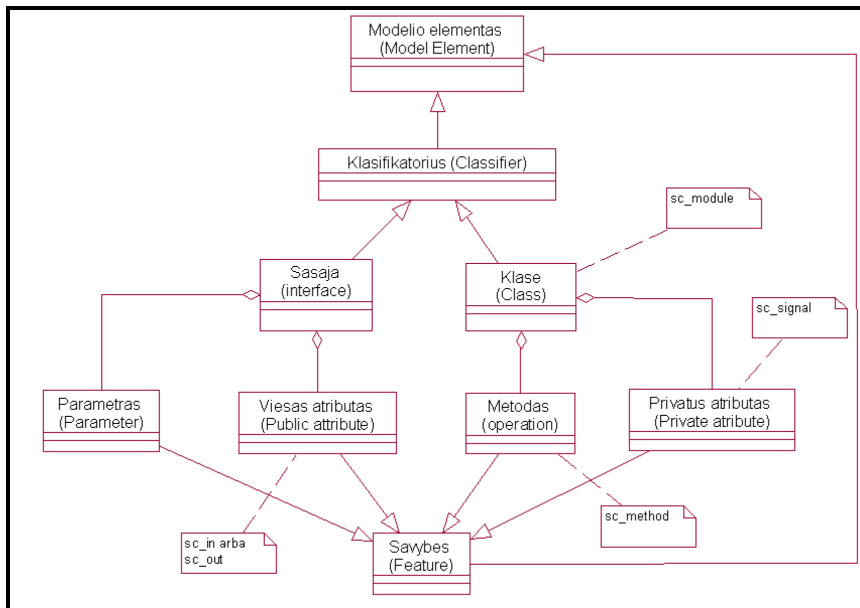
Klase
a : IN BIT;
y : OUT BIT
Realizacija
signalas : BIT
operacija1()
c1
{b => a}
Komp
b : IN BIT
z : OUT BIT
	`library IEEE;`
	`use`
	`IEEE.std_logic_1164.all;`
	`use`
	`IEEE.std_logic_arith.all;`
	`entity Klase is`
	`   -- ports`
	`   port( a: IN BIT;;`
	`         y: OUT BIT);`
	`end Klase;`
	`architecture Realizacija`
	`of Klase is`
	` component Komponentas`
	`   -- ports`
	`   port( b: IN BIT;`
	`         z: OUT BIT);`
	` end component;`
		`SIGNAL signalas: BIT;`
		
		`BEGIN`
		` operacija1: PROCESS(  )`
		` Begin`
		
		` END PROCESS operacija1;`
		
		` c1: Komponentas port map`
		`(`
		`      b=>a`
		` );`
		
		`END Realizacija;`

4.1.2 UML klasių diagrama ir SystemC

SystemC ir VHDL kalbos yra panašios tuom, kad abiejuose kalbose yra atskirta sąsaja ir funkcionalumas. Metamodelis, naudotas UML klasių diagramai ir VHDL kalbai susieti, gali būti pritaikytas ir SystemC kalbai (žr. 5 ir 6 pav.).



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



6 pav. UML klasių diagramos susiejimo su SystemC kalba metamodelis (bruožai)

UML klasių diagramos ir SystemC kalbos metamodelis nusako, kad kaip ir VHDL atveju, projektavimui gali būti panaudotos dviejų tipų klasės. Abstrakti klasė nurodo obejkto sąsają ir SystemC kalboje tiesiogiai nėra atvaizduojama, tačiau jos viešus atributus paveldi paprasta klasė, sujungta realizacijos ryšiu. Tokia klasė SystemC kalboje atvaizduojama moduliu (*sc_module*). Kompozicijos ryšys leidžia deklaruoti kitos klasės naudojimą modulio viduje. Abstrakčios klasės vieši atributai atvaizduojami SystemC kalbos modulio išvadais, paprastos klasės privatūs atributai – SystemC kalbos signalais. Kaip ir VHDL atveju, klasės metodai atvaizduojami SystemC kalbos procesais (*sc_method*).

UML klasių diagramos transformavimui į SystemC kalbą taisyklės:

```
#include <systemc.h>
#include <scALU.h>

SC_MODULE(<klasės vardas>) {
  //Ports
  <atributo tipas 1> <viešas atributas  1>;
  <atributo tipas 2> <viešas atributas  2>;
  ..
  //Signals
  <atributo tipas 1> <privatus  atributas 1>;
  <atributo tipas 2> <privatus  atributas 2>;

  //Submodels
  <klasės sujungtos agregatiniu ryšiu vardas> *<agregatinio ryšio vardas>;
```

```
  SC_CTOR(<klasės vardas>)
    {

      <agregatinio ryšio vardas> = new <klasės sujungtos agregatiniu ryšiu
vardas> ("<agregatinio ryšio vardas>");
      (*<agregatinio ryšio vardas>)(<agregatinio ryšio apribojimai>);

      SC_METHOD(<operacijos vardas>);
      Sensitive <<;

    };
  void <operacijos vardas> ();
  ..
};
```

Testinis pavyzdys pateiktas 4 lentelėje:

4 lentelė. SystemC generavimas iš UML klasių diagramos

<table>
<tr><th>UML klasių diagrama</th><th colspan="2">SystemC programos kodas</th></tr>
<tr>
<td>IfKlase
a : sc_in<bit>
y : sc_out<bit>

Klase
signalas : sc_signal<bit>
operacija1()

c1
{a}

Komp
b : sc_in<bit>
z : sc_out<bit></td>
<td><pre>
//---- file : Klase.h

#include "systemc.h"
#include "Komp.h"
SC_MODULE(Klase)
{

  //Ports
  sc_in<bit> a;
  sc_out<bit> y;

  //Submodels
  Komp c1;

sc_signal<bit>
signalas;
</pre></td>
<td><pre>
//Module Constructor
  SC_CTOR(Klase)

  {
    c1 = new Komp("c1");
    (*c1)(a);

SC_METHOD(operacija1);

    };
    void operacija1();

};//---- end of class
Klase
</pre></td>
</tr>
</table>

4.2 UML būsenų diagrama ir aparatūros aprašymo kalbos

UML būsenų diagramomis aprašant aparatūrinės įrangos elgseną, dažniausiai yra sudaromi baigtiniai automatai. Baigtiniais automatais patogu pavaizduoti kuriamos sistemos galimas būsenas, kaip keičiasi būsenos ir jų priklausomybę nuo įvedamų duomenų.

Skiriamos dvi baigtinių automatų klasės:

1) **Mili** (Mealey) automatai, kurių išėjimo funkcija λ aprašoma taip:

$$\mathbf{z_t} = \lambda\ (\mathbf{x_t}, \mathbf{s_t});$$

2) **Muro** (Moore) automatai, kurių išėjimo funkcija λ aprašoma kitaip:

$$\mathbf{z_t = \lambda (s_t)}.$$

Abiejų klasių automatų perėjimo funkcija δ aprašoma vienodai:

$$\mathbf{s_{t+1} = \delta (x_t, s_t)}.$$

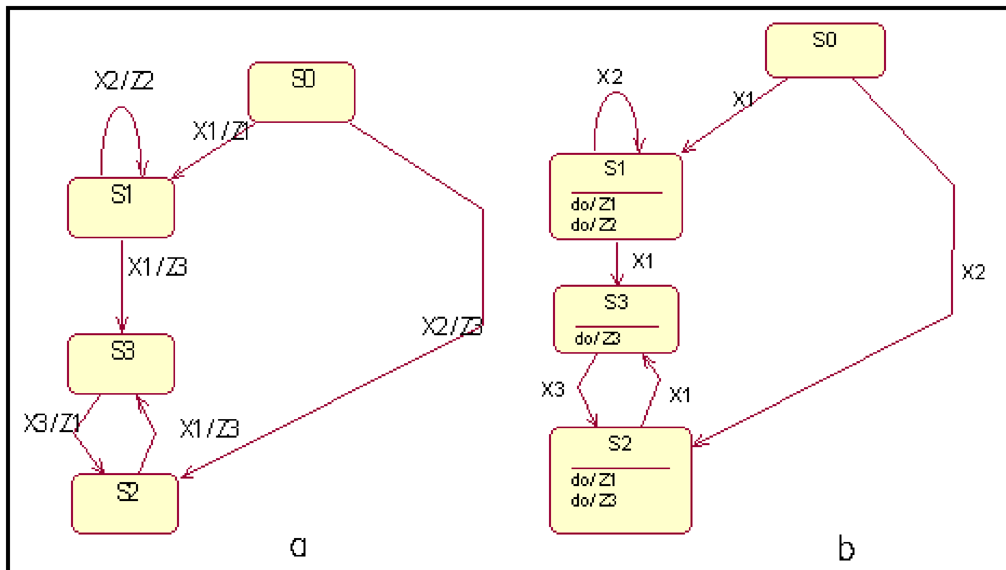
Baigtinius automatus patogu aprašyti panaudojant UML būsenų diagramas :

- Automato būseną atitinka UML būsena
- Automato perėjimai atvaizduojami UML perėjimais tarp būsenų
- Automato įėjimo signalą atitinka UML būsenos perėjimo įvykis (Event)
- Automato išėjimo signalą atitinka UML būsenos perėjimo veiksmas (Action)

Kaip matyti iš aukščiau užrašytų išėjimo funkcijų λ pavidalų, Mili automato išėjimas priklauso ir nuo būsenos, ir nuo tuo metu paduoto įėjimo signalo, tuo tarpu Muro automato išėjimas priklauso tik nuo jo būsenos.

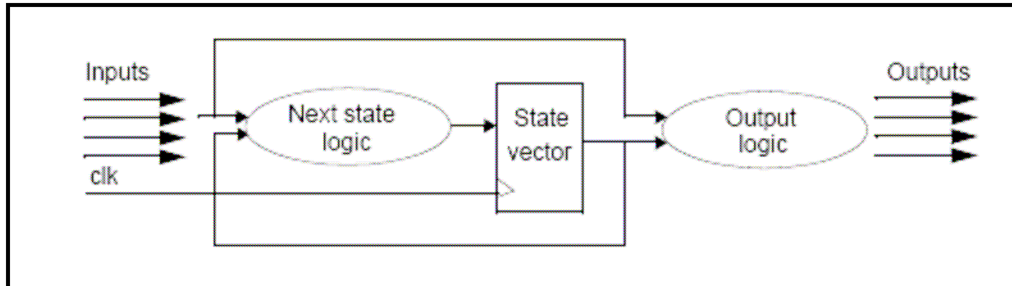
7 paveikslėlyje pavaizduoto automato :

- įėjimo alfabetas $X=\{x_1, x_2, x_3\}$,
- išėjimo alfabetas $Z=\{z_1, z_2, z_3\}$,
- būsenų aibė $S=\{s_0, s_1, s_2, s_3\}$.



7 pav. Mili(a) ir Muro(b) automatai aprašyis UML būsenų diagrama

Automato struktūra gali būti pavaizduota tokia diagrama (žr. 8 pav.)



8 pav. Automato struktūra

Automatas yra sudarytas iš vieno nuoseklaus elemento – būsenų vektoriaus, ir dviejų kombinacinių elementu : kitos būsenos apskaičiavimo ir išėjimų apskaičiavimo elemento. Šioje diagramoje sekančios būsenos apskaičiavimo ir išėjimų apskaičiavimo elementai yra realizuoti atskirai, tačiau jie gali būti sujungti į vieną loginį bloką .

