KAUNO TECHNOLOGIJOS UNIVERSITETAS

INFORMATIKOS FAKULTETAS

PROGRAMŲ INŽINERIJOS KATEDRA

Linas Ramanauskas

Sisteminio lygmens projektavimo automatizavimas naudojant aktoriais paremtą modeliavimą ir UML

Magistro darbas

Darbo vadovas

dr. R. Damaševičius

Kaunas, 2008

KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
PROGRAMŲ INŽINERIJOS KATEDRA

Linas Ramanauskas

Sisteminio lygmens projektavimo automatizavimas naudojant aktoriais paremtą modeliavimą ir UML

Magistro darbas

Recenzentas	Vadovas
prof. V. Jusas	dr. R. Damaševičius
2008-05-	2008-05-

Atliko
IFM-2/5 gr. stud.
Linas Ramanauskas
2008-05-

Kaunas, 2008

Santrauka

Modeliavimas aukštame abstrakcijos lygmenyje dažnai naudojamas išankstiniam kompromisų analizavimui vienlusčių sistemų projektavimo procese. Šiose magistro tezėse apžvelgiami du daugiausiai žadantys sisteminio lygmens specifikavimo metodai – vieninga modeliavimo kalba (UML) ir į aktorius orientuotas modeliavimas bei galimybės naudoti šiuos metodus kartu. Elgsenos projektavimo pavyzdžių abstrakcijos naudingos supaprastinant į duomenų perdavimą orientuotų sistemų projektavimą. Tradiciškai šie šablonai aprašomi naudojant UML diagramas, tačiau UML trūksta modelio vykdymą aprašančios sintaksės, dėl ko negalima atlikti UML šablonų modeliavimo kartu su šiuo metu vyraujančia vykdomųjų aprašymų technologija sisteminio lygmens projektavimui. Šiame dokumente pateikiamas metodas, kaip integruoti UML elgsenos šablonus kartu su vykdomaisiais sistemos modeliais. Šis metodas remiasi į aktorius orientuotu modeliavimu ir realizuotas kaip Ptolemy II papildymas.

Summary

Modeling at high levels of abstraction is often a need for early trade-off analysis within the Systems-on-Chip design flow. This master thesis overviews two the most promising approaches for system-level specification – Unified Modeling Language (UML) and actor oriented modeling. Also here is presented some possibilities of joint usage of those two approaches. Behavioral patterns are useful abstractions to simplify the design of the communication-centric systems. Such patterns are traditionally described using UML diagrams, but the lack of execution semantics in UML prevents the co-validation of the patterns together with simulation models and executable specifications which are the mainstream in today's system level design flows. In this paper there is described a method to validate UML-based behavioral patterns within executable system models. The method is based on actor orientation and was implemented as an extension of the Ptolemy II framework.

Turinys

Lentelių sąrašas....................6
Paveikslų sąrašas....................7
1 Įvadas....................8
2 Analitinė dalis....................10
2.1 Metamodeliavimas....................10
2.2 Metamodelių specifikavimas....................11
2.3 Objektinio modeliavimo pagrindinės koncepcijos....................12
2.3.1 Objektinio modeliavimo sąvokos....................12
2.3.2 Objektiniai modeliai ir ryšiai....................14
2.3.3 UML diagramos....................16
2.3.4 Aukšto lygmens modelių specifikavimas naudojant UML....................17
2.3.5 Privalumai ir trūkumai objektinio modeliavimo taikymo SoC projektavimui....................18
2.4 Į aktorius orientuoto projektavimo apibrėžimai ir pagrindinės sąvokos....................19
2.5 Į aktorius orientuoto projektavimo procesai, metodai ir įrankiai. Įrankių galimybių palyginimas. Specifikavimo kalbos/notacijos. Palyginimas su UML....................20
2.6 Vienlusčių sistemų į aktorius orientuotas projektavimas. Kitų autorių pasiekti rezultatai....................25
2.7 Vienlusčių sistemų projektavimas naudojant UML....................26
3 Projektinė dalis....................29
3.1 Ptolemy II sistemos architektūra ir galimybės....................29
3.2 Į aktorius orientuoto ir objektinio sistemų projektavimo integravimo modelis....................34
3.3 Kombinuota į aktorius orientuoto/objektinio projektavimo metodika vienlusčių sistemų projektavimui....................41
3.4 Siūloma Ptolemy II sistemos išplėtimo architektūra PtUMLemy....................43
3.5 Siūlomos metodikos ir Ptolemy II sistemos išplėtimo įvertinimas: pranašumai, trūkumai, tolesnio tobulinimo galimybės....................44
4 Eksperimentinė dalis....................44
4.1 Testuojamos sistemos aprašas....................44
4.2 Testų rezultatai....................45
5 Išvados....................47
6 Terminų žodynas....................49
7 Literatūra....................50

Lentelių sąrašas

1 lentelė. Objektinio ir komponentinio (struktūrinio) projektavimo metodų palyginimas.......12
2 lentelė. Simuliavimo rezultatai. Kiek kartų nepavyko pasiekti konsensuso tarp sensorių nuskaitymų..47

Paveikslų sąrašas

1 pav. Paprasčiausias aktoriaus pavyzdys Ptolemy II sistemoje. Parametrai ir vidinė būsena nėra vaizduojama. 21
2 pav. Į aktorius orientuoto modelių iliustracija (viršuje) ir jo hierarchinė abstrakcija (apačioje) 22
3 pav. Hierarchinės abstrakcijos panaudojimas kuriant heterogenines sistemas 23
4 pav. Komponento, generuojančio sinusoide, XML aprašas MoML kalba 23
5 pav. Objektiškai orientuoto projektavimo ir į aktorius orientuoto projektavimo skirtumai pagal Edward A. Lee 24
6 pav. Keletas pagrindinių Ptolemy II klasių, apibrėžiančių Ptolemy II abstrakčią sintaksę ir semantiką. Jos aprašytos kernel, kernel.util ir actor paketuose. 31
7 pav. Elgsenos projektavimo pavyzdys UML kalba 35
8 pav. Sekų diagramos integravimas į aktoriais paremtą modelį 37
9 pav. Į aktorius orientuotas modelis su integruota sekų diagrama ir išoriniu interfeisų apibrėžimu. 38
10 pav. Elgsenos projektavimo pavyzdžių integravimo metamodelis 39
11 pav. Į aktorius orientuotų diagramų metamodelis 40
12 pav. Sekų diagramų metamodelis 40
13 pav. Klasių diagramų metamodelis 41
14 pav. Ptolemy II papildymo klasių diagramų redaktorius 42
15 pav. Ptolemy II papildymo sekų diagramų redaktorius 42
16 pav. 9 paveikslėlyje vaizduojamo modelio realizacija Ptolemy II aplinkoje. 43
17 pav. Sekų diagramos testavimo modelis Ptolemy II sistemoje 46

1 Įvadas

Didžioji dalis mūsų gebėjimo projektuoti ir optimizuoti sistemas priklauso nuo mūsų sugebėjimo atpažinti problemų panašumą su kitomis problemomis, kurias mes jau žinome kaip išspręsti. Būtent šia prielaida remdamasi apie 1990 metus programinės įrangos projektavimo inžinierių bendruomenė pradėjo analizuoti ir grupuoti gerai žinomus dažnai objektiniam programinės įrangos projektavime pasitaikančių problemų būdus, pavadindami juos projektavimo pavyzdžiais. Nors ši idėja nebuvo originali, pavyzdžių dokumentavimas ir pakartotinio jų panaudojimo palengvinimas buvo lengvai priimtas ir peržengė objektiškai orientuoto programinės įrangos projektavimo pritaikymo SoC [23], realaus laiko sistemų [24], paskirstytųjų skaičiavimų [25] ir bevielių sensorių tinklų [26] projektavimo ribas.

Pagal Gamma [22], projektavimo pavyzdžiai pagal jų tikslą gali būti klasifikuojami į: kūrimo, struktūros ir elgsenos. Kūrimo pavyzdžiuose analizuojami objektų kūrimo procesai, tuo tarpu struktūros pavyzdžiuose apibūdinama klasių ar objektų kompozicija, o elgsenos pavyzdžiai aprašo klasių ar objektų sąveikos būdus. Tradiciškai šie šablonai aptašomi naudojant UML diagramas. [22] knygoje visi projektavimo pavyzdžiai, arba dar kitaip vadinami projektavimo šablonai, iliustruoti naudojant UML klasių diagramas, o daugumai elgsenos šablonų papildomai vaizduojamos sekų diagramos, atspindinčios tarp šablono objektų siunčiamus pranešimus.

Šiame darbe analizuojamas elgsenos projektavimo pavyzdžių panaudojimas kaip būdas skatinti pakartotinį panaudojimą komunikavimo schemų pavyzdžių specifikuojant sisteminio lygmens modelius. Tam, kad laikytis elgsenos projektavimo pavyzdžių vaizdavime nusistovėjusių taisyklių ir tuo pat metu gauti naudos naudojant senesnius šablonus, naudojamos standartinės UML diagramos. Siekiant padidinti potencialų šių šablonų panaudojimą siūloma integruoti juos į vykdomuosius specifikavimo modelius (tokius kaip Matlab/Simulink, SystemC, VHDL ir Verilog) kurie šią dieną yra plačiausiai naudojami projektavimo eigoje. Propaguodamas tokią integraciją šis požiūris išsiskiria iš kitų darbų, siūlančių vienlusčių sistemų specifikavimą UML kalba. Siūlomas integravimo modelis yra pagrįstas į aktorius orientuoto modeliavimo paradigma.

Dauguma dabartinių UML panaudojimo kaip aparatūros/programinės įrangos sisteminio lygmens aprašymo kalbos metodų aptaria statinę analizės arba kodo generavimo metodus. Oliveira knygoje [27] naudoja statinės analizės metodus įvertinant našumą, atminties ir energijos suvartojimą alternatyvių įterptinės programinės įrangos elgsenos šablonų naudojant UML sekų diagramas. [28] UML klasių diagramos naudojamos kaip

šablonai projektuojant ir analizuojant užimamą sistemos plotą. Taip pat galima rasti nemažai skirtingų kodo generavimo iš UML diagramų metodų tyrimų. Dauguma jų propaguoją bendrą UML ir SystemC naudojimą. Tyrėjų komanda iš Catania universiteto ir ST Microelectrinics kompanijos teigia, kad į UML reiktų žvelgi kaip į aukšto lygmens modeliavimo kalbą, tuo tarpu SystemC turėtų būti laikoma žemo lygmens sistemine kalba. Tam kad realizuoti sąryšį tarp šių kalbų, jie pasiūlė stereotipų rinkinį, kuris leidžia modeliuoti SystemC koncepcijas UML diagramomis. Stereotipai sugrupuoti ir pavadinti UML 2.0 SystemC palaikymo papildymu. Panašūs požiūriai, tik su skirtinais stereotipų apibrėžimais, buvo pasiūlyti tyrėjų iš Politecnico de Milano, Siemens ICM ir Fujitsu. Visais trim atvejais buvo realizuoti šablonai SystemC kodo generavimui iš UML modelių.

Siūlomas metodas skiriasi nuo minėtųjų anksčiau galimybe tikrinti UML diagramas kartu su papildomais sistemų ar posistemių modeliais. Statinė UML modelio analizė tegali suteikti informacijos, kuri gali padėti projektuotojui konstruojant konkretų kompozicinį modelį. Kodo generavimo metodai reikalauja radikalių modelio transformacijų norint tikrinti sistemos specifikaciją, parašyta UML. Tačiau nagrinėjamas metodas leidžia tiesiogiai tikrinti UML modelius įtraukiant juos į vykdomuosius sistemos modelius. Tuo pasiekiamas daug realesnis projektavimo procesas, nes pernelyg nepraktiška būtų tikėtis, kad sistema galės būti modeliuojama naudojant vien tik UML, o vėliau jos realizacija bus pilnai sugeneruota modeliais grindžiamo projektavimo metodais. Labiausiai tikėtinas scenarijus, kuriam ši sistema ir skirta, leidžia realizuoti heterogeninius aprašymus skirtingomis kalbomis ir skirtingus abstrakcijos lygmenis. Tokiu atveju, UML gali būti naudojas tik tada, kai jis yra geriausiai tinkantis metodas (pavyzdžiui, elgsenos šablonų modeliavimui).

Darbo tikslas – ištirti elgsenos projektavimo pavyzdžių integravimo galimybes į aktoriais paremtą paskirstyto vienlusčių sistemų modeliavimo aplinką.

Darbo uždaviniai:

1. Išanalizuoti į aktorius orientuoto modeliavimo metodiką
2. Išanalizuoti UML taikymo vienlustėse sistemose galimybes
3. Realizuoti elgsenos projektavimo pavyzdžių konstravimo ir modeliavimo įrankius
4. Ištirti realizacijos veikimą

2 Analitinė dalis

2.1 Metamodeliavimas

Metamodeliavimas yra žinių išgavimo iš duotos srities naudojant srities analizės metodus ir pateikimo aukštesniame abstrakcijos lygmenyje procesas. Metamodeliavimo arba kitaip tariant modeliais grįsto projektavimo principai yra plačiai naudojami projektuojant aparatūros ir įterptines sistemas.

Metamodeliavimas siekia surasti, suprasti ir užfiksuoti vidinę srities struktūrą bei sąryšius tarp srities dalių ir srities objektų. Reikia apibrėžti kontekstą, suprasti sąryšius tarp srities dalių ir išgauti sritį charakterizuojančias savybes. Metamodeliavimo rezultatas yra metamodelis – aukštesnio lygmens modelis, kuris aprašo sąryšius tarp žemesnio lygmens modelių ir jų dalių, projektavimo metodų abstrakcijų ir įrankių. Metamodelis yra srities modelių abstrakcijos ir apibendrinimo rezultatas. Jei modelį galime apibrėžti kaip tam tikrų realaus pasaulio reiškinių (objektų) abstrakciją, tuomet metamodelis yra aukštesnio lygmens abstrakcija, kuri apibrėžia paties modelio charakteristikas, struktūrą ir funkcionalumą. Modelis yra susijęs su savo metamodeliu panašiai kaip programa yra susijusi su programavimo kalbos, kuria ji yra parašyta, gramatika. Srities metamodelis išreiškia hierarchinę srities struktūrą, kur aukštesni lygmenys atitinka nuo srities nepriklausomas (esmines) sąvokas, o žemesni lygmenis atitinka specifines tos srities sąvokas.

Metamodeliavimas yra susijęs su objektiškai orientuota analize (OOA) ir gali būti naudojamas srities sąvokų klasifikacijai. Metamodeliavimo pavyzdys galėtų būti klasių identifikavimas analizuojant realaus pasaulio objektus ir jų sugrupavimas į skirtingas kategorijas (metaklases). Tai paprastai yra atliekama įvedant paveldėjimo hierarchiją. Pirmieji paveldėjimo hierarchijos lygmenys ir juose esantys ryšiai apibrėžia srities metamodelį. Konkretus modelis yra gaunamas iš metamodelio apibrėžiant konkrečias klases.

Metamodeliavimas taip pat gali būti naudojamas ne tik srities modelių analizei ir apibrėžimui, bet ir pačio modeliavimo proceso aprašymui [20]. Kitaip tariant, metamodeliavimas ne tik aprašo, ką mes modeliuojame, bet ir kaip mes modeliuojame. Šiuo atveju, metamodelis yra suprantamas kaip modeliavimo proceso modelis. Yra ir kitas panašus požiūris, kada metamodeliavimą apibrėžia kaip modeliavimo procesų suvokimo procesą. Pagal šią apibrėžtį, metamodelis yra konceptualus modeliavimo proceso (metodo) modelis, o metamodeliavimas yra srities kalbos modeliavimo procesas.

Pagal [20], metamodeliavimas turi tris matmenis:

1. Metamodeliavimas yra modeliavimo/specifikavimo kalbos modeliavimo procesas.
2. Metamodeliavimas yra žinių apie sritį/taikomąją srities sistemą abstrakcijos lygmenys.
3. Metamodeliavimas yra informacijos prie srities modelių taikymą ir naudojimą modeliavimas.

Pagrindinė metamodeliavimo užduotis yra atpažinti gerai suvoktas sritis, surasti gerai išbandytus modelius ir pritaikyti juos sistemos kūrimo metu. Gerai pažinti modeliai yra dažnai naudojamos aukšto lygmens projektavimo abstrakcijos, pvz., baigtiniai automatai (BA) skirti sudėtingų srities sistemų elgsenos aprašymui. Atsižvelgiant į tai, metamodeliavimą galima laikyti srities modelio supratimo ir kūrimo abstrakcijų ir taisyklių aprašymu.

Pagrindiniai metamodeliavimo iššūkiai yra šie:

1. Surasti ir aprašyti gerai išbandytus srities modelius ir šablonus, kuriuos paprastai naudoja srities projektuotojai.
2. Aprašyti gerai išbandytų srities modelių realizavimą naudojant aukšto lygmens abstrakcijas ir programavimo metodus.
3. Ieškoti (pusiau)automatinių gerai patikrintų modelių realizavimo metodų ir įrankių.

2.2 Metamodelių specifikavimas

Metamodelį galima nagrinėti 3 skirtingais požiūriais [21]:

1. Metamodelis kaip modelių kūrimo elementų ir taisyklių rinkinys.
2. Metamodelis kaip modeliuojamos srities modelis.
3. Metamodelis kaip kito modelio egzempliorius.

Metamodelis paprastai yra specifikuojamas naudojant unifikuotos modeliavimo kalbos (UML) poaibį. UML konstrukcijos naudojamos metamodeliuose yra šios: klasės, klasifikatoriai, asociacijos, asociacijos klasės, savybės, apribojimai ir operacijos.

Metamodelyje standartinės UML konstrukcijos yra panašios į UML konstrukcijas naudojamas modeliavimui. Asociacija yra semantinio ryšio tarp klasifikatorių, pvz., klasių, paskelbimas. Asociacijos klasė yra semantinių ryšių tarp klasifikatorių, kurie turi savo nuosavas savybes, paskelbimas. Atributas yra vardinė klasifikatoriaus būsena. Elgsenos savybė apibrėžia klasifikatoriaus elgsenos aspektą. Klasė aprašo objektų, kuria turi bendras savybes, įskaitant operacijas, atributus ir metodus, kurios yra bendros visai objektų aibei.

Klasifikatorius aprašo savybių rinkinį. Apribojimas yra būlinė išraiška, kuri yra priskirta su ja susietam modelio elementui, ir jos reikšmė turi būti lygi *true*. Savybė aprašo klasifikatoriaus egzemplioriaus arba paties klasifikatoriaus elgsenos arba struktūrinę charakteristiką. Operacija yra klasifikatoriaus egzemplioriaus elgsenos savybė, o atributas yra struktūrinis jo aspektas.