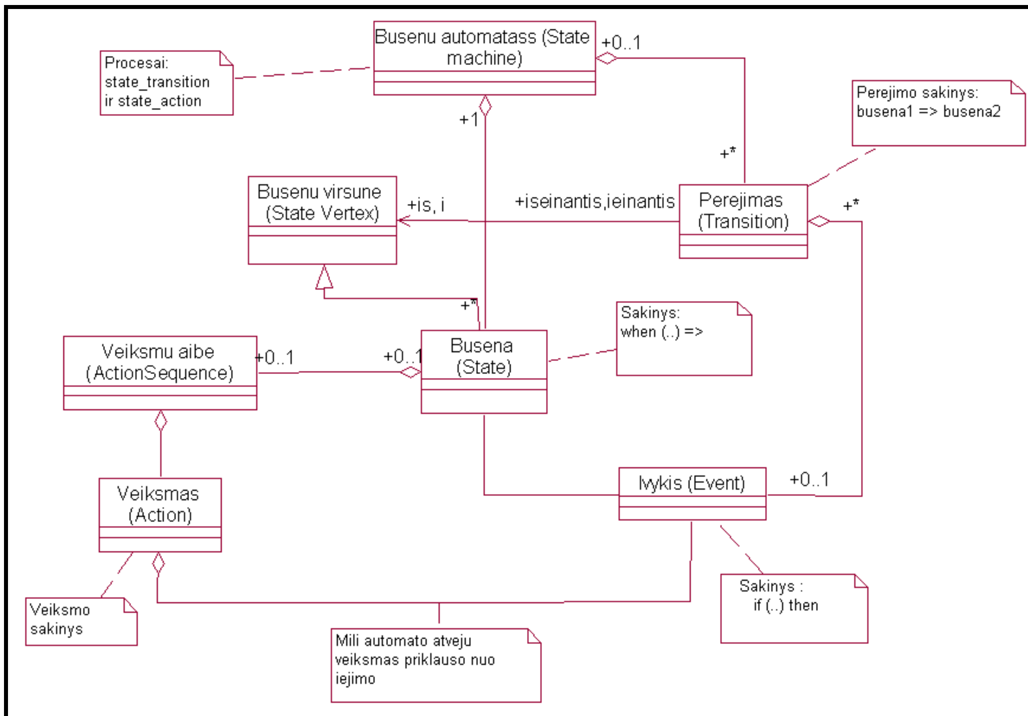
Išėjimų apskaičiavimo įrenginys *(Output logic*) visada yra dabartinės būsenos funkcija, ir pasirinktinai gali būti priklausomas nuo įėjimo signalų (Mili automato atveju). Kitos būsenos apskaičiavimo irenginys (*Next state logic*) taip pat yra visada dabartinės būsenos funkcija ir pasirinktinai gali būti priklausomas nuo iėjimo signalų.

Baigtinių automatų realizavimo būdai aparatūros aprašymo kalbomis:

- Vienas procesas būsenų vektoriaus atnaujinimui ir vienas bendras procesas apskaičiuoti sekančia būseną ir išėjimo signalus.
- Vienas procesas būsenų vektoriaus atnaujinimui, vienas procesas išėjimo signalams apskaičiuoti ir atskiras procesas sekančios būsenos apskaičiavimui.

4.2.1 UML būsenų diagrama ir VHDL

UML būsenų diagrama tinka aparatūrinės įrangos elgsenai specifikuoti. UML būsenos diagramos susiejimui su VHDL kalba buvo sukurtas metamodelis (žr. 9 pav.). Jis nurodo, kaip turi būti suformuojama būsenų diagrama ir kaip šios diagramos elementai yra surišami su VHDL kalbos konstruktais. Metamodelis nurodo, kad ryšiui su VHDL kalba bus naudojami 4 elementai : būsena, perėjimas, įvykis ir veiksmas.



9 pav. UML būsenų diagramos susiejimo su VHDL metamodelis

Jeigu klasė turi būsenų diagramą, jai automatiškai gali būti sugeneruoti du procesai :

1. *state_transition* – procesas, kuris apskaičiuoja automato kitą būseną.
2. *state_action* – procesas, kuris apskaičiuoja išėjimo signalus.

Taip pat sugeneruojamas būsenų tipas T_STATES, o jo galimos reikšmes yra būsenų diagramos būsenų aibė :

```
type T_STATES is ( busena1, busena2, ….., busenaN );
```

Metamodelis nusako, kad projektuojant aparatūrinės įrangos elgseną, naudojami šie UML būsenų diagramos elementai : būsenos (states), perėjimai (transitions), įvykiai (events) ir veiksmai (actions).

UML diagramos būsena nurodo, kokius veiksmus atlieka automatas, taip pat pateikia informaciją apie galimus perėjimus į kitas būsenas. VHDL kalboje būsena atvaizduojama į ***when*** sakinio bloką:

```
WHEN busena =>
  Veiksmai
  ..
```

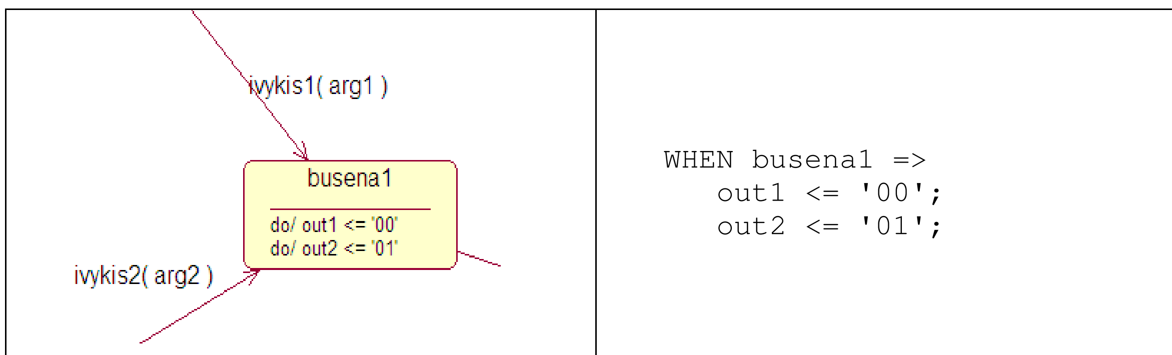
Įvykis inicijuoja būsenų keitimasi, taip pat jis gali nurodyti kokius veiksmus atlikti. Susiejant UML būsenų diagramos įvykį, naudojamas ir jo argumentas. VHDL kalboje įvykis atvaizduojamas į ***if*** sakinį ir yra ***when*** bloko sudėtinė dalis:

```
WHEN busena =>
   if( ivykis = argumentas ) then
      veiksmai;
```

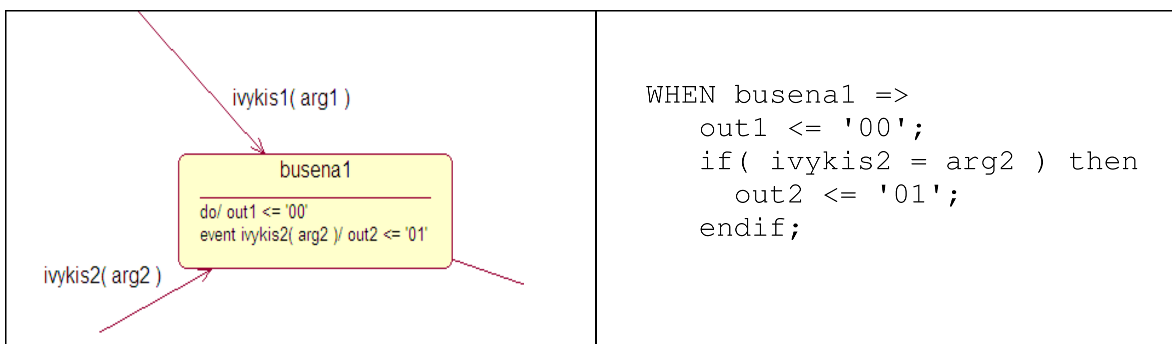
Perėjimas specifikuoja automato perėjimą nuo vienos būsenos prie kitos. Jis atvaizduojamas į VHDL kalbos operatorių **<=** , kuris sujungia esamą būseną su nauja. Suformuota VHDL konstrukcija yra ***if*** bloko sudėtinė dalis:

```
if( ivykis = argumentas ) then
      state <= busena1;
   else
      state <= busena2;
   end if;
```

Veiksmai, nurodo automato atliekamus veiksmus esant tam tikrai būsenai. VHDL kalboje veiksmas atvaizduojamas į VHDL kalbos sakinį kuris taip pat yra ***when*** bloko sudėtinė dalis. Jeigu automatas yra *Mili* tipo (žr. 11 pav.), tai veiksmas atvaizduojamas dar ir ***if*** sakinio viduje. Taip parodoma priklausomybė nuo automato įėjimo.



10 pav. Muro automato būsena ir VHDL kodas



11 pav. Mili automato būsena ir VHDL kodas

Proceso *state_transition* sudarymo taisyklės:

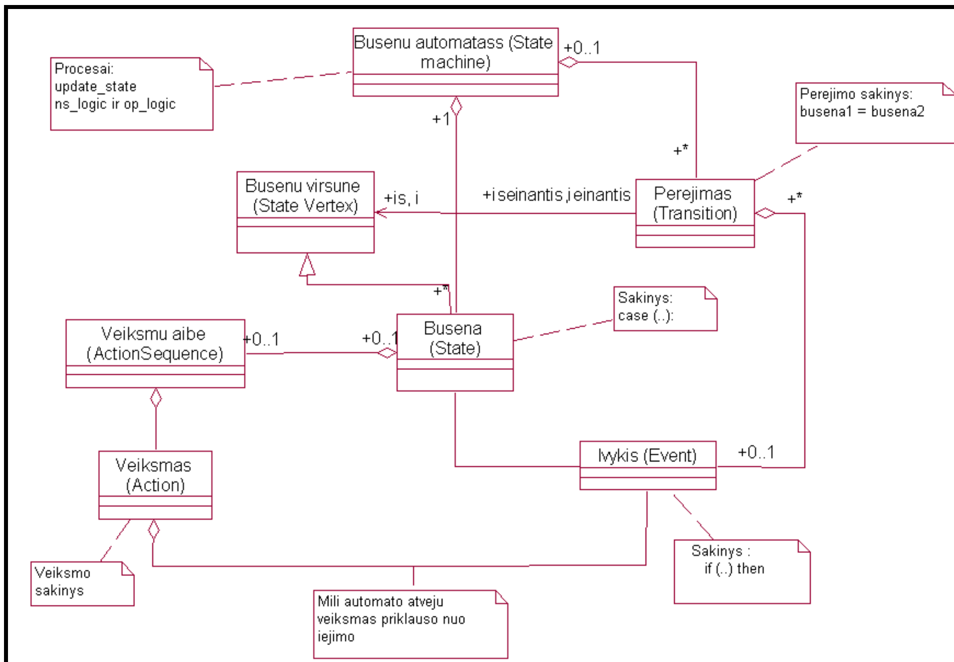
```
state_transition: process ( <įvykis 1>, <įvykis 2>,… , <įvykis N> )
begin
if CLK'event and CLK='1' then
    case state is
      .
      .
      WHEN <būsenos vardas 1> =>
        if( <įvykis 1> = <argumentas 1> ) then
           state <= <nukreipimo būsenos vardas 1>;
        {elsif(<įvykis 2> = <argumentas 2>) then
           state <= <nukreipimo būsenos vardas 2>;}
        else
           state <= <būsenos vardas>;
        end if;
      .
      .
    end case;
end if;
end process;
```

Proceso *state_action* sudarymo taisyklės:

```
state_action: PROCESS ( state )
    BEGIN
      CASE state IS
        .
        .
        WHEN <būsenos vardas 1>  =>
          <veiksmas 1>;
          if(<įvykis 1> = <argumentas 1> ) then
             <veiksmas 2>;
          endif;
        .
        .
       END CASE;
END PROCESS
```

4.2.2 UML būsenų diagrama ir SystemC

UML būsenos diagramos ir SystemC kalbos susiejimo metamodelis yra toks pat kaip VHDL kalbos atžvilgiu (žr. 12 pav.), tik skiriasi tikslo kalba.



12 pav. UML būsenų diagramos susiejimo su SystemC metamodelis

Jeigu neabstrakti klasė turi būsenų diagramą, tai generuojant SystemC kodą, arcitecture bloke automatiškai suformuojami trys *SC_METHOD* tipo procesai :

3. *update_state* – procesas, kuris keičia automato esamą būseną į sekančią būseną.
4. *ns_logic* – procesas, kuris apskaičiuoja kitą būseną.
5. *op_logic* – procesas, kuris apskaičiuoja išėjimo signalus.