2.3 Objektinio modeliavimo pagrindinės koncepcijos

2.3.1 Objektinio modeliavimo sąvokos

Nors objektinio modeliavimo principai yra seniai žinomi ir taikomi aparatūros projektavimo srityje, objektinis aparatūros projektavimas pastaruoju metu susilaukia didelio tyrėjų ir projektuotojų susidomėjimo. Tikimasi, kad objektinio modeliavimo principų taikymas pakels abstrakcijos lygmenį ir padidins projektavimo našumą aparatūros srityje, kaip tai įvyko programų projektavimo srityje.

Objektinio ir aparatūros projektuotojams geriau žinomo komponentinio (blokinio) projektavimo principai yra palyginami 1 lentelėje.

Ypatybė	*Komponentinis*	*Objektinis*
Projektavimo kryptis	Iš viršaus į apačią	Iš apačios į viršų
Projektavimo objektas	Algoritmai	Duomenys
Abstrakcijos lygmuo	Procedūrų	Duomenų
Duomenys	Viešai prieinami	Paslėpti

1 lentelė. Objektinio ir komponentinio (struktūrinio) projektavimo metodų palyginimas

Pirmieji bandymai pritaikyti objektinio projektavimo metodus aparatūros projektavimo srityje nebuvo labai sėkmingi. Kai kurios objektinio modeliavimo idėjos buvo sėkmingai pritaikytos, pvz., sistemos išskaidymas į modulius, informacijos slėpimas, tuo tarpu kitos idėjos (pvz., paveldėjimas) nebuvo populiarios. Pagrindinė priežastis – tinkamų abstrakcijų (kalbų, metamodelių), galinčių aprašyti ir išreikšti objektinio modeliavimo sąvokas aparatūros srityje trūkumas. Atsiradus SystemC ir pritaikius UML aparatūros ir įterptinių sistemų sričiai, susidomėjimas objektinio modeliavimo principų pritaikymo aparatūros projektavimui vėl išaugo.

Objektinio modeliavimo metodai gali padidinti modeliuojamų sistemų ir jų komponentų lankstumą ir pakartotinį panaudojamumą. Tai leidžia greičiau kurti aukštesnės kokybės sistemas.

Objektinio modeliavimo paradigma teigia kad realaus pasaulio sistemas galima modeliuoti turint aibę klasių ir ryšius tarp jų. Siekiant aprašyti ir modeliuoti sritį ir joje esančias sistemas virš srities abstrakcijų lygmens įvedamas naujas abstrakcijos lygmuo. Pagrindinės šio aukštesnio abstrakcijos lygmens veikėjai yra klasės – labai abstrakčios struktūros, kurios apima srities duomenis ir su jais atliekamas operacijas. Klasės yra susietos viena su kita sąryšiu tinklu. Šie sąryšiai atvaizduoja skirtingus klasifikavimo ir sąveikos tarp srities esybių tipus. Objektinės srities analizės tikslas yra išskaidyti srities sistemą į atskiras klases (objektus) ir ryšius tarp jų.

Objektas yra esybė, kuri turi būseną ir apibrėžtą aibę operacijų, kurios keičia jo būseną ir atlieka konkrečius veiksmus. Būsena yra objekto savybių rinkinys. Su objektu susietos operacijos teikia servisus kitiems objektams, kurie jų reikalauja kai reikia atlikti kokius nors veiksmus. Objektai yra kuriami pagal objektų klasių apibrėžimą. Objekto klasės apibrėžimas naudojamas kaip objekto šablonas. Joje yra nurodytos visos savybės bei servisai kurie turėtų būti susieti su šios klasės objektu

Objektinio modeliavimo metodologija iš esmės yra evoliucinio projektavimo metodologija, kuri yra ypač orientuota į praktinį projektavimo pakartotiniam naudojimui principo taikymą. Klasės yra specialiai projektuojamos su galimybe vėliau išplėsti naudojant paveldėjimo mechanizmą.

Objektinio modeliavimo keturi pagrindiniai metodai yra: abstrahavimas, koncepcijų atskyrimas, komponavimas ir apibendrinimas. Abstrahavimas yra srities objektų, jų savybių, struktūros ir elgsenos atvaizdavimas abstraktesniu pavidalu, pvz., naudojant klases ir objektus. Koncepcijų atskyrimas yra atskiras srities aspektų specifikavimas ir realizavimas, pvz., atskiriant sąsajas nuo konkrečių paslaugų, metodus nuo duomenų ir pan. Kompozicija yra komponentų apjungimas siekiant gauti norimą projektuojamos sistemos elgseną. Komponentai gali būti sujungiami pranešimų perdavimu (asociacija) arba fiziškai įdedami vienas į kitą (agregacija). Apibendrinimas leidžia nustatyti ir apjungti bendras srities objektų (klasių) savybes, elgseną arba struktūrą naudojant klasių hierarchiją, klasių šablonus, polimorfizmą arba projektavimo šablonus.

Objektinio modeliavimo charakteristikas galima apibendrinti taip:

- Objektai yra realaus pasaulio arba kompiuterinės sistemos būsenų abstrakcijos ir valdo patys save.
- Objektas slepia savo vidinę būseną: taip pasiekiama objektų tarpusavio nepriklausomybė. Nepriklausomus objektus patogu naudoti pakartotinai, skirtingose programose.
- Sistemos funkcijos suvokiamos kaip atskirų objektų teikiamos paslaugos.

- Bendros duomenų sritys yra eliminuotos. Objektai bendrauja perduodami pranešimus.
- Objektai gali būti paskirstyti ir gali būti vykdomi nuosekliai arba lygiagrečiai.

2.3.2 Objektiniai modeliai ir ryšiai

Objektinio modeliavimo metodologijos pagrindas yra sistemos objektinis modelis. Modelis yra klasių hierarchijos egzempliorius sudarytas iš tarpusavyje komunikuojančių objektų. Objektas yra unikalus klasės egzempliorius, kuris yra struktūriškai identiškas kitiems tos klasės egzemplioriams. Jis apibrėžia savo teikiamų paslaugų (operacijų, metodų) sąsają, per kurią galima prieiti prie to objekto vidinių duomenų ir būsenos. Objektiniai modeliai turi du aspektus:

1. Semantinė informacija (*semantika*) - semantinis modelio aspektas aprašo sistemų kaip loginių konstrukcijų (klasių, asociacijų, būsenų, pranešimų, užduočių) tinklą. Semantinis modelis turi sintaksinę struktūrą, taisykles, nusakančias, kokie modeliai yra sintaksiškai teisingi ir vykdymo dinamiką.
2. Vizualinė pateiktis (*notacija*) - vizualizuojant modelį, semantinė informacija pateikiama pavidalu, pritaikytu peržiūrinėti ir redaguoti tą informaciją. Modelis pateikiamas žmogui lengvai suprantama forma.

Objektinio sistemos modelio funkcionalumą apibrėžia ryšiai tarp modelio elementų – objektų arba klasių. Skiriami trijų pagrindinių tipų ryšiai, kurie aprašo sistemų struktūrą ir elgseną: paveldėjimas, agregavimas ir asocijavimas. Paveldėjimas leidžia praplėsti klasę naujais duomenimis ir operacijomis. Agregacija yra naudojama sistemoms apjungti iš atskirų komponentų. Asociacija yra naudojama pranešimams keistis.

Objektinis srities sistemos modelis gali būti atvaizduotas trijuose abstrakcijos lygmenyse:

1) *realizavimo lygmenyje* – klasės atvaizduoja programos kodo dalį, parašytą objektine programavimo kalba;
2) *specifikavimo lygmenyje* – klasės specifikuoja sistemos sąsajas aukštesniame abstrakcijos lygmenyje;
3) *konceptualiajame lygmenyje* – klasės atvaizduoja abstrakčias tyrimo srities sąvokas, pvz., platformas.

Bendrinius objektinius modelius galima aprašyti naudojant polimorfizmą. Polimorfizmas leidžia manipuliuoti su skirtingų klasių objektais žinant tik jų bendras savybes ir nekreipiant dėmesio į konkretų klasės tipą. Tai leidžia vienodai traktuoti bazines klases ir

naujas per paveldėjimą sukurtas klases,kurios išplečia bazines klases naujais duomenimis ir operacijomis su jais.

Objektinis srities modelis leidžia konceptualiai apjungti aukštesnį ir žemesnį srities abstrakcijos lygmenis. Objektiniai modeliai pateikia tik abstraktų srities sistemų vaizdą nenurodant konkretaus srities turinio. Šį žemesnio lygmens srities turinį reikia įvesti vėliau, kai objektiniai modeliai yra detalizuojami į srities sistemas (esybes).

Įvedus du skirtingus abstrakcijos lygmenis reikia papildomai apibrėžti ir objektinį srities metamodelį, kuris aprašo aukštesniame abstrakcijos lygmenyje specifikuotų modelių transformavimą į žemesnio abstrakcijos lygmens modelius. Objektinio modelio realizacija priklauso nuo apibrėžto objektinio srities metamodelio ir transformacijų taisyklių. Viršutinio (objektinio) abstrakcijų lygmens modeliams aprašyti galima naudoti UML diagramas arba objektinio programavimo kalbą. Perėjimas į srities lygmenį gali būti atliktas naudojant įvairius srities kodo generavimo metodus.

Struktūriniams ryšiams tarp klasių ir elgsenos sąveikoms tarp objektų aprašymui objektiniame modelyje naudojami trijų pagrindinių tipų sąryšiai:

1. Paveldėjimas leidžia poklasei paveldėti duomenis ir operacijas apibrėžtas savo tėvinėje klasėje ir papildyti jas papildomais duomenimis ir metodais. Klasės gali būti sutvarkytos klasės hierarchijoje, kur viena klasė (superklasė) yra vienos ar kelių kitų klasių (poklasių) apibendrinimas. Poklasė paveldi atributus ir operacijas iš savo superklasės ir gali būti papildyta naujais metodais ir atributais. Paveldėjimas - tai abstrakcijos mechanizmas, kuris gali būti panaudotas įvairių esybių klasifikavimui. Be to, tai yra pakartotinio panaudojimo mechanizmas tiek projektavimo, tiek programavimo lygyje, o klasių hierarchija yra organizacinių žinių apie sritis ir sistemas šaltinis.
2. Agregacija yra naudojama kai vienas objektas (konteineris) fiziškai arba konceptualiai turi kitą objektą (komponentą). UML dar apibrėžia stipresnį agregacijos atvejį, vadinamą kompozicija. Ši ryšys stipresnis ta prasme, kad objektas-dalis vienu metu gali priklausyti tik vienam sudėtiniam objektui. Be to, jų gyvavimo trukmė sutampa, t.y. objektas-dalis yra sukuriamas ir sunaikinamas vienu metu kaip ir visas sudėtinis objektas.
3. Asociacija atvaizduoja konceptualius ryšius tarp klasių ir yra naudojama pranešimų keitimuisi tarp objektų, pvz., kai viena klasė “žino” apie kitą klasę ir naudoja jos operacijas arba atributus.

2.3.3 UML diagramos

UML (angl. Unified Modeling Language) – tai standartinė grafinė kalba, pritaikyta specifikuoti, vizualizuoti, projektuoti, konstruoti ir dokumentuoti artefaktus, sukuriamus, kuriant programų sistemas ir kitas neprogramines sistemas. UML modeliavimo kalba sparčiai populiarėja visame pasaulyje ir yra naudojama daugelio IT specialistų, kurie projektuoja programinę įrangą. UML aprašo sistemų analizės, projektavimo ir realizavimo aspektus. Sistemos yra abstrakčiai atvaizduojamos modeliais naudojant gerai apibrėžtą sąvokų žodyną ir taisykles. Sistemų modeliai gali būti aprašomi tiksliai, nedviprasmiškai ir išbaigtai naudojant UML diagramas. UML yra vizuali kalba, turinti apibrėžtą 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.

UML diagramos naudojamos trims skirtingiems tikslams: (1) modeliuoti realaus pasaulio sistemas (pvz., verslo sistemas); (2) sistemoms specifikuoti ir projektuoti koncepciniu ir architektūriniu lygmenimis; (3) kuriamoms sistemoms realizuoti: projektuoti eskiziniu ir detaliuoju lygmenimis.

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

1. Struktūrinės diagramos: klasių, objektų, komponentų ir diegimo diagramos.
2. Elgsenos diagramos: panaudos atvejų, sekų, veiksmų, bendradarbiavimo ir būsenų diagramos.
3. Modelio valdymo diagramos: paketų, posistemių ir modelių diagramos.

Projektavimo metu, skirtingos UML diagramos yra naudojamos skirtingiems tikslams. Labai svarbu žinoti, kad UML pateikia tik bendro pobūdžio notaciją, kuri turi būti pritaikyta proceso vystymo metu.

Klasių diagrama ko gero yra 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.

UML klasių diagramos atvaizduoja sistemos statinę struktūrą: sistemos objektus, ir statinius ryšius tarp jų. Naudojamos reikalavimų analizės metu modeliuoti probleminės srities sąvokas, sistemos projektavimo metu modeliuoti posistemes ir sąsajas, objektinio projektavimo metu modeliuoti klases.

Klasių diagramose naudojamos tokios sąvokos. Objektas - tai realaus pasaulio “daiktas” atliekantis tam tikrus apibrėžtus veiksmus. Klasė - tai objektų atliekančių panašius veiksmus abstrakcija, paprastai turinti kintamuosius (atributus) ir metodus (procedūrinį kodą). Atributas - tai vardinė klasės savybė aprašanti reikšmių diapazoną, kurį gali įgyti klasės būsena. Metodas - tai objekto teikiama paslauga Apribojimas - tai taisyklė apibrėžianti galimas objektų, klasių, ryšių būsenas.

Paketų diagramos dažniausiai naudojamos paketų aprašymui, leidžia nurodyti paketų struktūrą ir sudėtį. Sistemos elgsenos modeliavimas aukščiausiu abstrakcijos lygiu, prasideda nuo panaudos atvejų ir susijusių su sistema aktorių 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.

2.3.4 Aukšto lygmens modelių specifikavimas naudojant UML

UML leidžia aprašyti tris skirtingus sistemos modelius: funkcinis modelis (sistemos elgsena vartotojo požiūriu) objektinis modelis (sistemos struktūra) dinaminis modelis (vidinė sistemos elgsena).

Modelis yra realaus pasaulio objekto supaprastintas atvaizdavimas. Aukšto lygmens specifikavimo notacijos ir abstrakcijos įvedimas leidžia išryškinti svarbiausius modelio aspektus. Gero modelio sukūrimas yra būtinas kuriamos programinės įrangos kokybei užtikrinti. Sistemos modelis padeda vizualizuoti, specifikuoti, konstruoti ir dokumentuoti kuriamą sistemą, padeda atskleisti ir suprasti įvairius kuriamos sistemos aspektus, padeda sistemų projektuotojams bendrauti tarpusavyje, leidžia išvengti pernelyg didelio sudėtingumo.

Vaizdinio modeliavimo privalumai, lyginant su formaliosiomis specifikacijomis (kur naudojama specialiai tam tikslui sukurta srities kalba), programos (parašytos tikra programavimo kalba) tekstu arba aprašymu natūralia kalba yra šie:

- Lengviau suprasti sudėtingas sistemas. Kompiuterinių sistemų sudėtingumas nuolat auga. Didelės sistemos projektuotojams sunku “suturėti” galvoje visas produkto plonybes, atpažinti atskirų jo dalių sąveikos atvejus ir numatyti galimas tos sąveikos pasekmes.
- Supaprastinamas bendravimas tarp srities žinovų, analitikų, projektuotojų ir programuotojų. Vietoje srities terminologijos (žargono) arba prie konkrečios programavimo kalbos pririštų abstrakcijų naudojama viena universali sistemų

modeliavimo kalba, kuri turi būti aiški, vienareikšmė ir paprasta, kad esant reikalui, srities žinovas galėtų ją sparčiai išmokti.

- Vaizdinis modeliavimas leidžia atskirti sistemos atvaizdą aukštame abstrakcijos lygmenyje (architektūrą) nuo jos realizavimui naudojamos kalbos. Toks atskyrimas leidžia ją realizuoti galima naudojant skirtingas kalbas bei technologijas.
- Dėka geresnio srities sistemų suvokimo, gaunamos programos, bibliotekos ir komponentai, kuriuos lengva plėtoti ar naudoti pakartotinai.
- Modeliuojant galima iš anksto, dar nepradėjus fizinės sistemos realizacijos, pastebėti ribines sistemos darbo sąlygas ir "kritines vietas". Tai leidžia ištaisyti programas anksčiau projektavimo proceso metu ir padidinti kuriamų sistemų patikimumą.
- Esant visuotinai priimtam modeliavimo standartui, galima kurti pagalbines priemones ir įrankius, kurie automatizuoja dalį darbo, atliekamo kuriant ir plėtojant kompiuterinius produktus.

Projektavimo šablonas yra dažnai pasikartojančios projektavimo problemos apibendrintas sprendimas. Šablonų ypatybės yra priklausomybė nuo konteksto, universalumas, esmingumas, abstraktumas, praktiškumas ir kompoziciškumas. Projektavimo šablonus galima laikyti dar viena srities abstrakcija, kuri padeda išreikšti apibendrintą srities turinį. Projektavimo šablonus taip pat galima apibendrinti kaip metašablonus.

2.3.5 Privalumai ir trūkumai objektinio modeliavimo taikymo SoC projektavimui

UML diagramų taikymo aparatūros projektavimui sunkumai yra šie:

- Ryšių tarp aparatūros komponentų specifikavimas. Aparatūros projektuotojai yra įpratę naudoti blokines diagramos, kurios yra labiau orientuotos į ryšių tarp komponentų atvaizdavimą. Tačiau UML diagramos vaizduoja sistemą aukštesniame abstrakcijos lygmenyje ir yra orientuotos į komponentų pakartotinį naudojimą ir priderinimą.
- Bendrinio srities funkcionalumo specifikavimas. UML specifikacijos paprastai yra naudojamos konkrečių sistemų specifikavimui. Jos nelabai gerai tinka panašių sistemų šeimynų aprašymui ir kintamų srities dalių valdymui.
- Objektiškai orientuoto modelio transformavimo sudėtingumas. Tas pats objektiškai aprašytas modelis gali būti įvairiai transformuojamas į aparatūrinio lygmens aprašą. Projektuotojai gali šiuo atveju naudoti įvairius metamodelius ir transformavimo metodus.

- Objektinio modelio validavimo problema. Aparatūriniai modeliai turi būti validuojami žymiai tiksliau negu programų modeliai. Tačiau UML modelių problema yra ta, jog aprašant sistemą aukštame abstrakcijos lygmenyje daug realizavimo detalių yra paliekama vėliau realizuoti projektuotojui.
- Padidėjęs pradinis sistemų kūrimo laikas. Projektuotojams reikia priprasti naudoti naujus projektavimo metodus ir susikurti pagrindinių srities modelių biblioteką.

Be to, dar daug darbo reikia atlikti adaptuojant UML diagramas aparatūros ir įterptinių sistemų projektavimo sričiai, kuriant UML įrankius ir integruojant juos į projektavimo srautą, kol bus galima visiškai panaudoti objektinio ir modeliais grįsto aparatūros projektavimo teikiamus pranašumus.

OOP principų taikymo aparatūros projektavimo srityje pranašumai:

- Klasės leidžia suvienyti aparatūros ir programinės įrangos projektavimo metodologijas ir pakelti HW/SW projektavimą į aukštesnį abstrakcijos lygmenį.
- Projektavimo šablonai leidžia specifikuoti aparatūros projektavimo problemą aukštesniame abstrakcijos lygmenyje grafiškai naudojant abstraktų projektavimo patirties aprašymą. Tai leidžia greitai suprasti projektavimo problemą, taip padidinant suprantamumą. Šablonais grįstą projektavimą galima realizuoti naudojant automatinio validavimo ir kodo sukūrimo įrankius ir taip apdidinti pakartotinį panaudojimą, projektavimo kokybę ir našumą. Pakėlus abstrakcijos lygmenį galima efektyviai projektuoti sudėtingas aparatūros sistemas.
- Inkapsuliavimas leidžia atskirti duomenis nuo funkcionalumo ir realizuoti koncepcijų atskyrimo principą aparatūros projektavimo srityje.
- Paveldėjimas ir polimorfizmas leidžia žymiai padidinti projektavimo našumą, nes jie leidžia pakarotinai naudoti jau sukurtus komponentus.
- Pranešimų siuntimas yra daug žadanti OOP sąvoka, kuri galėtų būti naudojama vienlusčiuose tinkluose.

2.4 Į aktorius orientuoto projektavimo apibrėžimai ir pagrindinės sąvokos

Į aktorius orientuotas modeliavimas (projektavimas) yra sisteminio lygmens modeliavimo (projektavimo) metodas naudojant lygiagrečius į duomenų srautą orientuotus komponentus, vadinamus aktoriais. Aktoriai abstrakčiai nusako veikseną nepriklausomai nuo (neatsižvelgdami) žemo lygmens realizavimo konstrukcijų, tokių kaip funkcijų kvietimai,

gijos ar paskirstytojo skaičiavimo infrastruktūra. Įprastai į aktorius orientuoti modeliai kuriami lygiagrečių sistemų statinės struktūros atvaizdavimui.

Į aktorius orientuotame projektavime, aktorius yra elementarus funkcinis vienetas. Aktorius griežtai apibrėžtą interfeisą, kuris apibendrina vidinę būseną ir aktoriaus vykdymą bei apriboja aktoriaus bendravimą su aplinka. Iš išorės interfeisas turi prievadus (port), kurie yra tarsi aktoriaus komunikavimo jungtys, ir parametrus, kurie naudojami aktoriaus elgsenos konfigūravimui. Dažniausiai parametrų reikšmės yra tarsi dalis išankstinės aktoriaus konfigūracijos ir modelio vykdymo metu nekinta. Tačiau ne visada.

Svarbiausi į aktorius orientuotame projektavime yra komunikacijos kanalai, kuriais perduodami duomenys iš vienos sąsajos į kitą pagal kažkokią pranešimų apsikeitimo schemą. Panašiai kaip objektiškai orientuotame projektavime, kur komponentai sąveikauja perduodami kontrolę per metodų kvietimą, aktoriais pagrįstame projektavime, sąveika realizuojama perduodant žinutes per ryšio kanalus. Kanalų tarpininkavimas komunikuojant reiškia, kad aktoriai „bendrauja“ tik su tais aktoriais, su kuriais jie yra sujungti kanalais.

Kaip ir aktoriai, modeliai taip pat gali turėti išorinę sąsają, vadinama hierarchine abstrakcija. Šis sąsaja susideda iš išorinių prievadų ir išorinių parametrų, kurie yra nepriklausomi nuo modelyje naudojamų individualių aktorių prievadų ir parametrų. Šie išoriniai modelio prievadai gali būti kanalais sujungiami su kitų modelių išoriniais prievadais arba su modelį sudarančių aktorių prievadais. Išoriniai modelio parametrai gali būti naudojami modelį sudarančių aktorių parametrų reikšmių apibrėžimui.

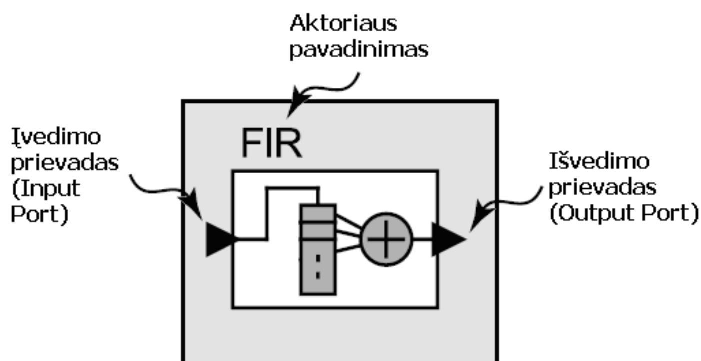
Tokios sąvokos, kaip modelis, aktorius, prievadas, parametras ir kanalas kartu sudaro abstrakčią į aktorius orientuoto projektavimo sintaksę. Abstrakčioji sintaksė nusako modelio struktūrą, tačiau nenusako kaip tas modelis funkcionuoja.

2.5 Į aktorius orientuoto projektavimo procesai, metodai ir įrankiai. Įrankių galimybių palyginimas. Specifikavimo kalbos/notacijos. Palyginimas su UML.

Ptolemy II yra atviro kodo sistema, skirta lygiagrečių sistemų modeliavimui, simuliavimui ir projektavimui. Ptolemy II simuliuojamų modelių pagrindas yra aktoriai, kurie bendrauja tarpusavyje siųsdami ir priimdami estafetės duomenų paketus per interfeisus, modeliuojamus kaip sąsajų rinkinys. Simuliuojami modeliai konstruojami ir simuliuojami pagal tam tikrą skaičiavimo modelį (MoC) kuris realizuotas kaip Ptolemy II sritis ir įtraukiamas į simuliuojamą modelį kaip *director* objektas. Skaičiavimo modelis – tai taisyklių

rinkinys, kuris lemia aktorių sąveiką apibrėždamas, kokią įtaką lygiagretumas ir laikas turi aktorių komunikavimui ir elgsenai. Ptolemy II sistemoje per sritis realizuojami įvairūs skaičiavimo modeliai. Keletas iš Ptolemy II realizuotų MoC pavyzdžių būtų nuoseklaus laiko semantika, naudojama Simulink (The MathWorks), duomenų tėkmės semantika ir LabVIEW (National Instruments), diskrečiųjų įvykių semantika iš OPNET Modeler (OPNET Technologies) ir daugelis kitų, kurių dalis yra eksperimentiniai.

Ptolemy II į aktorius orientuoto modeliavimo sintaksė leidžia konstruoti heterogeninius modelius. Jų dėka toje pačioje modeliuojamoje sistemoje naudojant hierarchinę kompoziciją galima atlikti eksperimentus su skirtingais lygiagretumo ir komunikavimo modeliais.

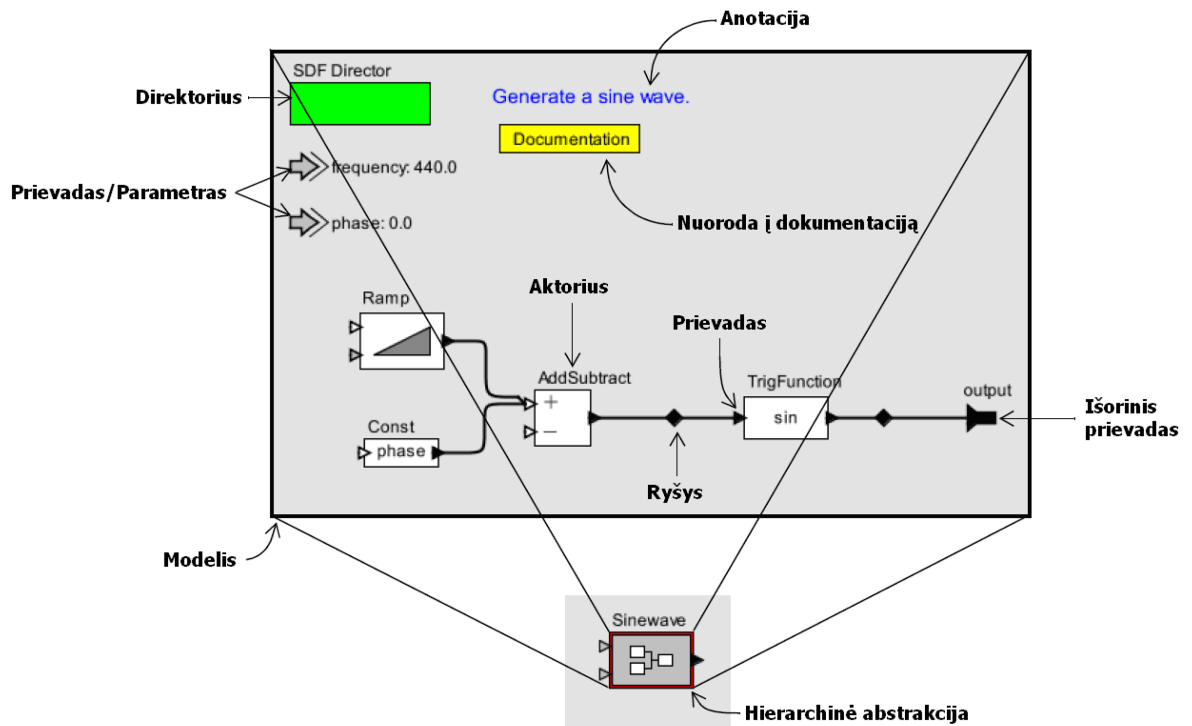


1 pav. Paprasčiausias aktoriaus pavyzdys Ptolemy II sistemoje. Parametrai ir vidinė būsena nėra vaizduojama.

Tam, kad būtų galima realizuoti heterogeniškumą, Edward A. Lee, Ptolemy Project grupės vadovas, visiškai atskyrė modelio sintaksę nuo modelio semantikos. Modelio struktūrinės savybės vaizduojamos naudojant į aktorius orientuoto projektavimo abstrakčiąją sintaksę. Aktoriaus objektai yra pakankamai abstraktūs ir gali talpinti aktorius, aprašytus skirtingomis kalbomis, kiekvienam aktoriui suteikiančiomis skirtingą semantinę prasmę. Ptolemy II sistemoje aktoriai gali būti aprašomi Java, C/C++, Cal, Matlab, Python ir kitomis kalbomis. Taip pat yra sistemos papildymų, kurie leidžia aprašyti aktorius VHDL, Verilog bei SystemC [15].

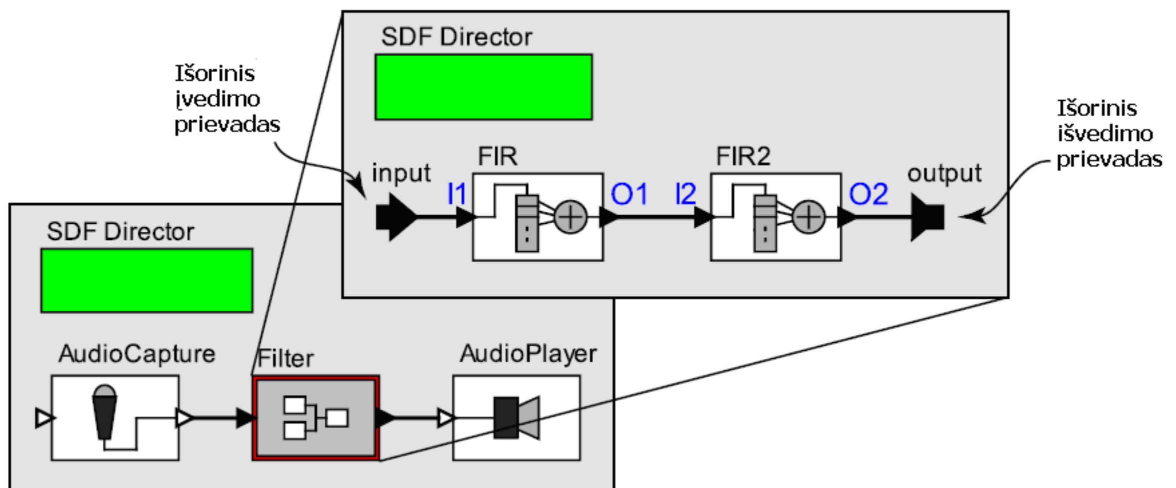
Esminė Ptolemy II savybė, kurios dėka realizuotas aktorių aprašymas UML kalba, yra galimybė hierarchiškai komponuoti heterogeninius modelius. Toks Lee pasiūlytas požiūris, kaip priešingybę dažniausiai naudojamam globaliam modelio vykdymo planui, leidžia kiekviename hierarchiniame lygmenyje priskirti skirtingą modelio vykdymo semantiką. Tai reiškia, kad aktoriai gali būti patalpinti į kitus aktorius, vadinamus sudėtiniais aktoriais, ir vidiniai aktoriai gali paveldėti vykdymo semantiką iš juos talpinančių aktorių arba jiems gali būti priskirta kitokia semantika nei talpinančiam aktoriui. Vykdymo semantika, dar

kitaip vadinama skaičiavimo modeliu, apibrėžia kaip ir kada kiekvienas aktorius gali komunikuoti, atlikti skaičiavimus ir atnaujinti savo vidinę būseną. Būtent šios išskirtinės Ptolemy II savybės išnaudojamos realizuojant UML elgsenos projektavimo pavyzdžių integravimą į Ptolemy II modelius.



2 pav. Į aktorius orientuoto modelių iliustracija (viršuje) ir jo hierarchinė abstrakcija (apačioje)

Aktorių kompozicija su kitais aktoriais naudojama sudėtinių aktorių arba modelių sudarymui. Ryšys tarp aktorių prievadų vaizduoja komunikacijos kanalus, kuriais persiunčiami estafetiniai duomenų pakertai iš vieno prievado į kitą. Kompozicijos semantika, įskaitant ir komunikavimo stilių, nusakoma skaičiavimo modeliu. Esant reikalui, skaičiavimo modelis vaizduojamas kaip nepriklausomas objektas – *director*. Dažnai modeliuose aprašomas išorinė aktoriaus sąsaja, leidžianti modelius panaudoti kompozicijoje su kitais modeliais. Kompozicinio aktoriaus pavyzdys matomas 2 paveikslėlyje, o kompozicinio aktoriaus panaudojimas modelyje kartu su paprastais aktoriais vaizduojamas 3 paveikslėlyje.



3 pav. Hierarchinės abstrakcijos panaudojimas kuriant heterogenines sistemas

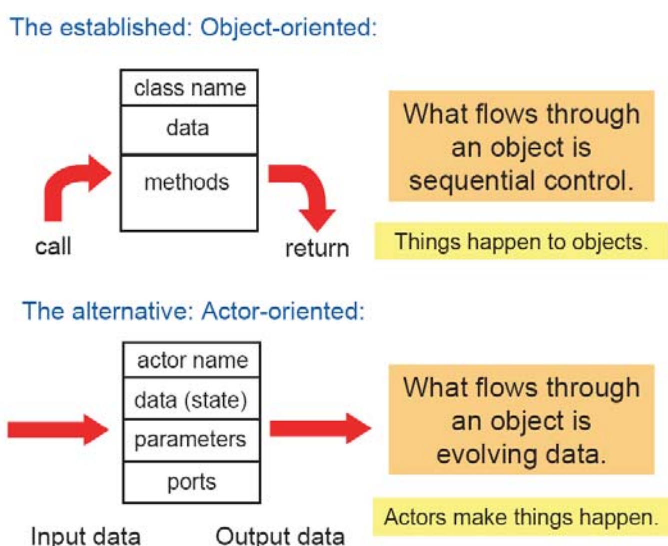
Be pagrindinių sistemos dalių, kurias sudaro paprasčiausi elementai aktorių konstravimui, sąsajos, ryšiai, estafetiniai duomenų paketai ir objektai *director*, Ptlemy taip pat turi turtingą specializuotų aktorių biblioteką, kuriais realizuojamos dažniausiai naudojamos funkcijos, pradedant nuo aritmetinių, loginių funkcijų ir baigiant sudėtingais signalų apdorojimo algoritmais. Visa sistema valdoma per grafinę vartotojo sąsaja, pavadinimu Vergil, kuri leidžia naršyti aktorių bibliotekose ir grafiškai konstruoti modelius. Vergil išsaugo modelius ASCII failuose naudodama XML failų struktūrą, vadinamą MoML. MoML yra pagrindinis PtolemyII modelių saugojimo failuose formatas. Taip pat tai pagrindinis modelių konstravimo mechanizmas, kurių aprašymas ir vykdymas yra paskirstytas tinkle.

```
<class name="Sinewave">
<property name="samplingFrequency" value="8000.0"/>
<property name="frequency" value="440.0"/>
<property name="phase" value="0.0"/>
<property name="SDF Director"
class="ptolemy.domains.sdf.kernel.SDFDirector"/>
<port name="output"><property name="output"/>
<entity name="Ramp" class="ptolemy.actor.lib.Ramp">
<property name="init" value="phase"/>
<property name="step" value="frequency*2*PI/samplingFrequency"/>
</entity>
<entity name="TrigFunction" class="ptolemy.actor.lib.TrigFunction">
<property name="function" value="sin"
class="ptolemy.kernel.util.StringAttribute"/>
</entity>
<relation name="relation"/>
<relation name="relation2"/>
<link port="output" relation="relation2"/>
<link port="Ramp.output" relation="relation"/>
<link port="TrigFunction.input" relation="relation"/>
<link port="TrigFunction.output" relation="relation2"/>
</class>
```

4 pav. Komponento, generuojančio sinusoide, XML aprašas MoML kalba

Šiuo metu Ptolemy II projektas yra koordinuojamas UC Berkeley universiteto Hibridinių ir Įterptinių sistemų centro, kuris gauna paramą iš Kalifornijos valstijos Nacionalinio Mokslo Fondo bei iš šių kompanijų: Agilent, DGIST, General Motors, Hewlett Packard, Microsoft, National Instruments ir Toyota.