Taip pat sugeneruojamas *enum* tipo kintamasis T_STATE, o jo galimos reikšmes yra būsenų diagramos būsenų aibė :

```
enum t_state {
     busena1, busena2, ….., busenaN
};
```

UML diagramos būsena SystemC kalboje atvaizduojama į ***case*** sakinio bloką:

```
case busena =>
  Veiksmai
  ..
```

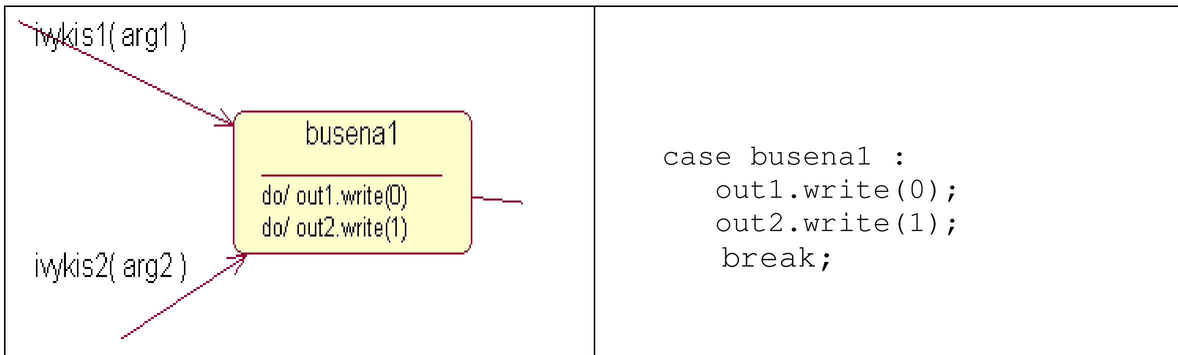
Susiejant UML būsenų diagramos įvykį, naudojamas ir jo argumentas. SystemC kalboje įvykis atvaizduojamas į ***if*** sakinį ir yra ***case*** bloko sudėtinė dalis:

```
case busena :
  if( ivykis.read() == argumentas ) {
     veiksmai;
```

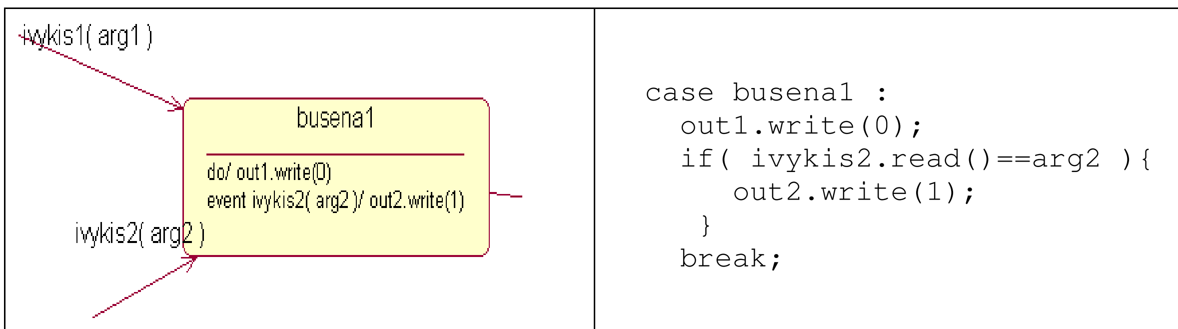
Perėjimas specifikuoja automato perėjimą nuo vienos būsenos prie kitos. Jis atvaizduojamas į SystemC kalbos operatorių **=** , kuris sujungia esamą būseną su nauja. Suformuota SystemC konstrukcija yra ***if*** bloko sudėtinė dalis:

```
if( ivykis.read == argumentas ) {
        state = busena1;
     }else{
        state = busena2;
     };
```

Veiksmai, nurodo automato atliekamus veiksmus esant tam tikrai būsenai. SystemC kalboje veiksmas atvaizduojamas į SystemC kalbos sakinį kuris taip pat yra ***case*** bloko sudėtinė dalis. Jeigu automatas yra *Mili* (žr. 14 pav.) tipo, tai veiksmas atvaizduojamas dar ir ***if*** sakinio viduje. Taip parodoma priklausomybė nuo automato įėjimo.



13 pav. Muro automato būsena ir SystemC kodas



14 pav. Mili automato būsena ir SystemC kodas

Proceso *ns_logic* sudarymo taisyklės:

```
void <klases vardas>::ns_logic() {
  switch(state) {
    ...
    case <būsenos vardas 1>:
         if (<įvykis 1>.read() == <argumentas 1>) {
            next_state = <nukreipimo būsenos vardas 1>;
         }
         break;
  }};
```

Proceso *op_logic* sudarymo taisyklės:

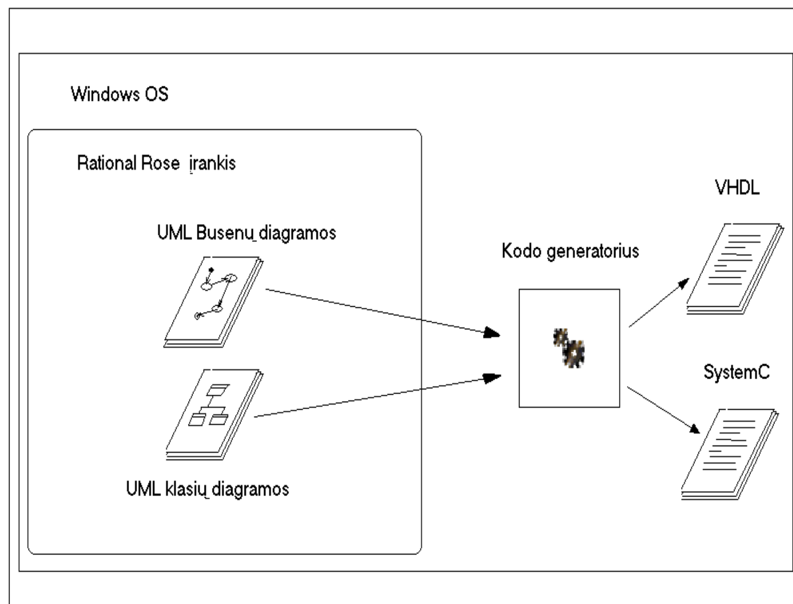
```
void <klases vardas>::op_logic() {
  switch(state) {
    ...
    case <būsenos vardas 1>:
       <veiksmas 1>;
       if (<įvykis 1>.read() == <argumentas 1>) {
               <veiksmas 2>;
             }

       break;
   ...
  }
};
```

5. Kodo generatoriaus testavimas

5.1. Kodo generatoriaus apžvalga

Magistro darbo tyrimams buvo sukurtas VHDL ir SystemC kalbų generatorius Rational Rose įrankiui. Jo struktūra pavaizduota 15 paveikslėlyje, o kodo fragmentas priede A.3.

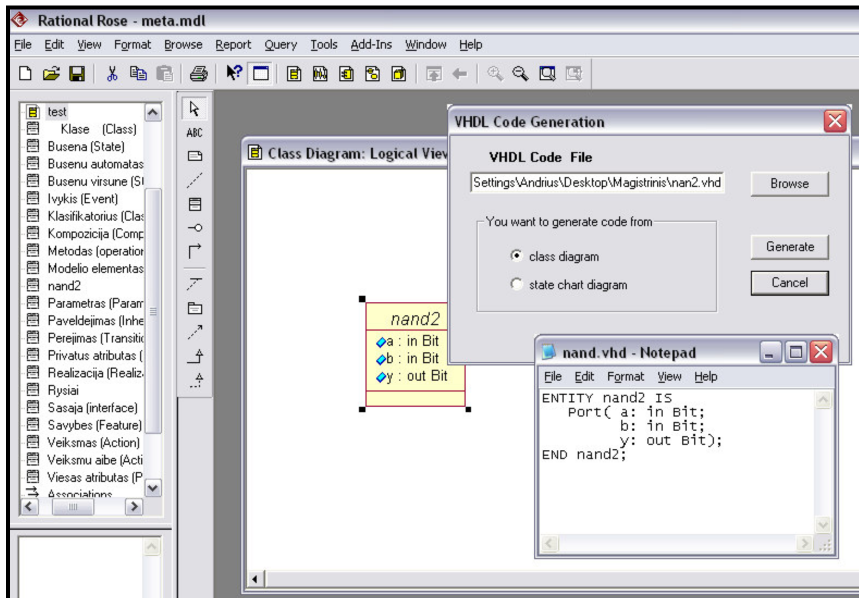


15 pav. Generatoriaus struktūra

Generatorius parašytas RoseScript kalba ir turi tokias galimybes:

1. Generuoti VHDL programos kodą iš UML klasių ir būsenų diagramų.
2. Generuoti SystemC programos kodą iš UML klasių ir būsenų diagramų.

Kadangi generatorius parašytas pasinaudojus Rational Rose paketo suteiktomis priemonėmis funkcionalumui praplėsti, jis lengvai gali būti integruotas į paketą, o juo naudotis nereikalingos papildomos žinios(žr. 16 pav.).



16 pav. Kodo generatoriaus naudojimas

Vartotojas pasirenka klases iš klasių diagramos, pasirenka iš meniu į kokia kalbą bus generuojami UML aprašai, nurodo tekstų išsaugojimo vietą ir generatorius sukuria failus su VHDL arba SystemC kalbos kodu.

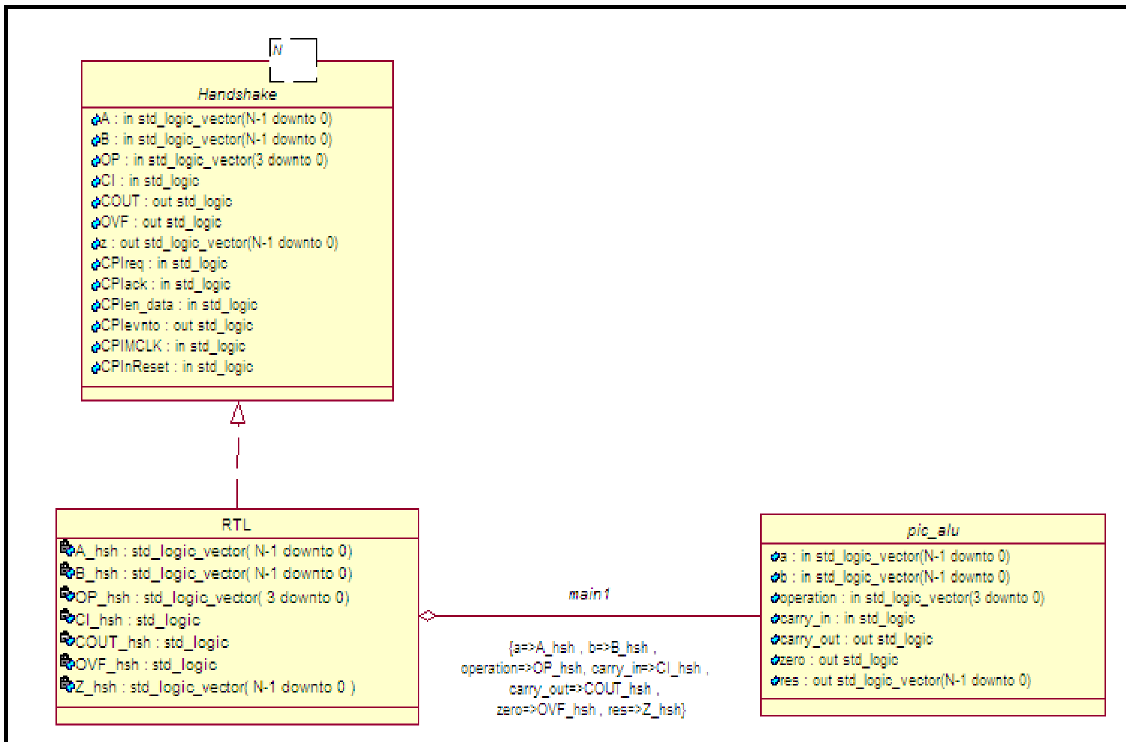
5.2. VHDL kodo generatoriaus tyrimas

Baigtiniai automatai dažnai naudojami įvairių protokolų specifikavimui. Tiriant VHDL kodo generatorių, nuspręsta suprojektuoti aparatūrinį įrenginį, kuris realizuotų *Handshake* (pasisveikinimo) protokolą.