Objektiškai orientuotame projektavime komponentai sąveikauja tarpusavyje perduodami kontrolę per metodų kvietimus, tuo tarpu į aktorius orientuotame projektavime jie sąveikauja persiųsdami pranešimus per ryšio kanalus. Tradiciniam objektiškai orientuotame projektavime, tai, kas „eina per objektus“ yra kontrolė, kitaip tariant objektams kažkas nutinka. Į aktorius orientuotame projektavime, tai, kas „eina per objektus“ yra besikeičiantys duomenys, t. y. aktoriai juos keičia.



5 pav. Objektiškai orientuoto projektavimo ir į aktorius orientuoto projektavimo skirtumai pagal Edward A. Lee

Pagrindinė į aktorius orientuoto projektavimo idėja yra ta, kad vidinė aktoriaus elgsena ir būsena yra paslėpta už aktoriaus sąsajų ir nėra matoma iš išorės. Ši griežto uždarumo savybė atskiria komponento elgseną nuo komponento sąveikos su kitais komponentais. Sistemų architektai gali projektuoti aukštame abstrakcijos lygyje atsižvelgdami į skirtingų skaičiavimo modelių elgsenos savybes nepriklausomai nuo komponentų elgsenos savybių. Be to, skirtingi skaičiavimo modeliai gali būti naudojami skirtinguose hierarchijos lygiuose, tokiu būdu realizuojant hierarchiškai nevienarūšę architektūrą.

Į aktorius orientuotas projektavimas bei kitos į žinutes orientuotos (mesage-oriented) sistemos yra gerai pritaikytos įterptinėms ar kitoms daug lygiagretumo turinčioms sistemoms,

kur reikia aptarnauti daugybę išorinių įrenginių ir pertraukimų išlaikant trumpą atsakymo laiką. Į aktorius orientuotas projektavimas gali būti apjungiamas su objektiškai orientuotu projektavimu ar kitomis į procedūras orientuotomis sistemomis siekiant išnaudoti geriausias jų savybes.

2.6 Vienlusčių sistemų į aktorius orientuotas projektavimas. Kitų autorių pasiekti rezultatai.

Terminas aktorius 1970 metais pirmą kartą buvo panaudotas Carl Hewitt apibrėžti autonomiškų mąstančių agentų sąvokai [29]. Agha ir kiti autoriai savo darbuose terminą pradėjo naudoti formalizuotų lygiagrečių skaičiavimų modeliams nusakyti. Kiekvienas Agha aktorius turi nepriklausoma kontrolės giją ir komunikuoja su kitais aktoriais per asinchronines žinutes. Edward A. Lee išplėtojo šią sąvoką, kad ji apimtų didesnes lygiagrečiųjų skaičiavimo modelių šeimas, kurių dauguma yra labiau apriboti nei vien tik žinučių persiuntimu. Aktoriai vis dar yra konceptualiai lygiagretūs, bet priešingai nei Agha aktoriams šiems nereikia turėti atskiros kontrolės gijos. Be to, nors vis dar komunikuojama per vienokios ar kitokios formos žinučių persiuntimus, šie nebūtinai turi būti asinchroniniai.

Į aktorius orientuotas programavimas ir objektiškai orientuotas programavimas yra vienas kitą papildantys, panašiai kaip Lauer ir Needham apibrėžia abipusį papildymą į žinutes orientuotų ir į procedūras orientuotų sistemų [34]. Lauer ir Needham teigia, kad nors „nei viena sistema nėra visais atžvilgiais tiksliai suderima su jos modeliu“[1], „dauguma modernių operacinių sistemų gali būti naudingai suklasifikuojama juos naudojant. Kai kurios sistemos yra realizuotos tokiu stiliumi, kuris yra labai artimas vienam ar kitam modeliui. Kitas sistemas galima suskaidyti į posistemes, kurių kiekviena atitinka vieną iš modelių, apjungtas išorinių interfeisų mechanizmais.“[2] Jie pateikia išvadą, kad „sistemos taikymo sritis nesuteikia jokio pagrindo realizuojamo modelio pasirinkimui. Galimus realizavimo modelius diktuoja

[1] “no real system precisely agrees with either model in all respects”

[2] “most modern operating systems can be usefully classified using them. Some systems are implemented in a style which is very close in spirit to one model or the other. Other systems are able to be partitioned into subsystems, each of which corresponds to one of the models, and which are coupled by explicit interface mechanisms.”

techninis sluoksnis, kuriame sistema realizuojama,“[3] „t.y. aparatinės įrangos architektūra ir/arba programavimo aplinka, kurias naudojant įgyvendinami procesai ir jų sinchronizavimo metodai. Sistemų projektavimo sprendimai, kurių pagrindu realizuojami procesai ir sinchronizacija, nulemia, kad vienas ar kitas realizavimo stilius tampa labiau patrauklus ar labiau atgrasus.“[4] Jie teigia, kad į žinutes (aktorius) orientuotas stilius geriausiai tinka, kuomet lengva išskirstyti žinučių blokus ir sudėti žinutes į eiles, bet sudėtinga sukurti apsaugotų procedūrinių kvietimų mechanizmą. Tuo tarpu kiti apribojimai yra „primesti mašinos architektūros ir aparatinės dalies“[5], tokie kaip „realios ir virtualios atminties organizacija, būsenos žodžio ilgis, kuris turi būti išsaugomas kiekvieną kartą keičiant kontekstą, planavimo ir paskirstymo lengvumas, išorinių įrenginių ir pertraukimų konfigūracija, instrukcijų rinkinio ir programuojamų registrų architektūra.“[6]

2.7 Vienlusčių sistemų projektavimas naudojant UML

Object Management Group (OMG) yra organizacija, atsakinga už UML ir su ja susietų metodikų vystymą ir standartizavimą. Pagrindinė modeliais grindžiamos architektūros propaguojama idėja teigia, kad projektavimas turi būti atliekamas nuo platformos nepriklausomu būdu. Sukurti sistemą aprašantys modeliai gali būti transformuojami į specializuotus tam tikrai platformai modelius atsižvelgiant į sistemos realizavimui keliamus reikalavimus ir apribojimus. Šios transformacijos atliekamos automatizuotai, jei ne iki galo, tai bent iki tam tikro laipsnio. UML yra kritinė šios metodologijos dalis, o jos antroji

[3] “the considerations for choosing which model to adopt in a given system are not found in the applications which that system is meant to support. Instead, they lie in the substrate upon which the system is built,”

[4] “i.e., machine architecture and/or programming environment—on which the process and synchronization facilities are implemented. The factors and design decisions of the system upon which the process and synchronization facilities are built are the things which make one or the other style more attractive or more tedious.”

[5] “imposed by the machine architecture and hardware,”

[6] “organization of real and virtual memory, the size of the stateword which must be saved on every context switch, the ease with which scheduling and dispatching can be done, the arrangement of peripheral devices and interrupts, and the arhitecture of the instruction set and the programmable registers.”

versija (2.0) sukurta siekiant patenkinti kalbos poreikį modeliais grindžiamų architektūrų aprašymui.

Aparatūros projektuotojai linkę naudoti modeliais grindžiamą projektavimą. Elektronikos pramonėje aparatūros aprašymo kalbos ar schematinio tipo projektavimas buvo dešimtis metų naudojama technologija. Be to, paskutiniai sisteminio lygmens projektavimo tyrimai, propaguojantys skaičiavimo ir komunikavimo atskyrimą, yra orientuoti į funkcinius aprašymus, kurie nėra pilnai paruošti įdiegimui į konkrečią platformą [30, 31]. Tad nesunku pastebėt, kad yra bendrų bruožų elektronikos sisteminio lygmens projektavimo ir UML pagrindu realizuojamų modelių evoliucijoje. UML adaptacija sparčiausiai vysto suinteresuotos verslo struktūros, dažniausiai propaguodamos UML ir SystemC apjungimą. Tyrėjų komanda iš Catania universiteto ir ST Microelectrinics kompanijos teigia, kad į UML reiktų žvelgi kaip į aukšto lygmens modeliavimo kalbą, tuo tarpu SystemC turėtų būti laikoma žemo lygmens sistemine kalba. Tam kad realizuoti sąryšį tarp šių kalbų, jie pasiūlė stereotipų rinkinį, kuris leidžia modeliuoti SystemC koncepcijas UML diagramomis. Stereotipai sugrupuoti ir pavadinti UML 2.0 SystemC palaikymo papildymu. Panašūs požiūriai, tik su skirtinais stereotipų apibrėžimais, buvo pasiūlyti tyrėjų iš Politecnico de Milano, Siemens ICM ir Fujitsu. Visais trim atvejais buvo realizuoti šablonai SystemC kodo generavimui iš UML modelių.

UML sintaksės praplėtimo poreikis buvo pastebėtas ne tik integruojant su kitomis modeliavimo kalbomis, tokiomis kaip SystemC, bet taip pat bandant apibrėžti sisteminio lygmens projektavimo dialektą UML kalba. Nekartą buvo pasirodę iniciatyvų sukurti UML papildymą laiko, našumo ir kitų kritinių SoC parametrų optimizavimui, kurių nebuvo galima aprašyti naudojant pagrindinę sintaksę. Tačiau šie papildymai sukūrė naują problemą – norint, kad įrankiai tarpusavyje būtų suderinti, reikalingas rimtas UML papildytos sintaksės standartizavimas. Yra daugybė konkuruojančių ir viens kitą papildančių UML profilių, bandančių išspręsti šią problemą. dėl ko labai sunku sekti kiekvieno jų evoliucionavimą. Pavyzdžiui, OMG priėmė UML papildymą veiksmų planavimo, našumo ir laiko modeliavimui, tačiau mažiau nei po dviejų metų priimtasis papildymas pradėtas laikyti pasenusiu ir yra ruošiamas naujas papildymas – MARTE. Tam, kad viskas būtų dar labiau komplikuota, keletas UML papildymų tampa tarsi atskiros kalbos: įvedami nauji žymėjimai, o kai kurios UML konstrukcijos, kurios nėra tiesiogiai susiję su analizuojama sritimi, tiesiog išmetamos. Tokios kalbos pavyzdys galėtų būti SysML [32], kuri įdiegta į daugybę programų ir yra remiama daugelio kompanijų ir vyriausybinių organizacijų. Kalbą propaguojantis konsorciumas bando pateikti pagrindinius modeliavimo primityvus, kurių dalis būtų iš UML 2.0 bei papildomai keletą praplėtimų sistemų inžinerijai. Net jeigu tai stokoja specifinių

vienlusčių sistemų modeliavimo konstrukcijų, tai galima pritaikyti reikalavimų inžinerijai ir projektavimo valdymui.

Dabartinis UML adaptavimas SoC projektavimui stokoja projektuotojų pripažinimo. Nors jis palankiai vertinamas organizacijų dėl standartizavimo ir dokumentavimo, projektuotojai įžvelgia tame daugiau apsunkinimą nei naudą. Gal būt projektuotojai tai priimtų, bet kaip papildomą paketą, kuris puikiai pritaptų prie kasdienės praktikos ir padėtų dirbti. Būtent tokiu būdu UML pelnė pripažinimą programinės įrangos kūrimo bendruomenėje. Ne savo galimybėmis dokumentuoti ar generuoti kodą, o galimybe nusakyti architektūrinius sprendimus, kuriuos galima taikyti sudėtingoms sistemoms. Šie sprendimai, vadinami projektavimo pavyzdžiais, galbūt buvo pagrindinė priežastis, kuri privertė daugybę programuotojų ir programinės įrangos kūrėjų įsisavinti UML tam, kad būtų galima projektuoti abstrakčiau ir analizuoti sąveiką tarp sistemos dalių aukštesniame lygyje negu programos kodo.

Būtent šiuo metu SoC projektuotojai susiduria su tokiu pat scenarijumi. Didėjantis sistemų sudėtingumas reikalauja daugiau dėmesio architektūriniams sprendimams, ko pasėkoje posistemių bendravimas tampa kritine vieta našumo, ploto ir energijos suvartojimo atžvilgiu. Tad sprendimai, leidžiantys projektuotojams abstraktesnį sistemų vystymą pakartotinai panaudojant architektūros šablonus skirtinguose projektuose yra labai laukiami (sisteminio lygmens aprašymo kalbos, tokios kaip SystemC būtent tai atlieka).

Lyginant su UML adaptavimo scenarijumi programinės įrangos kūrimo bendruomenėje, adaptavimas SoC projektavime turėtų būti lengvesnis, nes priešingai nei programinės įrangos programuotojai, dauguma SoC projektuotojų jau įpratę pasitikėti modeliais ir jų transformacijomis projektuojant. Tačiau egzistuoja svarbus trūkumas, kuris turėtų būti išspręstas norint adaptuoti UML SoC projektavime – tai UML sintaksės simuliavimo semantikos stoka. SoC projektuotojai kuria modelius, kuriuos gali simuliuoti įvairiuose abstrakcijos lygiuose analizuodami spartą, vėlinimus ir kitus aktualius parametrus. Programinės įrangos projektuotojai dažnai atlieka statinę UML modelių analizę tikrindami teisingumą ar kitus funkcinius parametrus, tačiau našumo rezultatai dažniausiai sužinomi po pirmo sistemos paleidimo. Nors yra bandymų išnaudoti UML modelių statinę analizę SoC projektavimui [33], tačiau pasiekti rezultatai yra labai riboti. Tam, kad geriausiai pritapti prie dabartinių SoC projektavimo metodų, geriausia alternatyva būtų adaptuoti modeliavimo semantiką UML sintaksei suteikiant galimybę modeliams būti vykdomiems laike. Tačiau nei vienas anksčiau paminėtų sprendimų nepaliečia šios problemos tiesiogiai.

Vienas žingsnis šia linkme yra vykdomasis UML (xUML – executable UML) [12], kuris remiasi veiksmų sąvoka, pristatyta paskutinėje UML specifikacijoje. Panašiai kaip

SysML, xUML gali būti laikoma kita kalba, nes ji palaiko tik tris UML konstrukcijas: klasių, būsenų ir veiksmų (actions) diagramas, įgyvendintas per nestandartizuotą veiksmų kalbą. Naudodamiesi šiomis konstrukcijomis, projektuotojai gali kurti modelius, kurie gali būti sukompiliuoti vykdomąjį kodą, skirtą tam tikrai platformai. Šį požiūrį realizuojančių kodo kompiliatorių yra nedaug, tad xUML adaptavimas SoC projektavimui reikalauja tolesnių tyrimų kuriant modelių generatorius, galinčius generuoti sintezuojamą kodą. Didesnė problema yra poreikis simuliuoti xUML modelius kartu su senu HDL kodu ar kitais sistemų modeliais, nes labai dažnai sistemos dizainas remiasi pakartotiniu prieš tai sukurtų komponentų panaudojimu.

3 Projektinė dalis

3.1 Ptolemy II sistemos architektūra ir galimybės.

Ptolemy II infrastruktūra palaiko keletą skaičiavimo modelių. Visa architektūra susideda iš paketų rinkinio, kurie teikia bendrą pagrindą visiems skaičiavimo modeliams ir iš specializuoto paketų rinkinio, kuris teikia labiau specializuotą pagrindą tam tikriems skaičiavimo modeliams. Pirmųjų pavyzdys gali būti tokie paketai, kurie turi matematinių funkcijų bibliotekas, grafų algoritmus, išraiškų interpretavimo kalbą, signalų vaizdavimo įrankius ir sąsajų su išoriniais įrenginiais valdymo bibliotekas. Antrųjų pavyzdys gali būti paketai, kurie palaiko modelių vaizdavimą paskirstytaisiais grafais, palaiko vykdomuosius sistemų modelius ir sritis. Šie paketai realizuoja konkretų skaičiavimo modelį.

Ptolemy II realizuotas moduliais, pagal kruopščiai suprojektuotą paketų struktūrą, palaikančia sluoksninį modelių analizavimą. Branduolio paketai realizuoja duomenų modelį, kitaip tariant abstrakčią sintaksę Ptolemy II projekte. Jais taip pat realizuojama abstrakti semantika, kuri leidžia veikti skirtingoms sritims maksimaliai apsaugant vidinius duomenis. Vartotojo sąsajos paketai palaiko XML failų formatą, kuris sistemoje vadinamas MoML ir grafinę vartotojo aplinką modelių kūrimui. Bibliotekų paketuose yra aktorių bibliotekos, kurios yra polimorfinės sričių atžvilgiu, t.y. jos gali būti naudojamos skirtingose srityse. Ir galiausiai sričių paketai, kuriuose realizuotos sritys, kurių kiekviena turi savo skaičiavimo modelį ir kai kurios iš jų savas, tik tai sričiai būdingų aktorių bibliotekas.

Keletas pagrindinių Ptolemy II klasių yra parodyta 6 paveikslėlyje. Tai UML statinės struktūros diagrama. Pagrindiniai sintaksės elementai yra stačiakampiai,

vaizduojantys klases, tuščiavidurės rodyklės, vaizduojančios paveldėjimą ir kitos rodyklės, vaizduojančios asociaciją. Kai kurios linijos turi mažus rombus, kurie vaizduoja agregaciją.