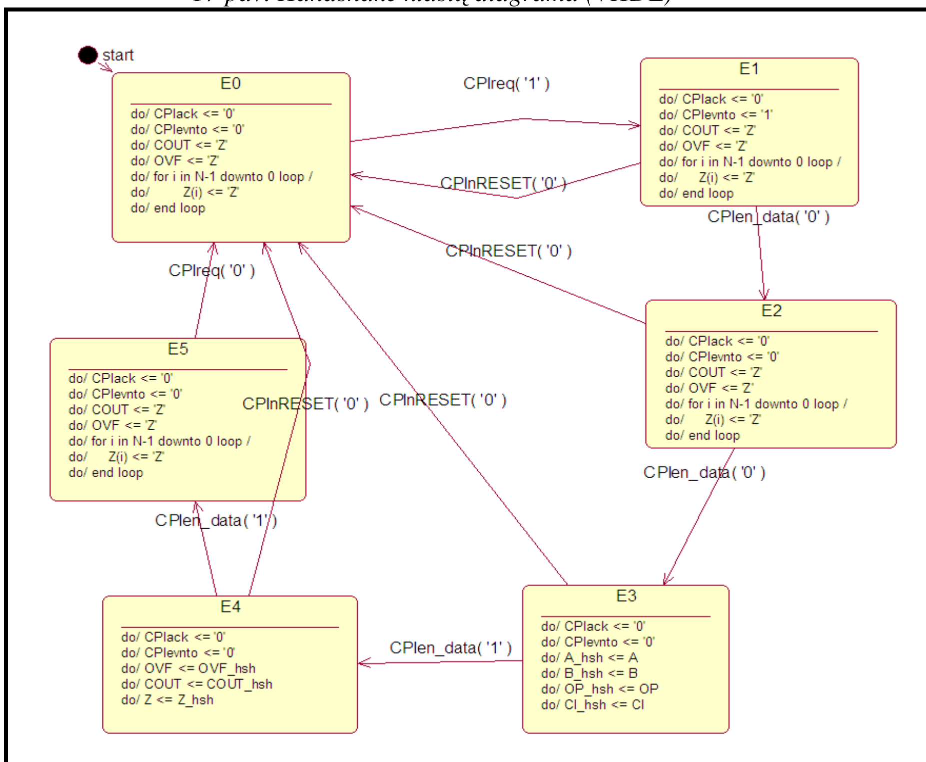
Projektuojamas įrenginys bus sudarytas iš dviejų komponentų :

1. Handshake – kuriame realizuotas „handshake“ protokolas.
2. Aritmetinio loginio įrenginio (ALĮ) duomenų apdorojimui.

Projektuojamas įrenginys bus specifikuotas UML kalba. Struktūriniam viso įrenginio aprašymui sudaryta UML klasių diagrama (žr. 17 pav.). *Handshake* komponento elgsenai aprašyti sudaryta UML būsenų diagrama, kurioje realizuotas „handshake“ protokolas (žr. 18 pav.). ALĮ kodas (žr. priedą A.5.) paimtas iš *PIC16C5X* mikrokontrolerio (The Hamburg VHDL, 2006).



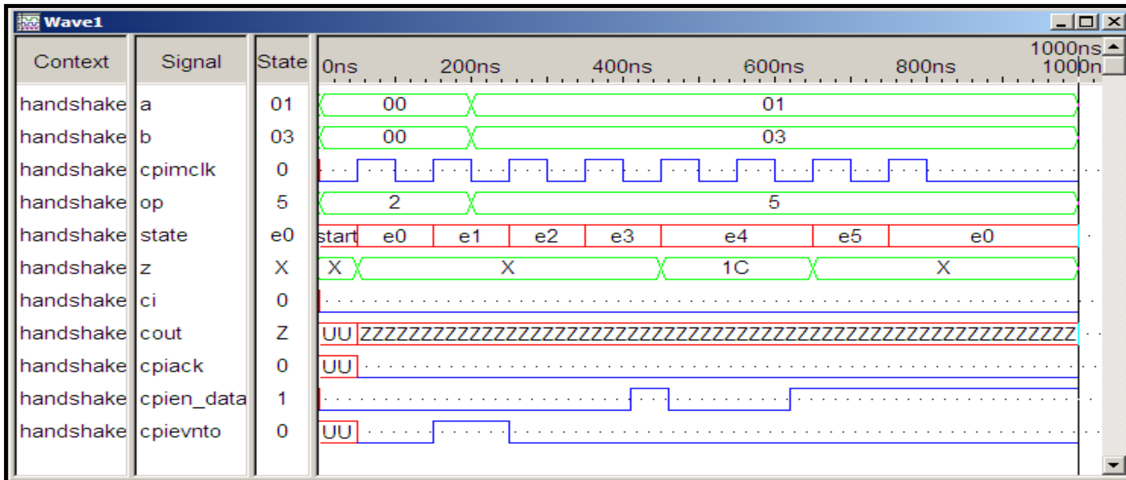
17 pav. Handshake klasių diagrama (VHDL)



18 pav. Handshake būsenų diagrama (VHDL)

Panaudojus VHDL kodo generatorių iš UML klasių ir būsenų diagramų sugeneruojame Handshake programos kodą (žr priedą A.2). Sugeneruotas VHDL programos kodas yra

pilnas ir gali būti modeliuojamas. Modeliavimui panaudota programa *Orcad Express*. Suformavus testinius duomenis, gauname tokią laikinę diagramą (žr. 19 pav.) ir lentelę pateikiančią modelio stimulų reikšmių skaitines reikšmes įvairiais laiko momentais (žr. 20 pav.)



19 pav. Handshake įrenginio laikinė diagrama (VHDL)

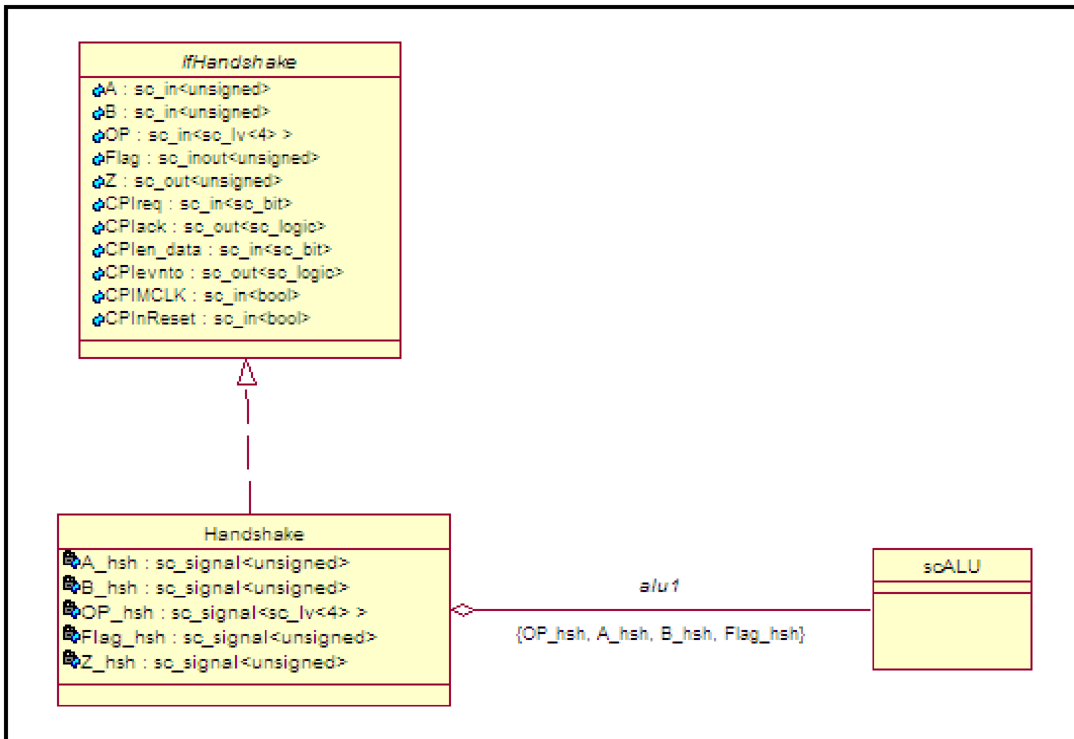
List2

ns	a	b	cout	cpiack	cpien_data	op	state	z
0	00	00	U	U	0	2	start	X
50	00	00	Z	0	0	2	e0	X
150	00	00	Z	0	0	2	e1	X
200	01	03	Z	0	0	5	e1	X
250	01	03	Z	0	0	5	e2	X
350	01	03	Z	0	0	5	e3	X
410	01	03	Z	0	1	5	e3	X
450	01	03	Z	0	1	5	e4	1C
460	01	03	Z	0	0	5	e4	1C
620	01	03	Z	0	1	5	e4	1C
650	01	03	Z	0	1	5	e5	X
750	01	03	Z	0	1	5	e0	X

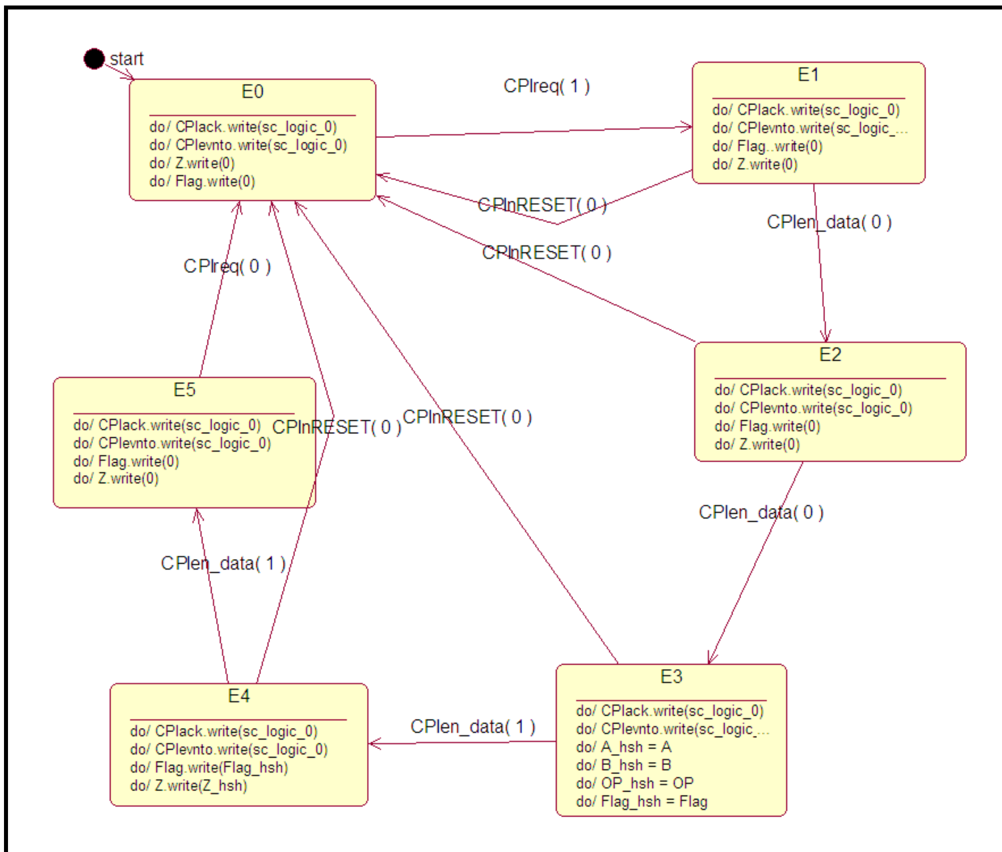
20 pav. Handshake įrenginio skaitinės reikšmės (VHDL)

5.3. SystemC kodo generatoriaus tyrimas

SystemC kodo generatoriaus tyrimui taip pat buvo pabandyta suprojektuoti tokį patį įrenginį kaip ir VHDL atveju. Klasių ir būsenų diagramos pavaizduotos 21 ir 22 paveikslėliuose.



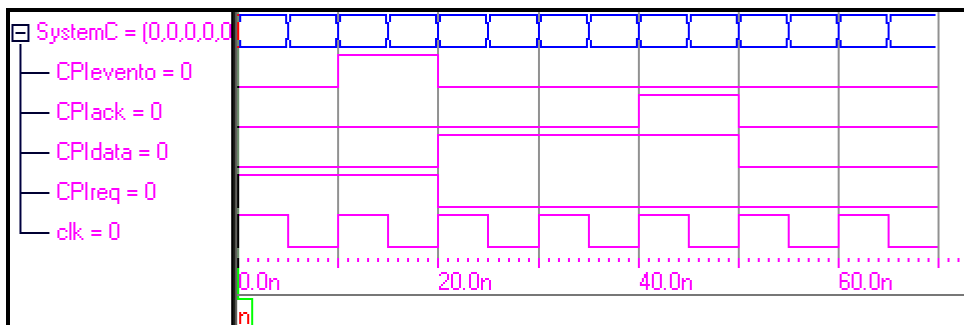
21 pav. Handshake klasių diagrama (SystemC)



22 pav. Handshake būsenų diagrama (SystemC)

Įrenginys taip pat susideda iš dviejų komponentų. ALĮ paimtas iš *Scarm* projekto (Scarm, 2006). ALĮ įrenginio SystemC kodą galima rasti priede (žr. priedą A.6). *Handshake* įrenginys suprojektuotas UML diagramomis. Panaudojus sukurtą SystemC kodo generatorių, generuojamas pilnas SystemC kalbos kodas, kuris gali būti sukompiliuotas. Sugeneruotą programos tekstą galima rasti priede (žr. priedą A.3).

Suprojektuoto įrenginio modeliavimui buvo sudaryti testiniai duomenys. SystemC programa sukompiliuota naudojant *VC++ 6.0* kompiliatorių ir *SystemC 2.1* biblioteką. Modeliavimo rezultai – laikinė diagrama (žr 23 pav.) ir skaitinių reikšmių kitimas (žr. 24 pav.).



23 pav. Handshake įrenginio laikine diagrama (SystemC)

```
                     ALL RIGHTS RESERVED
Note: VCD trace timescale unit is set by user to 1e-9 sec.
state: 0 inA 0 inB: 0 op: XXXX outZ: 0 Flag: 0  at time 0
state: 0 inA 0 inB: 0 op: XXXX outZ: 0 Flag: 0  at time 10
state: 1 inA 2 inB: 3 op: 0100 outZ: 0 Flag: 0  at time 20
state: 3 inA 2 inB: 3 op: 0100 outZ: 0 Flag: 0  at time 30
state: 4 inA 2 inB: 3 op: 0100 outZ: 5 Flag: 0  at time 40
state: 5 inA 2 inB: 3 op: 0100 outZ: 0 Flag: 0  at time 50
state: 0 inA 2 inB: 3 op: 0100 outZ: 0 Flag: 0  at time 60
state: 0 inA 2 inB: 3 op: 0100 outZ: 0 Flag: 0  at time 70
SystemC: simulation stopped by user.
Press any key to continue
```

24 pav. Handshake įrenginio skaitinės reikšmės (SystemC)

5.4. Atliktų rezultatų suvestinė

Aparatūros aprašymo kalbų generatoriaus tyrimui UML kalba buvo suprojektuotas įrenginys, relizuojantis *handshake* protokolą. Projektuojant panaudotos UML klasių ir būsenų diagramos, specifikuojančios įrenginio struktūrą ir elgseną. Suprojektuotų modelių ir sugeneruotų programos teksto charakteristikos pateiktos 5 lentelėje.

5 lentelė. Modelių ir programų tekstų charakteristikos.

Aparatūros aprašymo kalba	UML modeliai		Sugeneruotas kodas	
	Klasių skaičius	Būsenų skaičius	Eilučius skaičius	Dydis (kB)
VHDL	3	7	195	4,58
SystemC	3	7	177	3,95

5.5. Išvados ir įvertinimas

Analizuojant generatoriaus sugeneruotus VHDL ir SystemC programų aprašus ir šių programų modeliavimo rezultatus, galime teigti, kad UML kalba yra tinkama apratūros projektavimui.

Panaudojus UML klasių diagramas galime specifikuoti aparatūrinės įrangos struktūrą: sąsają su išoriniu pasaulių, architektūrą ir sudedamąsias dalis. UML būsenų diagramos tinka elektroninio įrenginio elgsenai specifikuoti. Įrenginiui, kurio elgsenos aprašymui gali būti sudarytas baigtinis būsenų automatas, galima suprojektuoti UML būsenų diagramą, o sukurtas generatorius leidžia sugeneruoti pilną programos kodą. Gali būti naudojamas tiek *Muro*, tiek *Mili* baigtinis automatas. Tyrimo metu sugeneruoti VHDL ir SystemC programų aprašai, be papildomų modifikacijų galejo būti sukompiliuoti ir modeliuojami. Reikėjo tik sudaryti testinius duomenis, reikalingus modeliavimui.

Tačiau aparatūrinės įrangos elgseną nevisada patogu ir įmanoma aprašyti baigtiniu automatu. VHDL ir SystemC kalbos pateikia daug įvairių konstruktų, leidžiančių aprašyti aparatūrinės įrangos funkcionalumą. Sukurtas generatorius apsiriboja ***case***, ***switch***, ***when***, ***if*** sakiniais. Tai neleidžia pilnai išnaudoti visų kalbos galimybių, skirtų aparatūros

elgsenos aprašymui. Literatūroje (McUmber, 1999; Xi, 2005) aprašyti panašūs eksperimentiniai generatoriai, leidžia į generuojamą kodą įtraukti daugiau kalbos konstrukcijų. Tam naudojamos hierarchinės , lygiagrečios būsenos ir kiti UML elementai. Taigi šis generatorius ir pasiūlyti metamodeliai, UML diagramų susiejimui su apratūros aprašymo kalbomis, gali būti tobulinami.

Sukurto generatoriaus privalumas prieš paminėtus kitų autorių kodo generatorius – jog vartotojas nėra pririštas prie konkriačios aparatūros aprašymo kalbos. Galima projektuoti aparatūrinę įrangą neatsižvelgiant į būsimą realizaciją. Vartotojas gali pasirinkti ar generuoti VHDL, ar SystemC programos kodą.

6. Bendrosios išvados ir tolesni darbai

6.1 Išvados

1. Aparatūrinės įrangos projektavimo našumo problemai spręsti galima kelti abstrakcijos lygmenį ir taikyti automatizuoto projektavimo metodus.
2. Aparatūros sistemų projektavimui galima naudoti standartinę aukšto lygmens sistemų specifikavimo kalbą UML:
   - Aparatūrinės įrangos struktūros specifikavimui galima naudoti UML klasių diagramas. Jos leidžia grafiškai specifikuoti aparatūros sistemų struktūrą nepriklausomai nuo realizavimo kalbos ir skatina taikyti atkartojimą srityje.
   - Aparatūrinės įrangos elgsenos specifikavimui galima naudoti UML būsenų diagramas. Jos leidžia specifikuoti aparatūros sistemų elgseną ir papildo UML klasių diagramas.
3. Išnagrinėtas UML klasių diagramų ir VHDL kalbos susiejimo metamodelis, kuris pritaikytas ir SystemC kalbai. Pasiūlyti metamodeliai UML būsenų diagramoms susieti su VHDL ir SystemC kalbomis. Šie metamodeliai aprašo UML diagramų transformavimo į tikslo kalbą principus, palengvina projektavimo srities suvokimą ir srities programų generatorių projektavimą.
4. Taikant heterogeninio metaprogramavimo principus sukurtas kodo generatorius (Rational Rose paketui, panaudojus Rose Script kalbą), leidžiantis iš aparatūros sistemas aprašančių UML klasių ir būsenų diagramų automatiškai generuoti VHDL ir SystemC programas. Generatorius leidžia generuoti sintaksiškai teisingus VHDL ir SystemC aprašus. Sugeneruotas VHDL ir SystemC programos kodas yra pilnas ir gali būti modeliuojamas be modifikacijų.

6.2 Tolesni darbai

Sukurtą kodo generatorių ir pasiūlytus metamodelius, UML klasių ir būsenų diagramų susiejimui su VHDL ir SystemC kalbomis, galima tobulinti. Reiketų įtraukti daugiau UML elementų ir jų savybių projektuojant aparatūrinę įrangą, tokių kaip hierarchinės ir

lygiagrečios būsenos, apsaugos sąlygos (*guard conditions*) perėjimuose tarp būsenų. Tai leistų panaudoti daugiau aparatūros aprašymo kalbų galimybių.
Suprojektuota aparatūrinę įranga reikia testuoti. Testinių duomenų sukurimui galima būtų panaudoti UML sekų (*sequenc*e) diagramas. Tai leistų automatizuoti elektroninių sistemų testavimą.

Literatūra

1. Bartlett J. (2005). ***The art of metaprogramming, Part 1**,* [žiūrėta 2006-03-14], http://www-128.ibm.com/developerworks/linux/library/l-metaprog1.html
2. Black David, Donovan Jack (2004) , ***SystemC from The Ground*** , Kluwer Academic Publishers, Boston, USA, pp. 34 – 50.
3. Booch, G., Rumbaugh, J., Jacobson, I., (1999). ***The Unified Modeling Language User Guide***. Addison-Wesley, Reading, MA.
4. Coyle F. and Thornton M. (2005), ***From UML to HDL: a Model Driven Architectural Approach to Hardware-Software Co-Design***, Computer Science and Engineering Dept Southern Methodist University, Dallas TX 75275.
5. Damasevicius R., Štuikys V., (2004). ***Application of UML for Hardware Design Based on Design Process Model***, *asp-dac*, Asia and South Pacific Design Automation Conference 2004 (ASP-DAC'04), pp. 244-249.
6. Damaševičius R. (2005). ***Subset-based Comparison of Main Design Languages*.** Information Technology & Control, 1(30), pp. 49-56.. *ISSN 1392-124X.*
7. Damaševičius R., (2001). ***Scenarijų kalba Open Promol: išplėtimas aplinka taikymas*:** magistro darbas. KTU Informatikos Fakultetas. [Kaunas]. pp. 4-10.
8. Gabor Karsai (1998) ***Models, Patterns, and Generators for Embeded System*** , Nashville , USA, pp. 2-3.
9. Jusas V. Bareiša. E. , Šeiniauksas R. (1997) , ***Skaitmeninių Sistemų Projektavimas VHDL kalba***, KTU, Kaunas, Lietuva , pp. 8 -15.
10. MagicDraw Corp (2006) [žiūrėta 2006-02-20], http://www.magicdraw.com/.
11. McUmber W.E. and B.H.C. Cheng, (1999). ***UML-based analysis of embedded systems using a mapping to VHDL***, Proc. of IEEE Int. Symposium on High Assurance Software Engineering (HASE'99), Washington, DC, USA, pp. 56-63.
12. Neighbors, James M. , (1994) , ***The Draco Aproach to Constructing Software from Reusable Components***, IEEE transactions on Software Engenering , vol SE-10(5), pp. 564-574.
13. Pohjonen R., Juha-Pekka Tolvanen (2004) ***Product Derivation through Domain-Specific Modeling**,* Jyväskylä, Finland pp. 3-12.
14. Rational Software Corp. (2006) [žiūrėta 2006-02-20], http://www-306.ibm.com/software/rational/
15. Scarm (2006), [žiūrėta 2006-05-01] < http://www.opencores.org/cvsweb.shtml/scarm/>
16. 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, .
17. Sinha V., Doucet D., C. Siska, and R. Gupta, (2000). ***YAML: a tool for hardware design visualization and capture***, Proc. of 13th Int. Symposium on System Synthesis (ISSS'00), 20-22 September, Madrid, Spain, pp. 9-16.
18. Sommerville I. (2000) , ***Software Engineering 6th Edition***, Addison Wesley, Harlow, England.
19. The Hamburg VHDL archive (2006), [žiūrėta 2006-05-05], http://www.pldworld.com/_hdl/1/tech-www.informatik.uni-hamburg.de/vhdl/index.htm.

20. Thibault, S., and C. Consel (1997). ***A Framework for Application Generator Design***. Proceedings of the 1997 Symposium on Software Reusability, May 17-19, 1997, Boston, MA, pp. 131-135. ACM Press.
21. Visual Paradigm International (2006), [žiūrėta 2006-02-20], http://www.visual-paradigm.com.
22. Weiss, D., Lai, (1999) C. T. R., ***Software Product-line Engineering****,* Addison Wesley Longman.
23. Wiley & Sons (2003), ***UML Bible*** , Wiley Publishing, Inc., Indianapolis, Indiana pp. 22-40
24. Xi Chen, Lu JianHua, Zhou ZuCheng, (2005). ***Modeling SystemC design in UML and automatic code generation*****,** Asia and South Pacific Design Automation Conference 2005 (ASP-DAC'05), , Shanghai, China, pp. 932 – 935.