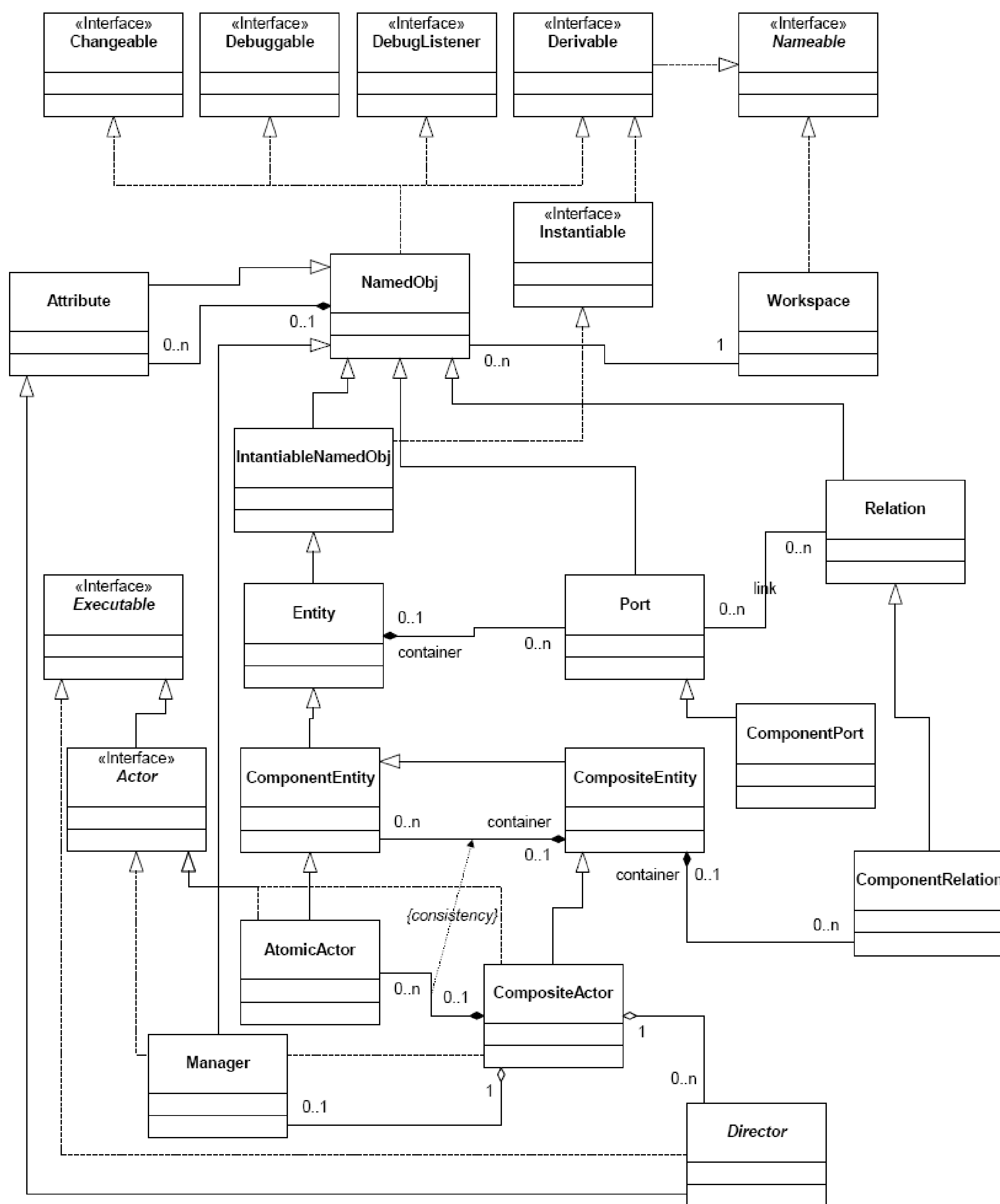
Visi objektai, kurie gali būti vaizduojami ekrane, gali turėti vardus – jie realizuoja Nameable interfeisą. Dauguma klasių paveldi NamedObj, kuris be to, kad gali turėti vardą, gali turėti su juo susietų atributų. Patys atributai taip pat yra NamedObj objektai.

Entity, Port ir Relation yra trys pagrindinės klasės, kurios tiesiogiai ar netiesiogiai paveldi NamedObj. Šios klasės sudaro Ptolemy II abstrakčią sintaksę. ComponentPort, ComonentRelation ir ComponentEntity papildo šias klases paskirstytųjų grafų palaikymo savybėmis. CompositeEntity paveldi ComponentEntity ir tuo pat metu turi agregacijos ryšį su ComponentEntity ir ComponentRelation objektais.

Interfeisas Executable naudojamas objektuose, kurie gali būti vykdomi. Interfeisas Actor paveldi Executable įgydamas galimybę persiųsti duomenis per prievadus, vadinamus portais. AtomicActor ir CompositeActor yra konkrečios klasės, kurios realizuoja Actor ir jo paveldėtą Executable interfeisus. Executable ir Actor interfeisai yra esminės Ptolemy II abstrakčios semantikos dalys.

Vykdomasis Ptolemy II modelis susideda iš aukščiausiame lygmenyje patalpinamo CompositeActor, kuriame yra tarpusavyje susieti objektai Director ir Manager. Objektas Manager kontroliuoja modelio vykdymą (paleidimą, sustabdymą, tęsimą). Objekte Direktor realizuota skaičiavimo modelio semantinė prasmė, nusakanti aktorių, kurie priklauso CompositeActor komponentui, vykdymo metodą.

Director yra pamatinė klasė kuriant naujus objektus, vadinamus *director*, kuriuose realizuojamas skaičiavimo modelis. Kiekvienas tok *director* objektas yra asocijuotas su tam tikra sritimi.



6 pav. Keletas pagrindinių Ptolemy II klasių, apibrėžiančių Ptolemy II abstrakčią sintaksę ir semantiką. Jos aprašytos kernel, kernel.util ir actor paketuose.

Ptolemy II yra trečios kartos sistema. Jos artimiausias pirmtakas, Ptolemy Classic, vis dar turi aktyvių vartotojų ir kūrėjų, daugiausiai dėl komercinio produkto, kuris yra realizuotas jo pagrindu, Agilent ADS. Ptolemy II yra gerokai išsiskiriantis iš ankstesniųjų versijų, o dėl Javos, lygiagretumo ir integravimo su tinklais, yra daugiau eksperimentinė versija. Keletas svarbiausių Ptolemy II galimybių, kurios tikima taps naujomis technologijomis modeliavimo ir projektavimo procese yra šios:

- ✓ *Aukštesnio lygmens lygiagretus projektavimas Java kalba.* Java lygiagretus projektavimas yra labai žemo lygmens, realizuotas gijomis (threads) ir stebėtojais (monitors). Tuo pat metu išlaikyti saugumą ir lankstumą gali būti pakankamai sunku [7][8]. Ptolemy II turi sričių (domenų) kurie palaiko lygiagrečių sistemų projektavimą daug aukštesniame abstrakcijos lygyje – šių sistemų programinės įrangos architektūros lygmenyje. Kai kurios šių sričių naudoja Java gijas kaip pamatinį mechanizmą, tuo tarpu kiti siūlo daug efektyvesnes, labiau praplečiamas ir suprantamas alternatyvas.
- ✓ *Pilnai realizuota polimorfinių duomenų tipų sistema.* Ptolemy Classic palaikė tik elementarų polimorfizmą per „anytype“ (bet kuris tipas) elementą. Netgi su šiuo ribotu polimorfizumu, tipų išskyrimo idėja pasirodė perspektyvi, tačiau realizavimas buvo silpnai išplėtotas. Ptolemy II turi modernesnę duomenų tipų sistemą pagrįstą daliniais tipais. Tipų išskaidymas atliekamas randant fiksuotą tašką naudojant algoritmus pagal [10]. Detaliau tipų sistemos yra aprašytos [15] ir [16].
- ✓ *Polimorfiniai aktoriai sričių atžvilgiu.* Aktorių bibliotekos buvo atskiros kiekvienai sričiai Ptolemy Classic projekte. Per posričių (subdomains) sistemą aktoriai galėjo veikti daugiau negu vienoje srityje. Ptolemy II ši idėja išvystyta daug pažangiau. Aktoriai su būdingu polimorfiniu funkcionalumu gali būti paršyti taip, kad dirbtų daugelyje sričių. Mechanizmas, kuris naudojamas komunikuoti su kitais aktoriais priklauso nuo srities, kurioje jie yra naudojami. Tai atliekama per architekrūrinį sprendimą, vadinamą *elgesio tipų sistema* [9].
- ✓ *Išplečiamas, XML pagrindu failų frormatas.* XML yra pripažintas standartas informacijos, kuri aprašo loginius ryšius tarp duomenų, vaizdavimui. Žmogui suprantamas failo formatas yra generuojamas stiliaus šablonų pagalba. XML failai Ptolemy II projekte yra naudojami kaip pagrindinis duomenų saugojimo formatas.
- ✓ *Geresnis suskaidymas į modulius naudojant paketus.* Ptolemy II yra suskirstytas į paketus, kurie gali būti naudojami nepriklausomai bei gali būti išskirstyti tinkle. Tai prieštarauja tradicinei programinės įrangos architektūrai, kai įrankiai dažniausiai yra integruoti į didžiulę sistemą su tarpusavyje priklausančiais elementais.
- ✓ *Visiškas atskyrimas abstrakčios sintaksės nuo semantikos.* Ptolemy projektai yra struktūrizuoti kaip paskirstyti grafai. Ptolemy II apibrėžia aiškią ir visapusiškai išsamią abstrakčią sintaksę šiems grafams bei atskiria ją nuo mechanizmų, kurie prideda grafams semantiką (analoginės schemos, baigtinių būsenų automatai ir t.t.).
- ✓ *Thread-safe lygiagretus vykdymas.* Įprastai Ptolemy modeliai yra lygiagretūs. Tačiau praeityje lygiagretus modelių vykdymas buvo labai primityvus. Ptolemy II visapusiškai palaiko lygiagretumą, leisdamas keisti atskirą modelio dalį kai tuo pat

metu vartotojo sąsaja keičia struktūra kitu būdu. Vientisumas yra išlaikomas monitoriais ir skaitymo/rašymo semaforais [5] realizuotais žemesnio lygio Java sinchronizavimo įrankiais.

- ✓ *Pilnai integruota išraiškų kalba.* Ptolemy II išraiškų kalba yra aukštesnio lygio, turtinga funkcijomis ir yra pilnai pritaikyta aktoriais paremtam modeliavimui. Tipų sistemos suderinamumo mechanizmas sklandžiai veikia tiek su išraiškų, tiek su parametrų, tiek su aktorių prievadų (portų) komponentais.
- ✓ *Programos architektūra pagrįsta objektinio modeliavimo technologija.* Nuo to laiko, kai buvo sukurtas Ptolemy Classic, programinės įrangos kūrime atsirado modernios objektinio modeliavimo ir projektavimo šablonų technologijos. Ptolemy II realizuotas naudojant šias technologijas, kas lėmė vientisesnį, aiškesnį ir labiau patikimą dizainą.

Ptolemy II turi keletą vis vystomų eksperimentinių galimybių:

- ✓ *Paskirstytieji modeliai.* Ptolemy II realizuota infrastruktūra palaikanti paskirstytuosius modelius naudojant CORBA arba Java RMI. Taip pat Ptolemy II palaiko migruojančius programinės įrangos komponentus.
- ✓ *Aukštesnio lygmens komponentai.* Ptolemy II turi aktorių biblioteką, kurie gali atlikti operacijas ne tik su įprastais duomenimis, bet ir su Ptolemy II modeliais. Nors Ptolemy Classic projekte taip pat realizuoti aukštesnio lygmens komponentai, taičiau veiksmai su modeliais buvo atliekami modelio vykdymo pradžioje. Ptolemy II jie gali būti atliekami bet kuriuo momentu.
- ✓ *Modelio gyvavimo trukmės valdymo komponentai.* Yra sukurta aktorių biblioteka, kurie valdo modelių gyvavimo trukmę. Pavyzdžiui RunCompositeActor kiekvieną kartą įvykus įvykiui pilnai įvykdo aktoriuje aprašytą modelį, o ModelReference įvykdo modelį, aprašytą atskirame faile, kuris gali būti patalpintas netgi internete.
- ✓ *Komponentų specializavimas.* Ptolemy II yra gerai išvystyti kodo generavimo mechanizmai smarkiai besiskiriantys nuo naudojamų Ptolemy Classic pakete. Kiekvienas Ptolemy Classic komponentas privalėjo turėti aprašą tikslo kalba ir kodo generatorius paprasčiausiais sujungdavo šiuos aprašus. Tuo tarpu Ptolemy II projekte komponentai yra aprašyti Java kalba ir būtent šis aprašas yra nagrinėjamas generuojant kodą. Realizuota aplikacijų programavimo sąsaja atlieka abstrakčios sintaksės modelių transformacijų optimizaciją, po ko naudojamas kompiliatorius galutinio kodo generavimui. Šios funkcijos detalesnis aprašymas pateiktas [11], [12], [13] ir [14].
- ✓ ***Cal*** *aktorių aprašymo kalba.* Tradiciškai Ptolemy II projekte aktoriai aprašomi Java kalba. Tačiau statinė Java programų analizė savybių/parametrų atžvilgiu, kas yra aktualu į aktorius orientuotose modeliuose, geriausiu atveju yra labai sudėtinga, o

blogiausiu iš vis neįmanoma. Cal aktorių aprašymo kalba leidžia realizuoti aktorius taip, kad jų aprašai nuo savybių tampa statiškai atskirti.

- ✓ *Eksperimentiniai skaičiavimo modeliai.* Ptolemy II realizuota keletas eksperimentinių skaičiavimo modelių, tame tarpe Giotto [4], paskirstytųjų diskrečiųjų įvykių [3], push-pull komponentų modelis [17] ir t.t.

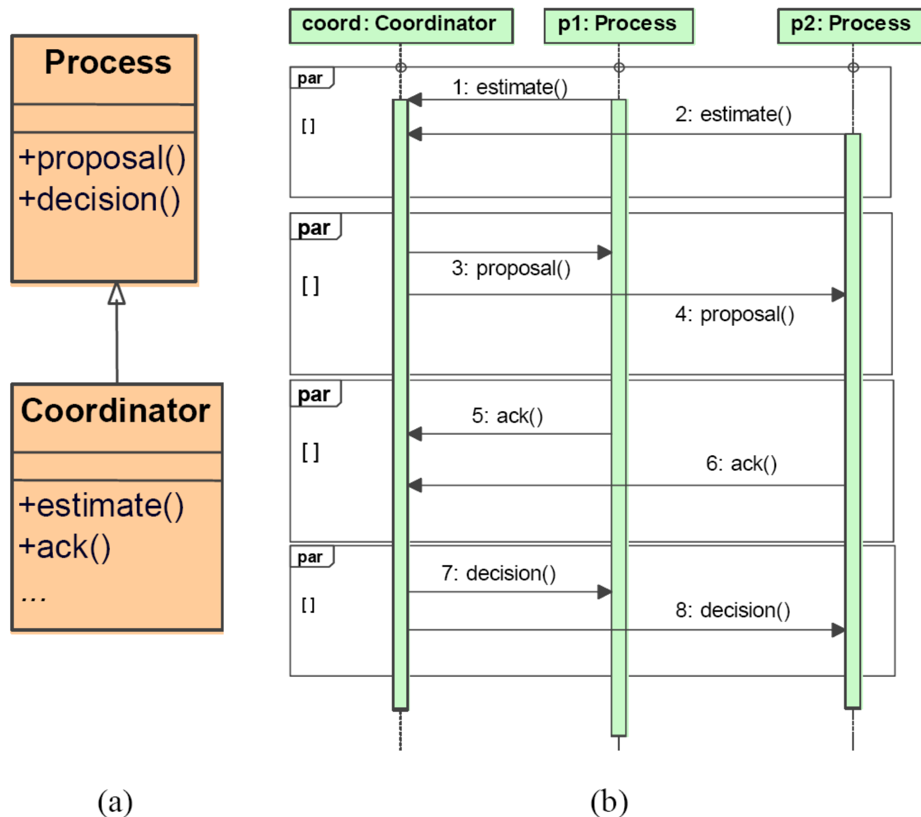
Ateityje tikimasi įdiegti šias naujas funkcijas:

- ✓ *Integruoti verifikavimo įrankiai.* Modernūs modelių verifikavimo įrankiai, galintys patikrinti bent jau baigtinių būsenų automatų modelius, galėtų būti integruoti į Ptolemy II aplinką. Taip pat tikima, kad valdymo logikos atskyrimas nuo lygiagretumo, smarkiai palengvintų verifikavimą.
- ✓ *Dinamikos atvaizdavimas.* Java palaiko statinės struktūros atvaizdavimą, bet ne procesais paremtų objektų dinamines savybes. Pavyzdžiui, duomenų struktūrą, kuri reikalinga komponentų bendravimui galima lengvai atvaizduoti, tačiau protokolo ne. Planuojama papildyti žymėjimo sistemą taip, kad būtų galima atvaizduoti šias dinamines objektų savybes.
- ✓ *Metamodeliavimas.* Ptolemy II sritys realizuotos intuityviu supratimu, kurios klasės bus naudingos modeliavime ir tuomet modelių specifikavimui ir vykdymui reikalinga infrastruktūra parašoma rankiniu būdu Java kalba. Yra sukurti įrankiai turintys potencialią galimybę pagerinti šią situaciją naudojant metamodeliavimą. Pavyzdžiui, Dome ir GME [6] aplinkose, pirmiausiai yra sumodeliuojama pati modeliavimo strategija ir tuomet vartotojo sąsaja, reikalinga atlikti šį modeliavimą, yra susintezuojama automatiškai iš modelio. Šiam tikslui pradžiai galima naudoti komponentais pagrįstą grafinę aplinką Vergil. Ateityje tikimasi, kad pats Ptolemy II bus naudojamas projektuoti ir realizuoti Ptolemy II sritis ir jų interfeisus.

3.2 Į aktorius orientuoto ir objektinio sistemų projektavimo integravimo modelis.

Tradiciškai elgsenos projektavimo pavyzdžiai aprašomi UML klasių ir sekų diagramomis. Klasių diagramomis nusakomi šablono dalyviai ir kokias žinutes kiekvienas jų gali priimti. Tuo tarpu sekų diagramomis detaliai atvaizduojama pranešimų pasirodymo seka ir ryšys tarp jų. 7 paveiksle pavaizduotas paprastas Chandra-Toueg algoritmo [18] paskirstytojo konsensuso šablono (distributed consensus pattern) pavyzdys, kuris yra plačiai naudojamas gedimams atspariose paskirstytosiose sistemose. Klasių diagrama (7 paveikslėlis

a dalis) vaizduoja dvi dalyvaujančias klases – Coordinator ir Process, jų sąryšį tarpusavyje (paveldėjimą) bei pranešimų antraštes. Šablono dinamika matoma 7 paveikslėlio b dalyje. Čia vertikaliomis linijomis vaizduojamas klasių (pavaizduotų 7 paveikslėlio a dalyje) objektų gyvavimo laikotarpis, o rodyklėmis – tarp objektų siunčiami asinchroniniai pranešimai.