Priedai

A.1. Kai kurių terminų paaiškinimas

Architecture (Architektūra) – aprašo objekto elgseną, funkcionavimą VHDL kalboje.

Entity (Objektas) – pirminė aparatūros abstrakcija VHDL kalboje.

Keitimo variklis – generatoriaus dalis, kuri sukuria išėjimo duomenų struktūrą, pagal kurią išėjimo sąsaja sugeneruoja programos kodą.

Metakalba – kalba, skirta apibendrinti ir komponentų variantams aprašyti bei manipuliuoti programomis, užrašytomis tikslo kalba.

Metamodel (metamodelis) - tai nedidelė aibė UML elementų, kuriais nurodoma UML diagramų sintaksė.

Metaprogramavimas – tai aukštesnio lygmens programavimas, dažniausiai daugiakalbis, kai manipuliuojama programomis kaip duomenimis.

Package (Paketas) – į paketą sujungiami bendrai vartojami duomenų tipai , konstantos paprogramės VHDL kalboje.

Process (Procesas) – yra bazinis vykdomasis vienetas VHDL kalboje.

Rational Rose – UML modeliavimo įrankis

Rational Rose Automation – leidžia integruoti kitas programas į Rational Rose aplinką.

Rational Rose Script – tai Basic kalbos pagrindu, sukurta scenarijų rašymo kalba

REI (Rational Rose Extensibility Interface) – Rational Rose Praplėtimo Sąsaja

Sc_module (modulis) – SystemC kalbos pagrindinis objektas.

SystemC – aparatūros aprašymo kalba sukurta C++ pagrindu.

Tikslo kalba – kalba, skirtasrities funkcionalumui išreiksšti arba algoritmams aprašyti; gali būti arba algoritminė programavimo kalba, arba aparatūros aprašymo kalba.

UML (Unified Modeling language) - standartifikuota ir plačiausiai vartojama notacija objektinio projektavimo grafiniam atvaizdavimui.

VHDL (Very High Speed Integrated circuit Hardware Description Language) yra aparatūros funkcionavimo aprašymo kalba.

A.2. Sugeneruotas *Handshake* programos kodas VHDL kalba

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
entity Handshake is
   generic( N: Integer := 8);

   -- ports
   port( A: in std_logic_vector(7 downto 0);
         B: in std_logic_vector(7 downto 0);
         OP: in std_logic_vector(3 downto 0);
```

```
         CI: in std_logic;
        -- MODE: in std_logic;
         COUT: out std_logic;
         OVF: out std_logic;
         Z: out std_logic_vector(7 downto 0);
         CPIreq: in std_logic;
         CPIack: in std_logic;
         CPIen_data: in std_logic;
         CPIevnto: out std_logic;
         CPIMCLK: in std_logic;
         CPInReset: in std_logic);
end Handshake;


-- code generated for RTL class

architecture RTL of Handshake is


 component pic_alu
  PORT (
         operation : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
         a         : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
         b         : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
         res       : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
         carry_in  : IN  STD_LOGIC;
         carry_out : OUT STD_LOGIC;
         zero      : OUT STD_LOGIC
       );
 END component;

   signal A_hsh: std_logic_vector( 7 downto 0);
   signal B_hsh: std_logic_vector( 7 downto 0);
   signal OP_hsh: std_logic_vector( 3 downto 0);
   signal CI_hsh: std_logic;
   signal COUT_hsh: std_logic;
   signal OVF_hsh: std_logic;
   signal Z_hsh: std_logic_vector( 7 downto 0 );

   type T_STATES is ( E1 ,E2 ,E3 ,E4 ,E5 ,start ,E0 );
   signal STATE : T_STATES := start;
begin

-- ++++ Transition Process +++++

    state_transition: PROCESS ( CPIMCLK, STATE , CPIen_data, CPInRESET,
CPIreq )
  begin
  if CPIMCLK'event and  CPIMCLK='1' then

      case state is

        WHEN start =>
             state <= E0;

        WHEN E1 =>
          if( CPIen_data = '0' ) then
             state <= E2;
```

```
        elsif( CPInRESET = '0' ) then
           state <= E0;
        else
           state <= E1;
        end if;

      WHEN E2 =>
        if( CPIen_data = '0' ) then
           state <= E3;
        elsif( CPInRESET = '0' ) then
           state <= E0;
        else
           state <= E2;
        end if;

      WHEN E3 =>
        if( CPIen_data = '1' ) then
           state <= E4;
        elsif( CPInRESET = '0' ) then
           state <= E0;
        else
           state <= E3;
        end if;

      WHEN E4 =>
        if( CPIen_data = '1' ) then
           state <= E5;
        elsif( CPInRESET = '0' ) then
           state <= E0;
        else
           state <= E4;
        end if;

      WHEN E5 =>
        if( CPIreq = '0' ) then
           state <= E0;
        else
           state <= E5;
        end if;

      WHEN E0 =>
        if( CPIreq = '1' ) then
           state <= E1;
        else
           state <= E0;
        end if;
    end case;
end if;
  end process;
-- ++++ Action Process +++++

  state_action: PROCESS ( state, A, B, CI, COUT_hsh, OVF_hsh, Z_hsh )
  BEGIN
    CASE state IS

      WHEN start =>
           -- nothing to do  --
```

```
WHEN E1 =>
  CPIack <= '0';
  CPIevnto <= '1';
  COUT <= 'Z';
  OVF <= 'Z';
  for i in 7 downto 0 loop
      Z(i) <= 'Z';
  end loop;


WHEN E2 =>
  CPIack <= '0';
  CPIevnto <= '0';
  COUT <= 'Z';
  OVF <= 'Z';
  for i in 7 downto 0 loop
      Z(i) <= 'Z';
  end loop;


WHEN E3 =>
  CPIack <= '0';
  CPIevnto <= '0';
  A_hsh <= A;
  B_hsh <= B;
  OP_hsh <= OP;
  CI_hsh <= CI;
  --Mode_hsh <= MODE;


WHEN E4 =>
  CPIack <= '0';
  CPIevnto <= '0';
  OVF <= OVF_hsh;
  COUT <= COUT_hsh;
  Z <= Z_hsh;


WHEN E5 =>
  CPIack <= '0';
  CPIevnto <= '0';
  COUT <= 'Z';
  OVF <= 'Z';
  for i in 7 downto 0 loop
      Z(i) <= 'Z';
  end loop ;


WHEN E0 =>
  CPIack <= '0';
  CPIevnto <= '0';
  COUT <= 'Z';
  OVF <= 'Z';
  for i in 7 downto 0 loop
         Z(i) <= 'Z';
  end loop;
```

```
      END CASE;
    END PROCESS;



 main1: pic_alu port map (
  a=>A_hsh , b=>B_hsh , operation=>OP_hsh , carry_in=>CI_hsh ,
    carry_out=>COUT_hsh ,
   zero=>OVF_hsh , res=>Z_hsh
 );
end RTL;
```

A.3. Sugeneruotas Handshake programos kodas SystemC kalba (Handshake.h ir Handshake.cpp).

Handshake.h

```
#include <systemc.h>
#include <scALU.h>

SC_MODULE(Handshake) {

  sc_in<unsigned> A;
  sc_in<unsigned> B;
  sc_in<sc_lv<4> > OP;
  sc_inout<unsigned> Flag;
  sc_out<unsigned> Z;
  sc_in<sc_bit>  CPIreq;
  sc_out<sc_logic> CPIack;
  sc_in<sc_bit>  CPIdata;
  sc_out<sc_logic> CPIevnto;
  sc_in<bool> CPIMCLK;
  sc_in<bool> CPInRESET;

  sc_signal<unsigned> A_hsh;
  sc_signal<unsigned> B_hsh;
  sc_signal<sc_lv<4> > OP_hsh;
  sc_signal<unsigned> Flag_hsh;
  sc_signal<unsigned> Z_hsh;

  scALU *alu1;
  // defining the states
  enum state_t {E0, E1, E2, E3, E4, E5};
  sc_signal<state_t> state, next_state;

  SC_CTOR(Handshake)
    {

      alu1 = new scALU ("alu1");
      (*alu1)(OP_hsh, A_hsh, B_hsh, Z_hsh, Flag_hsh);

      SC_METHOD(update_state);
      sensitive_pos(CPIMCLK);
      SC_METHOD(ns_logic);
      sensitive << state << in_valid << CPIreq << CPIdata;
```

```
        SC_METHOD(op_logic);
      sensitive << state << in_valid << CPIreq << CPIdata;

    };
  void update_state();
  void ns_logic();
  void op_logic();
};
```

Handshake.cpp

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

void Handshake::update_state() {

   state = next_state;
}

void Handshake::ns_logic() {

    switch(state) {
    case E0:
        if (CPIreq.read() == 1 ) {
            next_state = E1;
        } else {
            next_state = E0;
        }
      break;

    case E1:
      if (CPIdata.read() == 0 ) {
            next_state = E3;
      } else if  ( CPInRESET.read() == 0 ) {
            next_state = E0;
            }
        else {
            next_state = E2;
        }
      break;

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

```
            next_state = E3;
        }
      break;

    case E4:
      if (CPIdata.read() == 1 ) {
            next_state = E5;
        } else if  ( CPInRESET.read() == 0 ) {
            next_state = E0;
            }
        else {
            next_state = E4;
        }
      break;

     case E5:
      if (CPIreq.read() == 0 ) {
            next_state = E0;
        } else if  ( CPInRESET.read() == 0 ) {
            next_state = E0;
            }
        else {
            next_state = E5;
        }
      break;

    default:
      next_state = E0;
      break;
    }
  }

void Handshake::op_logic() {

    switch(state) {
    case E0:
      CPIack.write(sc_logic_0);
      CPIevento.write(sc_logic_0);
      Flag.write(0);
        Z.write(0);
      break;

    case E1:
      CPIack.write(sc_logic_0);
      CPIevento.write(sc_logic_1);
        Flag.write(0);
        Z.write(0);
      break;

    case E2:
      CPIack.write(sc_logic_0);
      CPIevento.write(sc_logic_0);
      Flag.write(0);
        Z.write(0);
      break;

    case E3:
      CPIack.write(sc_logic_0);
```

```
      CPIevento.write(sc_logic_0);
        A_hsh = A;
        B_hsh = B;
        OP_hsh = OP;
        Flag_hsh = Flag;
      break;

    case E4:
      CPIack.write(sc_logic_0);
      CPIevento.write(sc_logic_0);
        Flag.write(Flag_hsh);
        Z.write(Z_hsh);
      break;

     case E5:
      CPIack.write(sc_logic_1);
      CPIevento.write(sc_logic_0);
      Flag.write(0);
        Z.write(0);
      break;

    default:

      break;
    }
  }
```

A.4. Kodo generatoriaus išeities teksto fragmentas.

```
'--------@@@ Function geneates transition from state "theState"--------
--------------------
Function GeneratesWhenTM ( theState As State ) As String

     Dim theTransitions As TransitionCollection
     Dim theBuffer As String

     Set theTransitions = theState.Transitions

     theBuffer = ebCRLF + ebCRLF + "        WHEN " + theState.Name + "
=>"

    If IsFinalState( theState ) = False Then

      For t = 1 To theTransitions.Count
          Dim theTransition As Transition
          Set theTransition = theTransitions.GetAt(t)

          If Not (theTransition.GetTriggerEvent().name = "") Then
              If ( t = 1 )Then
               theBuffer = theBuffer + ebCRLF + "          if(
"+theTransition.GetTriggerEvent().name    + " = "
              ElseIf ( ( theTransitions.Count > 1 ) And ( t > 1 ) )
Then
                   theBuffer = theBuffer + ebCRLF + "          elsif(
"+theTransition.GetTriggerEvent().name    + " = "
              End If
```

```
                 theBuffer = theBuffer +
theTransition.GetTriggerEvent().Arguments
               theBuffer = theBuffer + " ) then"

          End If


          theBuffer = theBuffer + ebCRLF + "             state <= " +
theTransition.GetTargetState().name + ";"
        'If Not (theTransition.GetTriggerEvent().name = "") Then
        '   theBuffer = theBuffer + ebCRLF + "          End IF;"
        'End If
       Next t
         If IsInitialState( theState ) = False Then
           theBuffer = theBuffer + ebCRLF + "          else" + ebCRLF +
"              state <= " + theState.name + ";" + ebCRLF + "
end if;"
         End If
    Else
       theBuffer = theBuffer + ebCRLF + "             -- nothing to do
---"
    End If

       GeneratesWhenTM = theBuffer

End Function

'---------------------------------------------------------------------
@@@------------
'--------@@@ Function generates transition process block---------------
-------------
Function  GenerateTransitionProcessM ( aClass As Class ) As String

    Dim theBuffer As String
    Dim theStates As StateCollection
    Dim theState As State
    theBuffer = ebCRLF + "-- ++++ Transition Process +++++" + ebCRLF +
ebCRLF
    theBuffer = theBuffer + "    state_transition: PROCESS ( CPIMCLK,
STATE , "
      theBuffer = theBuffer +  GetEvents ( aClass ) + " )"  + ebCRLF +
"  begin" + ebCRLF
      theBuffer = theBuffer +  "  if CPIMCLK'event and  CPIMCLK='1'
then" + ebCRLF
    theBuffer = theBuffer + ebCRLF + "      case state is"

    Set theState = GetInitialState(aClass)
    theBuffer =  theBuffer + GeneratesWhenTM ( theState )
    Set theStates = aClass.StateMachine.States

    For i = 1 To theStates.Count
      Set theState = theStates.GetAt(i)
      If ((IsInitialState( theState ) = False) And (IsFinalState(
theState ) = False)) Then
         theBuffer =  theBuffer + GeneratesWhenTM ( theState )
      End If
    Next i
```

```
    Set theState = GetFinalState ( aClass )
    If HasFinal Then
      theBuffer =  theBuffer + GeneratesWhenTM ( theState )
    End If
      theBuffer = theBuffer + ebCRLF + "      end case;" + ebCRLF+ "
end if;" + ebCRLF + "    end process;"

    GenerateTransitionProcessM = theBuffer

End Function

'-------------------------------------------------------------------
@@@------------


'--------@@@ Function returns action name when system goes to state
"theState"-------------
Function GetActionM (  aClass As Class,  theState As State ) As String

    Dim theBuffer As String
      Dim theStates As StateCollection
      Dim theActionCollection As ActionCollection
      Dim theAction As Action
      Dim aState As State
      Dim found As Boolean
      Dim theEventCollection As EventCollection
      Dim theEvent As Event
      Dim aAction As String

      theBuffer = ""

      Set theActionCollection = theState.GetDoActions ()
      For i = 1 To theActionCollection.Count
       Set theAction = theActionCollection.GetAt(i)
       If ( Mid$(theAction.name,Len(theAction.name),1) = "/" ) Then
           PrintErrMsg  Mid$(theAction.name,Len(theAction.name),1)
             aAction = Mid$(theAction.name,1,Len(theAction.name)-1)
             PrintErrMsg  aAction
       Else
           aAction =    theAction.name + ";"
       End If

       theBuffer = theBuffer + "          "
       theBuffer = theBuffer + theAction.Arguments   + aAction + ebCRLF
    Next i

    Set     theEventCollection = theState.GetUserDefinedEvents ()
      For j = 1 To theEventCollection.Count
       Set theEvent = theEventCollection.GetAt(j)
       Set theAction = theEvent.GetAction ()
       theBuffer = theBuffer + "          "
       theBuffer = theBuffer + "If ( "+ theEvent.name + " = " +
theEvent.Arguments
       'PrintErrMsg  "aa"+ Mid$(theAction.name,1,3)
       MsgBox "Pasirinkta " +
Mid$(theAction.name,Len(theAction.name),1)
       If ( Mid$(theAction.name,Len(theAction.name),1) = "/" ) Then
           PrintErrMsg  Mid$(theAction.name,Len(theAction.name)-1,1)
```

```
             aAction = Mid$(theAction.name,1,Len(theAction.name)-1)
       Else
           aAction =    theAction.name + ";"
       End If
       theBuffer = theBuffer + " ) then" + ebCRLF + "            " +
aAction
       theBuffer = theBuffer  + ebCRLF + "          end if;" + ebCRLF
      Next j

   GetActionM = theBuffer

End Function
'--------------------------------------------------------------------
@@@------------
```

A.5. Aritmetinio loginio įrenginio kodas VHDL kalba (pic_alu.vhd).

```
-- "pic_alu.vhd"
--
-- Copyright (C) 1998  Ernesto Romani  (romani@ascu.unian.it)
--
-- This program is free software; you can redistribute it and/or modify
-- it under the terms of the GNU General Public License as published by
-- the Free Software Foundation; either version 2 of the License, or
-- (at your option) any later version.
--
-- This program is distributed in the hope that it will be useful,
-- but WITHOUT ANY WARRANTY; without even the implied warranty of
-- MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
-- GNU General Public License for more details.
--
-- You should have received a copy of the GNU General Public License
-- along with this program; if not, write to the Free Software
-- Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
--
--
-- Arithmetic-Logic Unit.

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;

ENTITY pic_alu IS
  PORT (
         operation : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
         a         : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
         b         : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
         res       : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
         carry_in  : IN  STD_LOGIC;
         carry_out : OUT STD_LOGIC;
         zero      : OUT STD_LOGIC
       );
END pic_alu;

ARCHITECTURE dataflow OF pic_alu IS

CONSTANT  ALUOP_ADD        : STD_LOGIC_VECTOR (3 DOWNTO 0)  := "0000";
```

```
CONSTANT  ALUOP_SUB        : STD_LOGIC_VECTOR (3 DOWNTO 0)  := "0001";
CONSTANT  ALUOP_AND        : STD_LOGIC_VECTOR (3 DOWNTO 0)  := "0010";
CONSTANT  ALUOP_OR         : STD_LOGIC_VECTOR (3 DOWNTO 0)  := "0011";
CONSTANT  ALUOP_XOR        : STD_LOGIC_VECTOR (3 DOWNTO 0)  := "0100";
CONSTANT  ALUOP_COM        : STD_LOGIC_VECTOR (3 DOWNTO 0)  := "0101";
CONSTANT  ALUOP_ROR        : STD_LOGIC_VECTOR (3 DOWNTO 0)  := "0110";
CONSTANT  ALUOP_ROL        : STD_LOGIC_VECTOR (3 DOWNTO 0)  := "0111";
CONSTANT  ALUOP_SWAP       : STD_LOGIC_VECTOR (3 DOWNTO 0)  := "1000";
CONSTANT  ALUOP_BITCLR     : STD_LOGIC_VECTOR (3 DOWNTO 0)  := "1001";
CONSTANT  ALUOP_BITSET     : STD_LOGIC_VECTOR (3 DOWNTO 0)  := "1010";
CONSTANT  ALUOP_BITTESTCLR : STD_LOGIC_VECTOR (3 DOWNTO 0)  := "1011";
CONSTANT  ALUOP_BITTESTSET : STD_LOGIC_VECTOR (3 DOWNTO 0)  := "1100";
CONSTANT  ALUOP_PASSA      : STD_LOGIC_VECTOR (3 DOWNTO 0)  := "1101";
CONSTANT  ALUOP_PASSB      : STD_LOGIC_VECTOR (3 DOWNTO 0)  := "1110";

SIGNAL temp_b     : STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL temp_carry : STD_LOGIC;

SIGNAL result : STD_LOGIC_VECTOR (8 DOWNTO 0);

SIGNAL bit_pattern : STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL bit_test    : STD_LOGIC_VECTOR (7 DOWNTO 0);

BEGIN

  bit_pattern <= "00000001" WHEN b(7 DOWNTO 5) = "000" ELSE
                 "00000010" WHEN b(7 DOWNTO 5) = "001" ELSE
                 "00000100" WHEN b(7 DOWNTO 5) = "010" ELSE
                 "00001000" WHEN b(7 DOWNTO 5) = "011" ELSE
                 "00010000" WHEN b(7 DOWNTO 5) = "100" ELSE
                 "00100000" WHEN b(7 DOWNTO 5) = "101" ELSE
                 "01000000" WHEN b(7 DOWNTO 5) = "110" ELSE
                 "10000000" WHEN b(7 DOWNTO 5) = "111" ELSE
                 "XXXXXXXX"; -- Only for simulation.

  -- temp_b and temp_carry are used for ADD/SUB operations.
  -- temp_carry is the carry-in of the ADDER.
  -- Note that the ALU's carry-in is only used in shift operations.

  temp_carry <= '1' WHEN operation = ALUOP_SUB ELSE '0';
  temp_b <= NOT b WHEN operation = ALUOP_SUB ELSE b;

  result <= ("0" & a) + temp_b + temp_carry
      WHEN (operation = ALUOP_ADD OR operation = ALUOP_SUB) ELSE
"ZZZZZZZZZ";
  result <= "-" & (a AND b)
      WHEN operation = ALUOP_AND ELSE "ZZZZZZZZZ";
  result <= "-" & (a OR b)
      WHEN operation = ALUOP_OR ELSE "ZZZZZZZZZ";
  result <= "-" & (a XOR b)
      WHEN operation = ALUOP_XOR ELSE "ZZZZZZZZZ";
  result <= "-" & (NOT a)
      WHEN operation = ALUOP_COM ELSE "ZZZZZZZZZ";
  result <= a(0) & carry_in & a(7 DOWNTO 1)
      WHEN operation = ALUOP_ROR ELSE "ZZZZZZZZZ";
  result <= a & carry_in
      WHEN operation = ALUOP_ROL ELSE "ZZZZZZZZZ";
  result <= "-" & a(3 DOWNTO 0) & a(7 DOWNTO 4)
```

```
      WHEN operation = ALUOP_SWAP ELSE "ZZZZZZZZZ";
  result <= "-" & ((NOT bit_pattern) AND a)
      WHEN operation = ALUOP_BITCLR ELSE "ZZZZZZZZZ";
  result <= "-" & (bit_pattern OR a)
      WHEN operation = ALUOP_BITSET ELSE "ZZZZZZZZZ";
  result <= "-" & a
      WHEN operation = ALUOP_PASSA ELSE "ZZZZZZZZZ";
  result <= "-" & b
      WHEN operation = ALUOP_PASSB ELSE "ZZZZZZZZZ";

  res <= result(7 DOWNTO 0);
  carry_out <= result(8);

  bit_test <= bit_pattern AND a;

  zero <= '1' WHEN result(7 DOWNTO 0) = "00000000" ELSE
          '1' WHEN (bit_test /= "00000000") AND
                   (operation = ALUOP_BITTESTSET) ELSE
          '1' WHEN (bit_test = "00000000") AND
                   (operation = ALUOP_BITTESTCLR) ELSE
          '0';

END dataflow;
```

A.6. Aritmetinio loginio įrenginio kodas SystemC kalba (scALU.h ir scALU.cpp).

scALU.h

```
//////////////////////////////////////////////////////////////////////
////////
// This program is free software; you can redistribute it and/or
// modify it under the terms of the GNU General Public License
// as published by the Free Software Foundation; either version 2
// of the License, or (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this program; if not, write to the Free Software
// Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-
1307, USA.
//////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////
//
//  Original Author: Allen Tao Zhong,
//  University of Electronic Science and Technology in China
//  email: zhong@opencores.org
//  info   This is a SystemC ARM model,I "stole" some codes from
//       "swarm" , author Michael Dales (michael@dcs.gla.ac.uk)
//  scALU.h: interface for the scALU class.
//
//////////////////////////////////////////////////////////////////////