7 pav. Elgsenos projektavimo pavyzdys UML kalba

Sekų diagramų integravimui į aktoriais paremtus modelius reikalingos šios prielaidos:

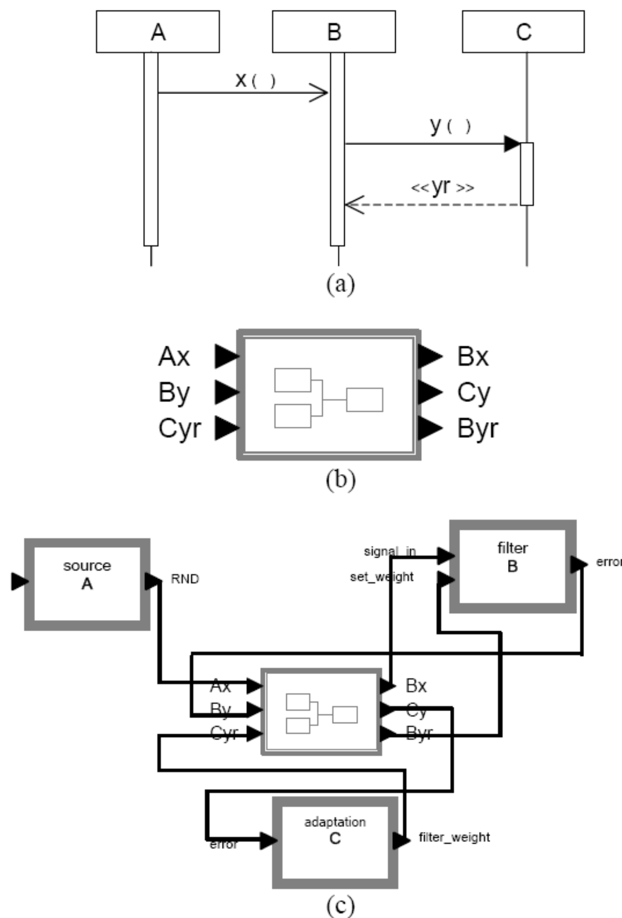
- (1) visos sekų diagramos yra sudėtiniame aktoriuje (composite actor);
- (2) kiekvienas klasės objektas, kuriam sekų diagramoje yra vaizduojama gyvavimo linija, statiškai ir vienareikšmiškai (vienas objektas su viena linija) susiejamas su aktoriumi, kuris yra patalpintas į sudėtinį aktorių, atitinkantį pirmąją prielaidą;
- (3) kiekvienas pranešimas tarp sekų diagramos objektų yra statiškai atvaizduojamas kaip prievadas (port) ir ryšys (relation) su atitinkamu aktoriumi tokiu būdu:
    - (3.1) asinchroniniai pranešimai vaizduojami sukuriant:
        - išvedimo prievadą aktoriuje, susietam su objektu, siunčiančiu pranešimą;
        - įvedimo prievadą aktoriuje, susietam su objektu, kuris priima siunčiamą pranešimą;
        - ryšiu tarp jų.

- (3.2) sinchroniniai pranešimai vaizduojami taip pat kaip ir asinchroniniai, tačiau papildomai sukuriant:
  - įvedimo prievadą aktoriuje, susietam su objektu, siunčiančiu pranešimą;
  - išvedimo prievadą aktoriuje, susietam su objektu, kuris priima siunčiamą pranešimą;
  - ryšiu tarp jų.

  Šie trys papildomi komponentai reikalingi grįžtamojo ryšio duomenims perduoti.

8 paveikslėlyje vaizduojamas modelis, kuriame matyti visos minėtosios prielaidos. Paprasčiausia sekų diagrama, kurioje yra viena asinchroninė ir viena sinchroninė žinutė, matyti 8 paveikslėlio a dalyje. Diagrama yra patalpinta aktoriuje, kuris matyti 8 paveikslėlio b dalyje.

Anksčiau minėtosios prielaidos yra aprašytos nagrinėjant du skirtingus sekų diagramų integravimo metodus į aktoriais paremtus sistemų modelius [18] ir [19] dokumentuose. Pirmajame, sekų diagrama yra integruojama į modelį kaip sudėtinis aktorius, kuris unikaliai siunčia/priima pranešimus į/iš prievadų pagal jame aprašytą sekų diagramą. Aktorių ir gyvavimo linijų susiejimas atliekamas sukuriant ryšį tarp aktorių prievadų, kurie yra susieti su siunčiama žinute (8 paveikslėlio c dalis). Tuo tarpu antrajame, sekų diagrama realizuojama kaip komponento *director*, susieto su sudėtiniu aktoriumi, atributas. Tai reiškia, kad sekų diagramoje objektų gyvavimo linijos sukuriamos atitinkamai pagal sudėtiniame aktoriuje esančius aktorius ir pranešimų siuntimo seka yra kontroliuojama objekto *director*.



8 pav. Sekų diagramos integravimas į aktoriais paremtą modelį

Nors minėtosios prielaidos yra pakankamos paprastų elgsenos projektavimo pavyzdžių integravimui į aktoriais paremtus modelius, kai kurie apribojimai vis dar išlieka:

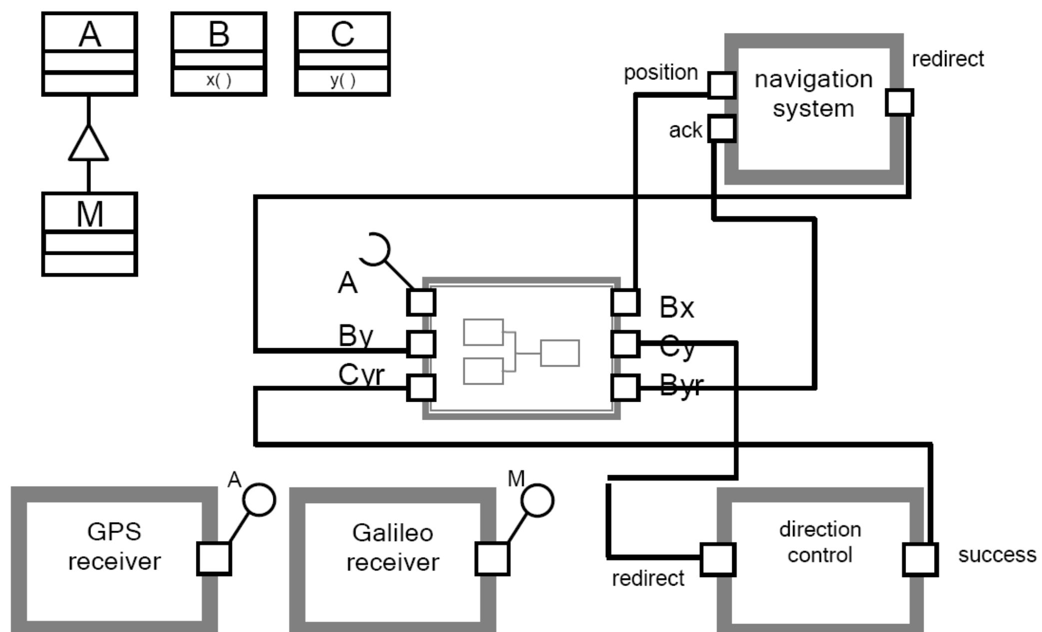
- elgsenos šablono struktūra negali būti pilnai modeliuojama, nes tik žinučių antraštės naudojamos sekų diagramose. Tipų apibrėžimai nėra tiesiogiai modeliuojami, taigi vardiniai potipiai negali būti analizuojami, o struktūrinių potipių analizavimas tampa apribotu;
- sekų diagramos objektų ir aktorių susiejimas pagal (2) prielaidą kliudo dinaminiam polimorfizmui, kuris gali būti reikalingas tam tikruose elgsenos projektavimo šablonuose. Pavyzdžiui, 7 paveikslėlyje objektas *Coordinator* paveldi objektą *Process*, kas reiškia, kad kiekvienas *Coordinator* objektas tuo pat metu yra ir *Process* objektas. Toks sąryšis tarp objektų reikalaujamas Chandra-Toueg algoritmo, nes kiekvieną kartą įvykus sutapimui, skirtingi procesai imasi koordinatoriaus rolės.

Laikantis anksčiau aprašytų prielaidų, pilnas algoritmo realizavimas pagal 7 paveikslėlyje vaizduojamą šabloną būtų neįmanomas, nes tik vienas aktorius būtų pastoviu ryšiu susietas su koordinatoriaus role.

Tam, kad išvengti šių apribojimų, reiktų minėtąsias prielaidas pakeisti taip:

- (2) kiekvienas klasės objektas, kuriam sekų diagramoje yra vaizduojama gyvavimo linija, statiškai arba dinamiškai susiejamas su aktoriumi, kuris yra patalpintas į sudėtinį aktorių, atitinkantį pirmąją prielaidą. Dinaminio susiejimo funkcija privalo laikytis (5) punktu pažymėtos prielaidos;
- (4) klasių diagramos gali būti įtrauktos kaip sudėtinių aktorių atributai. Sekų diagramose esančioms objektų gyvavimo linijoms gali būti priskirtos klasės ar interfeisai, kurie yra sudėtinio aktoriaus, talpinančio šią diagramą, atributai. Taip pat gali būti naudojamos klasės ar interfeisai, esantys rekursiškai aukštesnio lygmens konteineriuose iki pat aukščiausio lygmens. Aktoriaus prievadams, lygiai taip pat kaip ir gyvavimo linijoms sekų diagramose, gali būti priskiriamos klasės arba interfeisai, aprašyti kaip sudėtinių aktorių atributai iki pat aukščiausiojo lygmens. Panašiai kaip UML 2.0 kompozicinės struktūros diagramose (composite structure diagram), įvedimo prievadams šis priskyrimas apibrėžia reikalaujamą sąsajos tipą, tuo tarpu išvedimo prievadams – teikiamą sąsajos tipą.
- (5) dinaminis gyvavimo linijų ir aktorių susiejimas atliekamas tikrinant tipus, t. y. gyvavimo linija bus susieta tik su tuo aktoriumi, kuriam priskirtas tipas (klasė arba interfeisas) yra toks pats kaip ir gyvavimo linijos arba yra potipis.

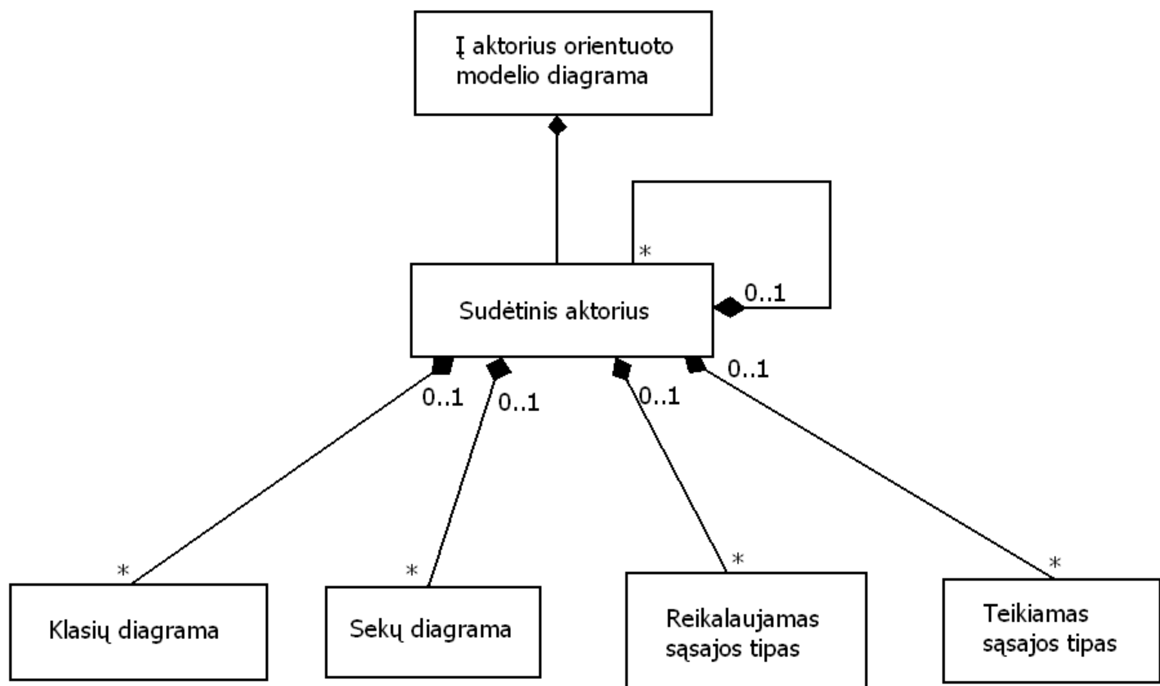
Remiantis šiuo papildytu prielaidų rinkiniu, sistemos aprašas gali būti daromas naudojant trijų tipų UML diagramas – klasių, sekų ir kompozicinės struktūros, kurios bus visiškai integruotos į aktoriais paremtą modelį.



9 pav. Į aktorius orientuotas modelis su integruota sekų diagrama ir išoriniu interfeisų apibrėžimu.

9 paveikslėlyje pavaizduotas naujas sekų diagramos iš 8 paveikslėlio a dalies panaudojimas, tik šį kart pagal papildytą prielaidų rinkinį. Kairiajame viršutiniame kampe matyti klasių diagrama, kuri yra kaip modelio atributas, leidžiantis aiškiai ir tik pagal vardą (nėra hierarchijos lygių) apibrėžti tipus. Pats modelis, kuris vis dar yra į orientuotas į aktorius, atitinką grafinę kompozicinės struktūros diagramos sintaksę su reikalaujamų ir teikiamų sąsajų tipų priskyrimais prievadams. Pavyzdyje matyti statiniai ryšiai tarp aktorių ir sekų diagramos gyvavimo linijų per prievadų sujungimus (lygiai taip pat, kaip 8 paveikslėlyje), bet šiuo atveju tiek „GPS Receiver" tiek „Galileo Receiver" aktoriai gali būti dinamiškai asocijuoti su „A" gyvavimo linija iš sekų diagramos, nes abiejų prievadai atitinka sąsajos reikalavimus („GPS Receiver" teikia sąsajos tipą „A", o „Galileo Receiver" teikia sąsajos tipą „M", kuris yra „A" tipo potipis).

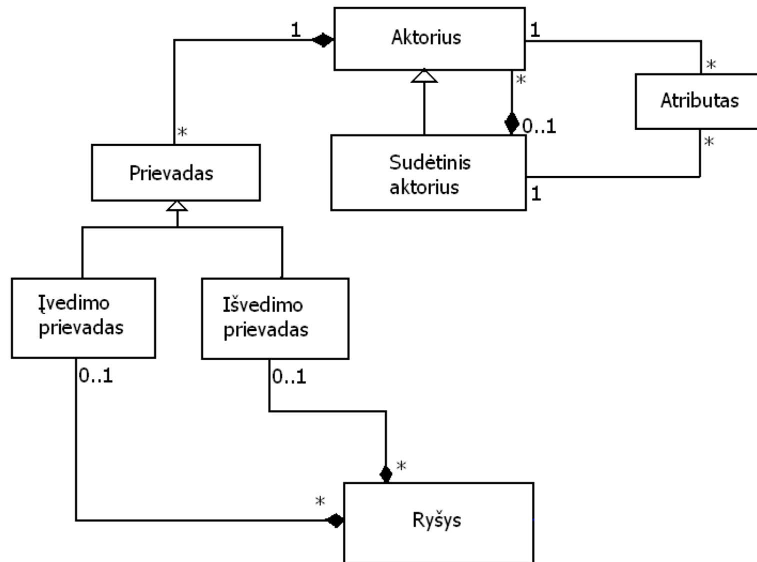
UML elgsenos projektavimo pavyzdžių integravimo metamodelis vaizduojamas 10 paveikslėlyje.



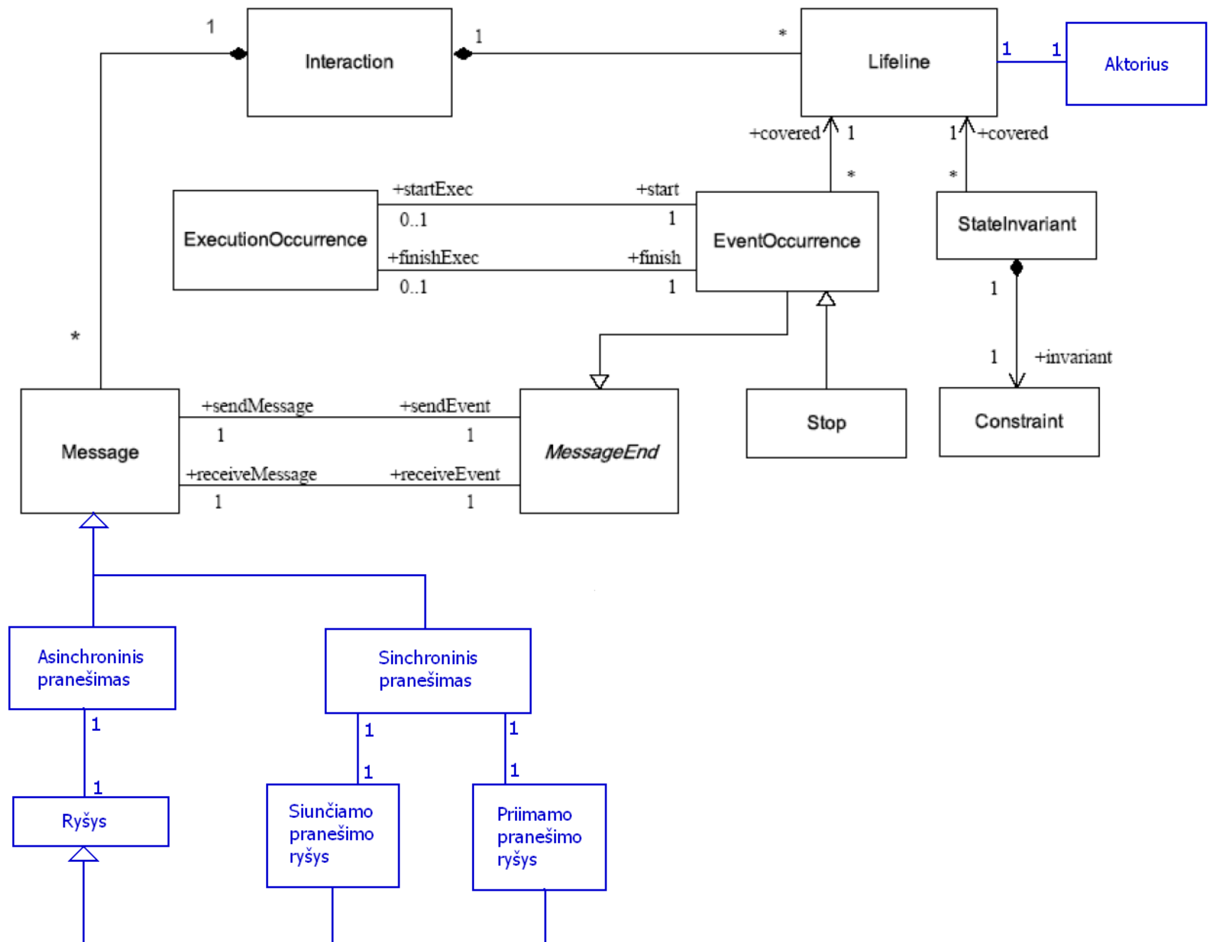
10 pav. Elgsenos projektavimo pavyzdžių integravimo metamodelis

Ptolemy II sistemoje kiekvienas modelis yra sudėtinis aktorius. Pats sudėtinis aktorius viduje gali talpinti kitus sudėtinius aktorius. Elgsenos šabloną realizuojantis aktorius savyje talpina klasių diagramos aktorių, sekų diagramos aktorių, sąsajos tipus nurodančius aktorius. Šiais aktoriais realizuojamas elgsenos šablonas. Taip pat, minėtasis sudėtinis aktorius gali talpinti paprastus, kitaip dar Ptolemy II sistemoje vadinamus atominius, ar sudėtinius aktorius.

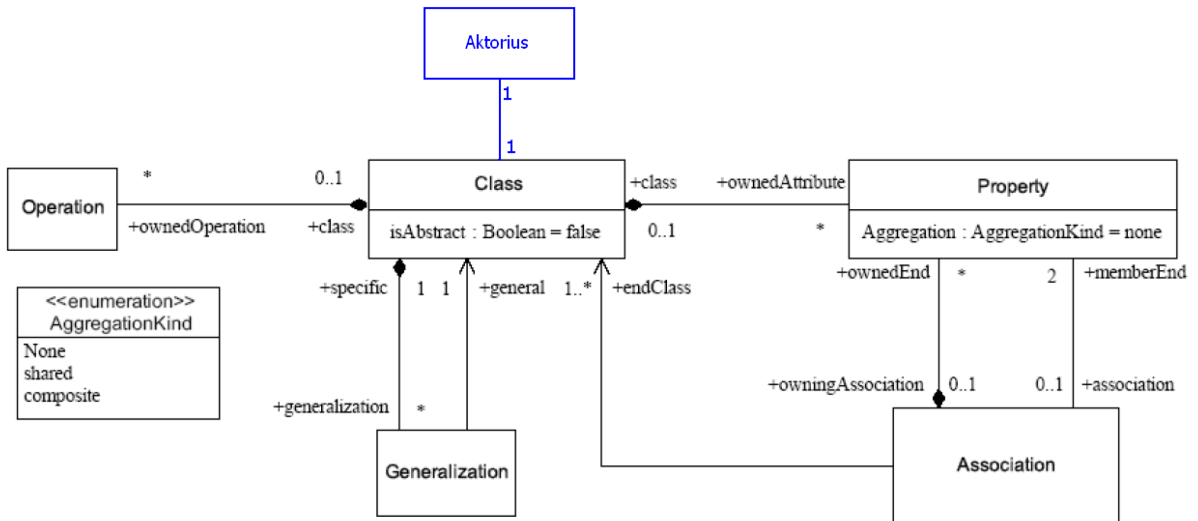
Sekantys trys paveikslėliai vaizduoja sistemos išplėtimo metamodelį. Sekų (12 paveikslėlis) ir klasių (13 paveikslėlis) diagramose mėlyna spalva vaizduojami komponentai, per kuriuos realizuotas sąryšis tarp skirtingų diagramų.



11 pav. Į aktorius orientuotų diagramų metamodelis



12 pav. Sekų diagramų metamodelis



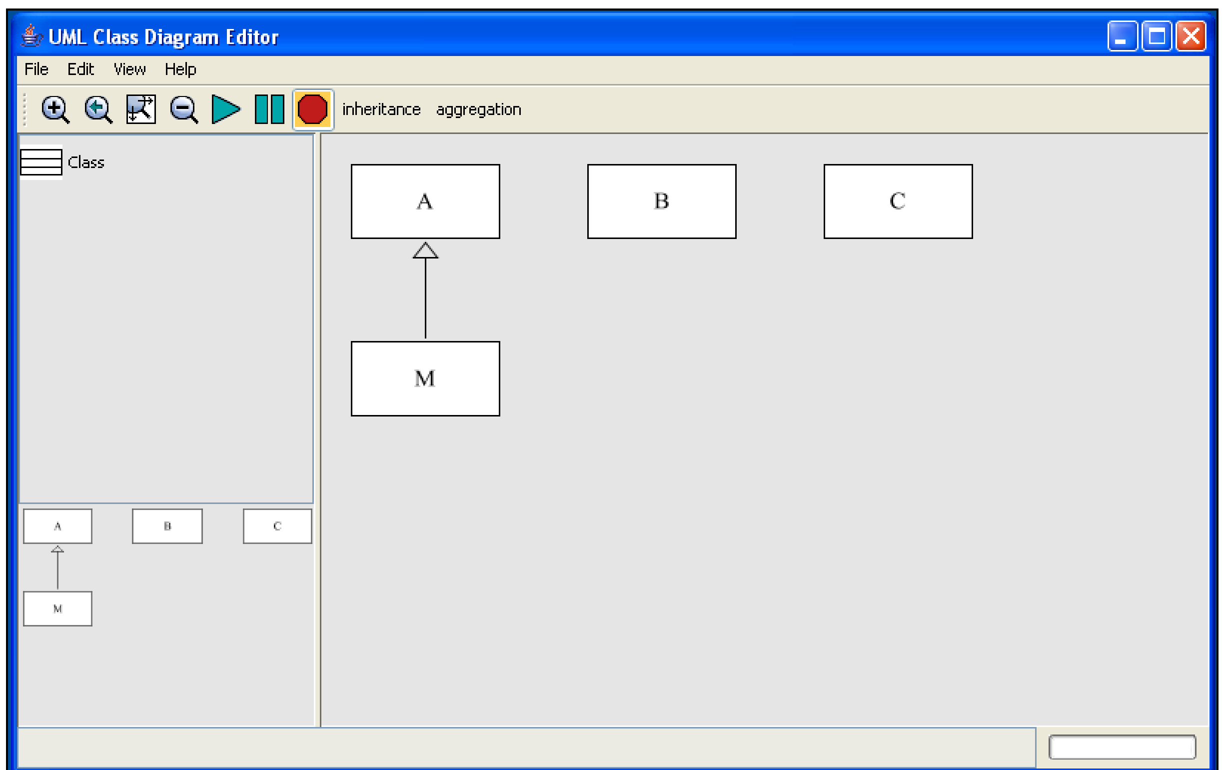
13 pav. Klasių diagramų metamodelis

Sekų diagramos gyvavimo linijos komponentas Lifeline su klasių diagramos komponentu Class susiejamas per aktoriaus komponentą iš į aktorius orientuoto projektavimo metamodelio (11 paveikslas). Ryšių pagalba sekų diagramos Lifeline komponentas susiejamas su Aktoriaus komponentu, o Aktoriaus komponentas naudojant atributus susiejamas su Class komponentu sekų diagramoje.

3.3 Kombinuota į aktorius orientuoto/objektinio projektavimo metodika vienlusčių sistemų projektavimui.

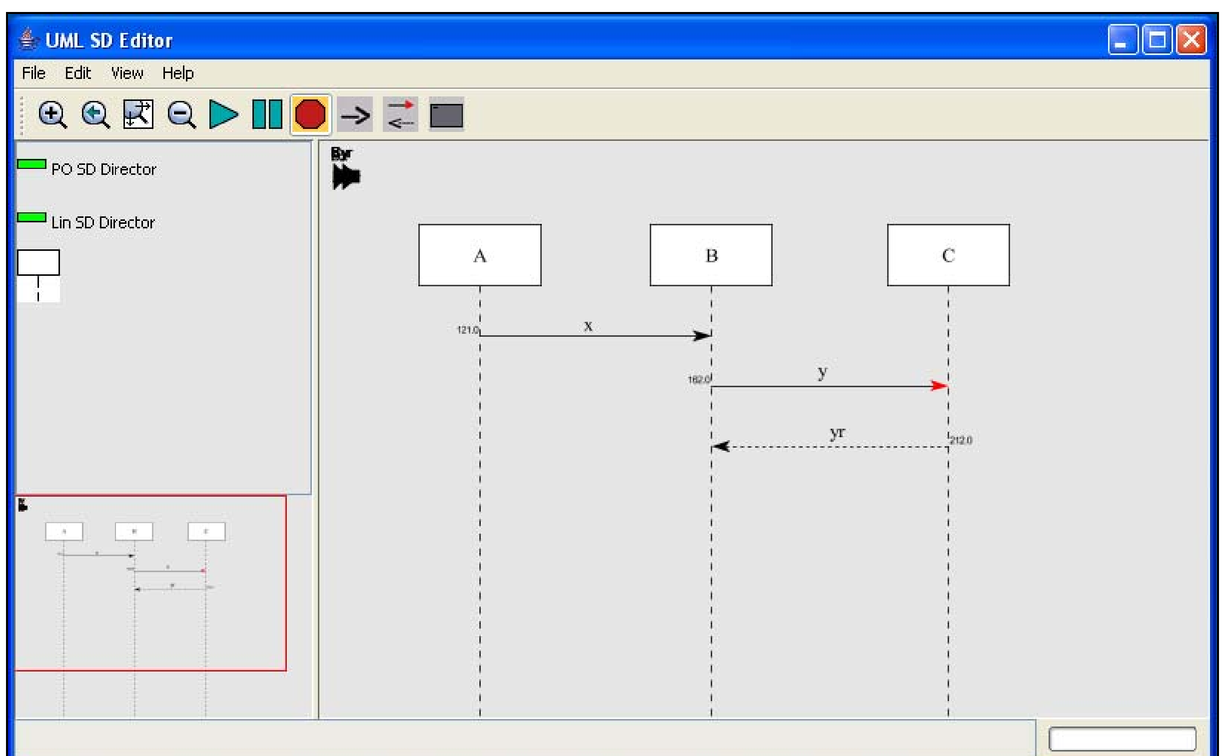
Elgsenos projektavimo pavyzdį išreiškiančios UML diagramos yra talpinamos į tam specialiai realizuotus sudėtinius aktorius. Šie aktorių komponentai į konstruojamą modelį įkeliami iš kairėje esančios aktorių naršyklės (matoma 16 paveikslėlyje). Priklausomai nuo diagramos tipo, komponento vidinės struktūros specifikavimui atidaromi langai turi skirtingas specializuotas vartotojo sąsajas.

14 paveiksle matyti 9 paveikslėlio klasių diagrama realizuota Ptolemy II aplinkoje. Dabartinė Ptolemy II praplėtimo versija neleidžia aprašyti metodų antraščių ir palaiko vienintelį asociacijos ryšį – paveldėjimą. Metodų antraščių aprašymą dalinai pakeičia galimybė pervadinti pranešimus sekų diagramose. Klasių skaičius diagramoje neribojamas kaip ir pačių diagramų projekte.



14 pav. Ptolemy II papildymo klasių diagramų redaktorius

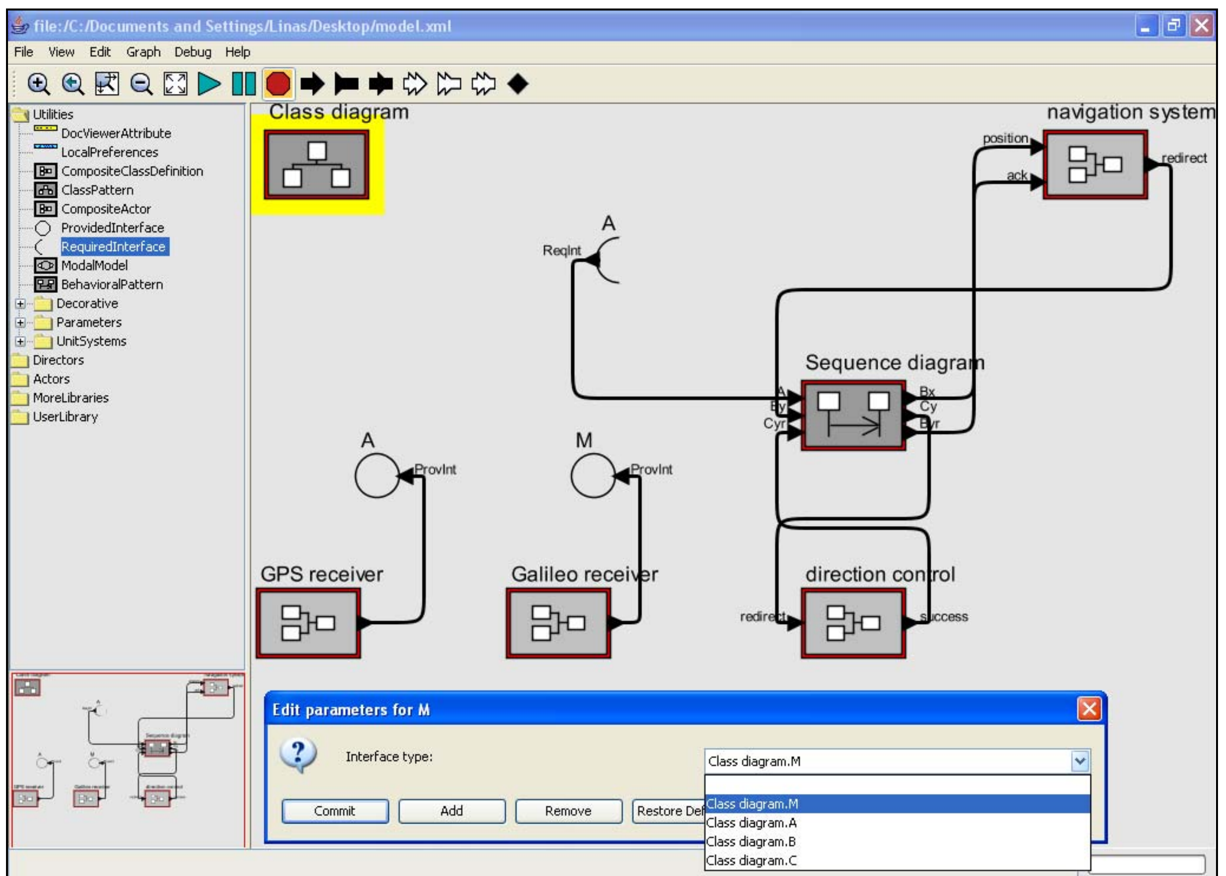
15 paveiksle matyti 8.a paveikslėlio sekų diagrama realizuota Ptolemy II aplinkoje. Kiekvienas pranešimas tarp objektų automatiškai aktoriui sukuria įvedimo ir išvedimo prievadus, matomus redaktoriaus viršutiniame kairiajame kampe. Pranešimai yra paskirstomi laike. Sekų diagramos interpretacija apibrėžiama į aktorių įkeliat *director* objektą.



15 pav. Ptolemy II papildymo sekų diagramų redaktorius

Tiek klasių, tiek sekų diagramų redaktoriai naudoja tik UML sintaksei būdingus elementus, o per specializuotą sąsają yra apribojamas standartinių Ptolemy II aktorių panaudojimas.

16 paveikslėlyje pavaizduotas 9 paveikslo modelis, panaudojant visus realizuotus Ptolemy II praplėtimo komponentus. Reikalaujamų ir teikiamų interfeisų specifikavimui naudojami aktoriai į modelį įtraukiami iš kairėje pusėje esančios aktorių naršyklės. Konkretus interfeiso tipas nurodomas per konfigūracinį meniu iš išsiskleidžiančio sąrašo. Sąraše egzistuoja tik tie interfeiso tipai, kurie apibrėžti klasių diagramoje.



166 pav. 9 paveikslėlyje vaizduojamo modelio realizacija Ptolemy II aplinkoje.

3.4 Siūloma Ptolemy II sistemos išplėtimo architektūra PtUMLemy.

Sistemos išplėtimą turėtų sudaryti keletas paketų, kuriuose būtų atskirai aprašomi naujai realizuoti aktoriai, sritys, vartotojo grafinės aplinkos elementai. Norint įdiegti į standartinę Ptolemy II programos versiją UML įrankių papildymą, reiktų atlikit nemažai konfiguravimo tad būtų pravartu realizuoti diegimo vedlį. Tačiau Ptolemy II projektas yra pastoviai atnaujinamas, o PtUMLemy papildymai yra tik eksperimentinėje stadijoje ir jų struktūra nėra iki galo aiški. Tam, kad būtų paprasčiau programuoti ir reiktų atlikti mažiau

pakeitimų kiekvieną kartą integruojant PtUMLemy į naujausią Ptolemy II versiją, šiuo metu yra sukurtas tik vienas paketas, kuriame aprašytos visos UML papildymų klasės. Paketas įterptas į vartotojo sąsajos bibliotekų paketą *ptolemy.vergil* ir pavadintas *uml*. Šiame pakete (uml) realizuotas sekų diagramų redaktorius. Uml paketas praplėstas *classModel* paketu, kuriame aprašytas klasių diagramų redaktorius.

3.5 Siūlomos metodikos ir Ptolemy II sistemos išplėtimo įvertinimas: pranašumai, trūkumai, tolesnio tobulinimo galimybės.

Pristatytas UML elgsenos projektavimo pavyzdžių integravimo modelis į aktoriais paremtų sistemų aprašus skiriasi nuo panašių metodų, tuo kad tuo pačiu metu yra vykdomi ir aktoriais ir UML šablonais paremti modeliai. Toks integravimo požiūris UML modeliams suteikia realias galimybes būti vartojamiems sisteminio lygmens projektuotojų, nes tereikia įsisavinti tik tas UML ypatybes, kurios tiesiogiai duos naudą jų jau naudojamoje projektavimo metodologijoje. Vietoj to, kad mokytis naują ir pakankamai sudėtingą kalbą nuo pat pradžių, užtenka išsiaiškinti tik keletą UML kalbos ypatybių ir suprasti ją kaip klasikinę blokinę modeliavimo ir simuliavimo aplinką.

Analizuojamas Ptolemy II sistemos išplėtimas vis dar yra kūrimo stadijoje, nes nėra pilnai realizuotas 3.2 skyriuje aprašytas papildytas prielaidų rinkinys. Nėra galimybės dinamiškai susieti aktorius su sekų diagramoje esančiomis gyvavimo linijomis, nes komponente *director* nerealizuotas tipų tikrinimo mechanizmas. Minima savybė yra labai aktuali naudojant elgsenos projektavimo pavyzdžius ad-hoc bevielių tinklų (šablonai suformuojami dinamiškai, kai tam tikro tipo įrenginiai atsiduria viens kito kaimynystėje) arba daugiaprocesorinių sistemų, turinčių laisvai konfigūruojamus procesorius (procesoriai gali dinamiškai perkonfigūruoti duomenų perdavimo kelius pagal tam tikrus šablonus) modeliavime.