#ifndef ALU_H
```

```
#define ALU_H
#include"scTypes.h"
#include<systemc.h>
#include<sc_mslib.h>
enum ALU_AI {AI_NORM = 0, AI_HACK, AI_MAGIC, AI_MULT_LO, AI_MULT_HI};
enum ALU_BI {BI_NORM = 0, BI_HACK, BI_NULL, BI_MULT_LO, BI_MULT_HI};
enum COND {C_EQ = 0x0, C_NE = 0x1, C_CS = 0x2, C_CC = 0x3,
         C_MI = 0x4, C_PL = 0x5, C_VS = 0x6, C_VC = 0x7,
         C_HI = 0x8, C_LS = 0x9, C_GE = 0xA, C_LT = 0xB,
           C_GT = 0xC, C_LE = 0xD, C_AL = 0xE, C_NV = 0xF};
enum OPCODE {OP_AND = 0x00, OP_EOR = 0x01, OP_SUB = 0x02, OP_RSB =
0x03,
           OP_ADD = 0x04, OP_ADC = 0x05, OP_SBC = 0x06, OP_RSC = 0x07,
           OP_TST = 0x08, OP_TEQ = 0x09, OP_CMP = 0x0A, OP_CMN = 0x0B,
           OP_ORR = 0x0C, OP_MOV = 0x0D, OP_BIC = 0x0E, OP_MVN = 0x0F};

class scALU:public sc_module
{
public://ports


      sc_in<OPCODE>    in_OP;
      sc_in<uint32_t>  in_n_A;
      sc_in<uint32_t>  in_n_B;
    sc_out<uint32_t> out_n_Out;
      sc_inout<uint32_t> inout_n_Flag;
public:
      void display();
      SC_HAS_PROCESS(scALU);
      scALU(sc_module_name name) : sc_module(name)
    {
       SC_METHOD(entry);
         sensitive<<in_n_A<<in_n_B<<in_OP;
    }
      virtual ~scALU();
private://implementation
      void entry(void);
 typedef uint32_t alu_fn(uint32_t a, uint32_t b, uint32_t* cont);

         alu_fn and_op;
         alu_fn eor_op;
         alu_fn sub_op;
         alu_fn rsb_op;
         alu_fn add_op;
         alu_fn adc_op;
         alu_fn sbc_op;
         alu_fn rsc_op;
         alu_fn tst_op;
         alu_fn teq_op;
         alu_fn cmp_op;
         alu_fn cmn_op;
         alu_fn orr_op;
         alu_fn mov_op;
         alu_fn bic_op;
         alu_fn mvn_op;

};
#endif
```

scALU.cpp

```
///////////////////////////////////////////////////////////////////////
////////
// This program is free software; you can redistribute it and/or
// modify it under the terms of the GNU General Public License
// as published by the Free Software Foundation; either version 2
// of the License, or (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this program; if not, write to the Free Software
// Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-
1307, USA.
//////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////
//
//  Original Author: Allen Tao Zhong,
//  University of Electronic Science and Technology in China
//  email: zhong@opencores.org
//  info   This is a SystemC ARM model,I "stole" some codes from
//       "swarm" , author Michael Dales (michael@dcs.gla.ac.uk)
//
// scALU.cpp: implementation of the scALU class.
//
//////////////////////////////////////////////////////////////////////

#include "scALU.h"
#define DEBUG
//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////

#include "scALU.h"
#include"scARMInstruction.h"
#define CARRY_FROM(_a,_b,_r) ((_a >> 31) ? ((_b >> 31) | ((~_r) >> 31))
: ((_b >> 31) * ((~_r) >> 31)))

#define BORROWED_FROM(_a,_b,_r) ((_a >> 31) ? ((_b >> 31) & (_r >> 31))
: ((_b >> 31) | (_r >> 31)))

//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////



scALU::~scALU()
{

}
```

```
/***********************************************************************
********
 *
 */
uint32_t scALU::adc_op(uint32_t a, uint32_t b, uint32_t* cond)
{
  int64_t temp;
  //int64_t result;
  uint32_t short_res;
  uint32_t c = (*cond);

  temp = (int32_t)a;
  temp += (int32_t)b;
  if (c & C_FLAG)
    temp++;

  //result = temp & 0x00000000FFFFFFFFL;
  short_res = (uint32_t)((uint64_t)temp);

  // Clear flags
  *cond &= 0x0FFFFFFF;

  // N Flags = Rd[31]
  if (((temp >> 31) & 0x1) == 1)
    (*cond) |= N_FLAG;

  // Z Flag = if Rd == 0 then 1 else 0
  if (short_res == 0)
    (*cond) |= Z_FLAG;

  // C Flag = CarryFrom(Rn + shifter_operand + C Flag)
  if (CARRY_FROM((uint32_t)a, (uint32_t)b, short_res))
    (*cond) |= C_FLAG;

  // V Flag = OverflowFrom(Rn + shifter_operand + C Flag)
  if (((temp >> 32) & 0x1) != ((temp >> 31) & 0x1))
    (*cond) |= V_FLAG;

  return (uint32_t)short_res;
}



uint32_t scALU::add_op(uint32_t a, uint32_t b, uint32_t* cond)
{
  int64_t temp;
  //int64_t result;
  uint32_t short_res;
  uint32_t c = (*cond);

  temp = (int64_t)((int32_t)a);
  temp += (int64_t)((int32_t)b);

  //result = temp & 0x00000000FFFFFFFFL;
  short_res = (uint32_t)((uint64_t)temp);

  // Clear flags
  *cond &= 0x0FFFFFFF;
```

```
  // N Flags = Rd[31]
  if (((temp >> 31) & 0x1) == 1)
    (*cond) |= N_FLAG;

  // Z Flag = if Rd == 0 then 1 else 0
  if (short_res == 0)
    (*cond) |= Z_FLAG;

  // C Flag = CarryFrom(Rn + shifter_operand)
  if (CARRY_FROM((uint32_t)a, (uint32_t)b, short_res))
    (*cond) |= C_FLAG;

  // V Flag = OverflowFrom(Rn + shifter_operand)
  if (((temp >> 32) & 0x1) != ((temp >> 31) & 0x1)) {
    (*cond) |= V_FLAG;
  }

  return (uint32_t)short_res;
}

/*****************************************************************************
********
 *
 */
uint32_t scALU::and_op(uint32_t a, uint32_t b, uint32_t* cond)
{
  uint32_t temp;

  temp = a & b;

  // Clear flags - V flag uneffected
  *cond &= (0x0FFFFFFF | V_FLAG);

  // N Flag = Rd[31]
  if (((temp >> 31) & 0x1) == 1)
    (*cond) |= N_FLAG;

  // Z Flag = if Rd == 0 then 1 else 0
  if (temp == 0)
    (*cond) |= Z_FLAG;

  // C Flag = shifter_carry_out
  // Done otherwhere

  // V Flag = unaffected

  return temp;
}


/*****************************************************************************
********
 *
 */
uint32_t scALU::bic_op(uint32_t a, uint32_t b, uint32_t* cond)
{
  uint32_t temp;
```

```
  temp = a & ~b;

  // Clear flags
  *cond &= (0x0FFFFFFF | V_FLAG);

  // N Flag = Rd[31]
  if (((temp >> 31) & 0x1) == 1)
    (*cond) |= N_FLAG;

  // Z Flag = if Rd == 0 then 1 else 0
  if (temp == 0)
    (*cond) |= Z_FLAG;

  // C Flag = shifter_carry_out
  // Done otherwhere

  // V Flag = unaffected

  return temp;
}


/**********************************************************************
********
 *
 */
uint32_t scALU::cmn_op(uint32_t a, uint32_t b, uint32_t* cond)
{
  int64_t temp;
  //int64_t result;
  uint32_t short_res;
  uint32_t c = (*cond);

  temp = (int64_t)a;
  temp += (int64_t)b;

  //result = temp & 0x00000000FFFFFFFFL;
  short_res = (uint32_t)((uint64_t)temp);

  // Clear flags
  *cond &= 0x0FFFFFFF;

  // N Flags = Rd[31]
  if (((temp >> 31) & 0x1) == 1)
    (*cond) |= N_FLAG;

  // Z Flag = if Rd == 0 then 1 else 0
  if (short_res == 0)
    (*cond) |= Z_FLAG;

  // C Flag = CarryFrom(Rn + shifter_operand)
  if (CARRY_FROM((uint32_t)a, (uint32_t)b, short_res))
    (*cond) |= C_FLAG;

  temp = (int64_t)((int32_t)a);
  temp += (int32_t)b;
```

```
  // V Flag = OverflowFrom(Rn + shifter_operand)
  if (((temp >> 32) & 0x1) != ((temp >> 31) & 0x1))
    (*cond) |= V_FLAG;

  return 0;
}


/*********************************************************************
********
 *
 */
uint32_t scALU::cmp_op(uint32_t a, uint32_t b, uint32_t* cond)
{
  int64_t temp, result;
  uint32_t short_res;
  uint32_t c = (*cond);

  temp = (int32_t)a;
  temp -= (int32_t)b;

  result = temp & 0x00000000FFFFFFFFL;
  short_res = (uint32_t)((uint64_t)temp);

  uint64_t temp2 = (uint32_t)a;
  temp2 -= (uint32_t)b;

  uint64_t result2 = temp2 & 0x00000000FFFFFFFF;

  // Clear flags
  *cond &= 0x0FFFFFFF;

  // N Flag = Rd[31]
  if (((temp >> 31) & 0x1) == 1)
    (*cond) |= N_FLAG;

  // Z Flag = if Rd == 0 then 1 else 0
  if (short_res == 0)
    (*cond) |= Z_FLAG;

  // C Flag = NOT BorrowFrom(Rn - shifter_operand)
  //if (result2 == temp2)
  if (BORROWED_FROM((uint32_t)a, (uint32_t)b, short_res) == 0)
    (*cond) |= C_FLAG;

  // V Flag = OverFlowFrom (Rn - shifter_operand)
  if (((temp >> 32) & 0x1) != ((temp >> 31) & 0x1))
    (*cond) |= V_FLAG;

  return 0;
}


/*********************************************************************
********
 *
 */
uint32_t scALU::eor_op(uint32_t a, uint32_t b, uint32_t* cond)
```

```
{
  uint32_t temp;

  temp = a ^ b;

  // Clear flags (overflow unaffected)
  *cond &= (0x0FFFFFFF | V_FLAG);

  // N Flag = Rd[31]
  if (((temp >> 31) & 0x1) == 1)
    (*cond) |= N_FLAG;

  // Z Flag = if Rd == 0 then 1 else 0
  if (temp == 0)
    (*cond) |= Z_FLAG;

  // C Flag = shifter_carry_out
  // Done otherwhere

  // V Flag = unaffected

  return temp;
}


/***********************************************************************
********
 *
 */
uint32_t scALU::mov_op(uint32_t a, uint32_t b, uint32_t* cond)
{
  // Clear flags (overflow unaffected)
  *cond &= (0x0FFFFFFF | V_FLAG);

  // N Flag = Rd[31]
  if (((b >> 31) & 0x1) == 1)
    (*cond) |= N_FLAG;

  // Z Flag = if Rd == 0 then 1 else 0
  if (b == 0)
    (*cond) |= Z_FLAG;

  //cout<<"move"<<endl;
  // C Flag = shifter_carry_out
  // Done otherwhere

  // V Flag = unaffected

  return b;
}


/***********************************************************************
********
 *
 */
uint32_t scALU::mvn_op(uint32_t a, uint32_t b, uint32_t* cond)
{
```

```
  // Clear flags (overflow unaffected)
  *cond &= (0x0FFFFFFF | V_FLAG);

  // N Flag = Rd[31]
  if (((b >> 31) & 0x1) == 1)
    (*cond) |= N_FLAG;

  // Z Flag = if Rd == 0 then 1 else 0
  if (~b == 0)
    (*cond) |= Z_FLAG;

  // C Flag = shifter_carry_out
  // Done otherwhere

  // V Flag = unaffected

  return ~b;
}


/**********************************************************************
********
 *
 */
uint32_t scALU::orr_op(uint32_t a, uint32_t b, uint32_t* cond)
{
  uint32_t temp;

  temp = a | b;

  // Clear flags (overflow unaffected)
  *cond &= (0x0FFFFFFF | V_FLAG);

  // N Flag = Rd[31]
  if (((b >> 31) & 0x1) == 1)
    (*cond) |= N_FLAG;

  // Z Flag = if Rd == 0 then 1 else 0
  if (temp == 0)
    (*cond) |= Z_FLAG;

  // C Flag = shifter_carry_out
  // Done otherwhere

  // V Flag = unaffected

  return temp;
}
```