4 Eksperimentinė dalis

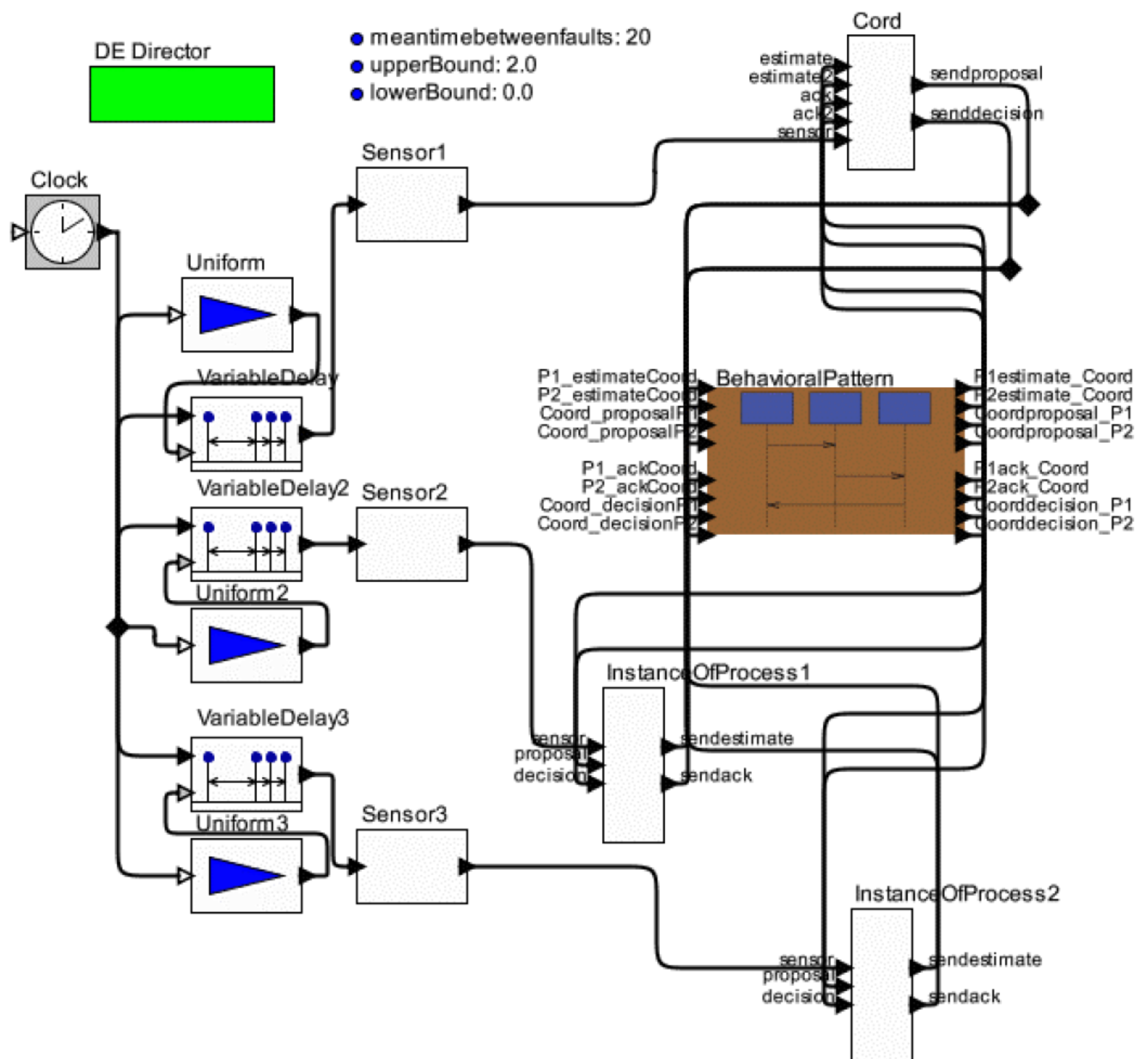
4.1 Testuojamos sistemos aprašas

Ptolemy II sistemos papildymas elgsenos projektavimo pavyzdžių integravimui su į aktorius orientuotais modeliais buvo naudojamas elgsenos šablonui, pavaizduotam 7 paveiksle, modeliuoti. Diagrama buvo patalpinta į sudėtinį aktorių laikantis anksčiau aprašytų

prielaidų. Tam, kad patikrinti šio aktoriaus funkcionavimo teisingumą, naudojant Ptolemy II aktorius buvo modeliuojamas bandymo stendas - aktoriais realizuota paskirstytoji sistema, susidedanti iš trijų sensorių, matuojančių tam tikrą parametrą (pavyzdžiui, infraraudonųjų spindulių intensyvumą kažkokioje vietoje). Šie trys sensoriai nuskaitytų duomenų teisingumo tikrinimui (normaliomis darbo sąlygomis visos trys reikšmės turi būti vienodos) naudoja paskirstyto sutarimo šabloną (distributed consensus pattern). Modeliuojamoj testinėje aplinkoje, kiekvienas sensorius asocijuojamas su procesu, kurių vienas yra koordinatorius. Sensoriai periodiškai siunčia duomenis procesams, o procesai nuskaitytų reikšmių teisingumo tikrinimui naudoja elgsenos šabloną. Realizuotas į aktorius orientuotas sistemos modelis pavaizduotas 17 paveikslėlyje, kuriame dešinėje centre esantis didesnysis aktorius talpina sekų diagramą.

4.2 Testų rezultatai

Modelio vykdymas parodė, kad sekų diagrama atlieka savo funkciją tinkamai, ko pasėkoje koordinatorius teisingai keitėsi žinutėmis su likusiais dviem procesais ir konsensusas sėkmingai buvo priimtas. Tam, kad patikrinti modeliuojamą sistemą realistiškesnėmis sąlygomis, į modelį buvo įtrauktos dviejų rūšių klaidos: sensorių nuskaitymo vėlinimas ir sensorių klaidingas nuskaitymas. Pirmojo tipo klaidos imituoja kintamo dydžio vėlinimo intervalus, kurie gali atsirasti, pavyzdžiui, dėl sensoriaus procesoriaus atliekamo paveiksliukų apdorojimo ar duomenų suspaudimo. Antrojo tipo klaidos imituoja trumpalaikę klaidą, kuri paveikė išmatuotą reikšmę.



17 pav. Sekų diagramos testavimo modelis Ptolemy II sistemoje

Vėlinimas buvo modeliuojamas kaip tolygiai pasiskirstęs kintamasis, tuo tarpu sensorių nuskaitymo klaidos buvo modeliuojamos kaip „poisson“ procesas, charakterizuojamas vidutiniu laiku tarp klaidų (VLTK). 2 lentelėje vaizduojami simuliavimo rezultatai, kiek kartų nepavyko pasiekti konsensuso per 1000 kiekvieno sensoriaus nuskaitymų (sensoriaus nuskaitymas buvo atliekamas per vieną laiko vienetą), esant skirtingoms vėlinimo ir VLTK reikšmėms. Kiekviena reikšmė buvo gauta skaičiuojant dešimties simuliavimų vidurkį.

Vėlinimas	*VLTK*				
	20	***30***	***40***	***50***	***60***
0-0.2	*6.4*	*2.5*	*1.1*	*0.0*	*0.0*
0-2.0	*6.0*	*2.7*	*0.3*	*0.0*	*0.0*

2 lentelė. Simuliavimo rezultatai. Kiek kartų nepavyko pasiekti konsensuso tarp sensorių nuskaitymų.

Iš rezultatų galima teigti, kad sistema yra patikima, kad vidutinis laikas tarp klaidingų sensorių nuskaitymų yra ilgesnis nei 50 laiko vienetų. Šiuo atveju naudojamas konsensuso šablonas priėmė sprendimus visus kartus netgi esant klaidingiems duomenims viename iš sensorių. Taip pat svarbu pastebėti, kad elgsenos šablonas garantavo teisingą procesų bendravimą, kai vėlinimo trukmė siekė intervalą tarp sensorių reikšmių nuskaitymo.

5 Išvados

Sėkmingas objektiškai orientuoto ir į aktorius orientuoto projektavimo apjungimas leistų išnaudoti geriausias abiejų savybes viename modelyje: gausų lygiagretaus komunikavimo elgsenos konstrukcijų rinkinį ir galimybę vykdyti modelius bei hierarchinę vykdymo elgsenos kompoziciją.

Pristatytas UML elgsenos projektavimo pavyzdžių integravimo modelis į aktoriais paremtų sistemų aprašus skiriasi nuo panašių metodų, tuo kad tuo pačiu metu yra vykdomi ir aktoriais ir UML šablonais paremti modeliai. Toks integravimo požiūris UML modeliams suteikia realias galimybes būti vartojamiems sisteminio lygmens projektuotojų, nes tereikia įsisavinti tik tas UML ypatybes, kurios tiesiogiai duos naudą jų jau naudojamoje projektavimo metodologijoje. Vietoj to, kad mokytis naują ir pakankamai sudėtingą kalbą nuo pat pradžių, užtenka išsiaiškinti tik keletą UML kalbos ypatybių ir suprasti ją kaip klasikinę blokinę modeliavimo ir simuliavimo aplinką.

Analizuojamas Ptolemy II sistemos išplėtimas vis dar yra kūrimo stadijoje, nes nėra pilnai realizuotas 3.2 skyriuje aprašytas papildytas prielaidų rinkinys. Nėra galimybės dinamiškai susieti aktorius su sekų diagramoje esančiomis gyvavimo linijomis, nes komponente *director* nerealizuotas tipų tikrinimo mechanizmas. Minima savybė yra labai aktuali naudojant elgsenos projektavimo pavyzdžius ad-hoc bevielių tinklų (šablonai suformuojami dinamiškai, kai tam tikro tipo įrenginiai atsiduria viens kito kaimynystėje) arba daugiaprocesorinių sistemų, turinčių laisvai konfigūruojamus procesorius (procesoriai gali

dinamiškai perkonfigūruoti duomenų perdavimo kelius pagal tam tikrus šablonus) modeliavime.

6 Terminų žodynas

CORBA – (angl. Common Object Request Broker Architecture) OMG (angl. Object Management Group, www.omg.org) grupės apibrėžtas standartas, leidžiantis programuotojui dirbti su objektais, esančiais kitame, nutolusiame kompiuteryje taip, tarsi jie priklausytų tam pačiam procesui, kaip ir iškviečiantis kodas.

HDL – (*angl. Hardware Desctiption Language*) - aparaturos aprašymo kalba.

MARTE – (angl. UML Profile for Modeling and Analysis of Real-Time and Embedded Systems) – UML kalbos papildymas realaus laiko bei įterptinių sistemų modeliavimui ir analizei.

MoML – modeliavimo ženklų schema XML kalba. Skirta aprašyti sujungimams tarp parametrizuotų komponentų.

Poisson procesas – stochastinis procesas, skirtas modeliuoti atsitiktinius įvykius, kurie vyksta nepriklausomai vienas nuo kito.

SystemC – artima C++ kalbai aparatūros aprašymo kalba.

Skaičiavimo modelis (angl. models of computation (MoC)) - tai taisyklių rinkinys, kuris lemia aktorių sąveiką apibrėždamas, kokią įtaką lygiagretumas ir laikas turi aktorių komunikavimui ir elgsenai.

SoC – (*angl. System on Chip*) vienlustės sistemos.

UML - (angl. Unified Modeling Language) - vieninga (unifikuota) modeliavimo kalba – modeliavimo ir specifikacijų kūrimo kalba, skirta specifikuoti, atvaizduoti ir konstruoti objektiškai orientuotų programų dokumentus.

Verilog – viena iš naudojamų aparatūros aprašymo kalbų.

VHDL – viena iš aparatūros aprašymo kalbų (*angl. Very high speed integrated circuit Hardware Description Language*).

XML - (angl. eXtensible Markup Language) yra W3C rekomenduojama bendros paskirties duomenų struktūrų bei jų turinio aprašomoji kalba.

7 Literatūra

[1] R. Damaševičius, V. Štuikys. Application of the Object-Oriented Principles for Hardware and Embedded System Design. INTEGRATION, the VLSI Journal, 2004, 38(2), pp.-309-339. Elsevier.

[2] R. Damaševičius. Transformational design processes based on higher level abstractions in hardware and embedded system design. PhD. Dissertation, Kaunas University of Technology, Kaunas, Lithuania, 2005.

[3] John Davis II, "Order and Containment in Concurrent System Design," Ph.D. thesis, Memorandum UCB/ERL M00/47, Electronics Research Laboratory, University of California, Berkeley, September 8, 2000. (http://ptolemy.eecs.berkeley.edu/publications/papers/00/concsys/)

[4] T. A. Henzinger, B. Horowitz and C. M. Kirsch, "Giotto: A Time-Triggered Language for Embedded Programming," EMSOFT 2001, Tahoe City, CA, Springer-Verlag.

[5] C. A. R. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, Vol. 21, No. 8, August 1978.

[6] G. Karsai, "A Configurable Visual Programming Environment: A Tool for Domain-Specific Programming," *IEEE Computer*: 36-44, March 1995

[7] D. Lea, *Concurrent Programming in JavaTM*, Addison-Wesley, Reading, MA, 1997.

[8] E. A. Lee, "The Problem with Threads," in IEEE Computer, 39(5):33-42, May 2006 as well as an EECS Technical Report, UCB/EECS-2006-1, January 2006. (http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.html)

[9] E. A. Lee and Y. Xiong, "A Behavioral Type System and Its Application in Ptolemy II," *Formal Aspects of Computing Journal*, special issue on Semantic Foundations of Engineering Design Languages, Volume 16, Number 3, August 2004.

[10] R. Milner, *A Theory of Type Polymorphism in Programming*, Journal of Computer and System Sciences 17, pp. 384-375, 1978.

[11] Stephen Neuendorffer, Actor-Oriented Metaprogramming, Ph.D. Thesis, Technical Memorandum No. UCB/ERL M05/1, University of California, Berkeley, December 21, 2004.

[12] S. Neuendorffer, "Automatic Specialization of Actor-Oriented Models in Ptolemy II," Master's Report, Technical Memorandum UCB/ERL M02/41, University of California, Berkeley, CA 94720, December 25, 2002. (http://ptolemy.eecs.berkeley.edu/papers/02/actorSpecialization)

[13] J. Tsay, “A Code Generation Framework for Ptolemy II,” ERL Technical Report UCB/ERL No. M00/25, Dept. EECS, University of California, Berkeley, CA 94720, May 19, 2000. (http:// ptolemy.eecs.berkeley.edu/publications/papers/00/codegen)

[14] J. Tsay, C. Hylands and E. A. Lee, "A Code Generation Framework for Java Component-Based Designs," *CASES '00*, November 17-19, 2000, San Jose, CA.

[15] Y. Xiong and E. A. Lee, “An Extensible Type System for Component-Based Design,” *6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Berlin, Germany, March/April 2000. LNCS 1785.

[16] Y. Xiong, “An Extensible Type System for Component-Based Design,” Ph.D. thesis, Technical Memorandum UCB/ERL M02/13, University of California, Berkeley, CA 94720, May 1, 2002. (http://ptolemy.eecs.berkeley.edu/papers/02/typeSystem).

[17] Y. Zhao, “A Model of Computation with Push and Pull Processing,” Masters Thesis, Technical Memorandum No. UCB/ERL M03/51, University of California, Berkeley, December 16, 2003. (http://ptolemy.eecs.berkeley.edu/papers/03/communicationModeling)

[18] T. D. Chandra, S. Toueg, „Unreliable Failure Detectors for Reliable Distributed Systems,“ Journal of the ACM , 43(2), pp. 225-267.

[18] L. S. Indrusiak, A. Thuy, and M. Glesner, „On the Integration of UML Sequence Diagrams and Actor-Oriented Simulation Models,” presented at DAC Workshop on UML for SoC Design (UML-SOC), Anaheim, USA, 2005.

[19] A. Thuy, L. S. Indrusiak, and M. Glesner, „Applying Communication Patterns to Actor-Oriented Models with UML Sequence Diagrams“, in Proc. Forum on Spec. and Design Languages, 2006.

[20] J.P. van Gigch, System Design Modeling and Metamodeling, Plenum Press, New York, 1991.

[21] J.N. Martin. The Seven Samurai of Systems Engineering: Dealing with the Complexity of 7 Interrelated Systems. Fourteenth Annual Int. Symposium of the International Council On Systems Engineering (INCOSE), 21 – 24 June 2004.

[22] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Reading: Addison-Wesley, 1995.

[23] R. Damaševičius, G. Majauskas, and V. Štuikys, “Application of design patterns for hardware design,” in Proc. 40th Design Automation Conf., 2003, pp. 48-53.

[24] B. P. Douglass. Real-Time Design Patterns: Robust Scalable Arch. for Real-Time Systems. Addison Wesley, 2003.

[25] André De Hon et al, “Design Patterns for Reconfigurable Computing”, in Proc. IEEE Symp. Field-Programmable Custom Computing Machines, 2004, pp. 13-23.

[26] D. Gay, P. Lewis, and D. Culler, “Software design patterns for TinyOS,” in Proc. ACM SIGPLAN/SIGBED Conf. on languages, compilers, and tools for embedded systems (LCTES), 2005, pp. 40-49.

[27] M.F. S. Oliveira, L. Brisolara, F.R. Wagner, L. Carro.“Embedded SW Design Exploration Using UML-basedEstimation Tools,” presented at DAC Workshop on UML for SoC Design (UML-SOC), Anaheim, USA, 2005.

[28] L. S. Indrusiak, M. Glesner, M. E. Kreutz, A. A. Susin, and R. A. L. Reis, “UML-Driven Design Space Delimitation and Exploration: A Case Study on Networks-on-Chip,” in Proc. IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems, 2004, pp. 5-12.

[29] C. Hewitt, “Viewing control structures as patterns of passing messages,” Journal of Artificial Intelligence, 8(3):323–363, June 1977.

[30] J. A. Rowson, A. Sangiovanni-Vicentelli. "Interface-based Design". Proc. of 34th IEEE/ACM Design Automation Conference (DAC), Anaheim, 1997. p. 178-183.

[31] D. D. Gaj ski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao. SpecC: Specification Language and Methodology. Kluwer Academic Publishers, 2000.

[32] SysML Specification v. 0.9 Draft. Available at: http://www.sysml.org

[33] M.F. S. Oliveira, L. Brisolara, F.R. Wagner, L. Carro. Embedded SW Design Exploration Using UMLbased Estimation Tools. DAC Workshop on UML for SoC Design (UML-SOC), Anaheim, 2005.

[34] Hugh C. Lauer and Roger M. Needham. On the duality of operating system structures. In Proc. Second International Symposium on Operating Systems. IRIA, Oct 1978. Reprinted in Operating Systems Review, 13,2 April 1979, pp. 3–19.