УНИВЕРЗИТЕТ У БЕОГРАДУ
ЕЛЕКТРОТЕХНИЧКИ ФАКУЛТЕТ

Стефан И. Тубић

Језик за опис архитектуре софтверских система заснован на функционалној декомпозицији

докторска дисертација

Београд, 2024.

UNIVERSITY OF BELGRADE
SCHOOL OF ELECTRICAL ENGINEERING

Stefan I. Tubić

Language for description of an architecture of software systems based on functional decomposition

Doctoral Dissertation

Belgrade, 2024.

Ментор

др Милош Цветановић, ванредни професор,
Универзитет у Београду, Електротехнички факултет

Чланови комисије

др Захарије Радивојевић, ванредни професор,
Универзитет у Београду, Електротехнички факултет

др Зоран Шеварац, редовни професор,
Универзитет у Београду, Факултет организационих наука

др Горан Квашчев, редовни професор,
Универзитет у Београду, Електротехнички факултет

др Бошко Николић, редовни професор,
Универзитет у Београду, Електротехнички факултет

др Саша Стојановић, ванредни професор,
Универзитет у Београду, Електротехнички факултет

Датум одбране

_____

Захвалница

Породица ми је била највећи ослонац током целог живота. Захваљујем се својој мами Сузи, тати Илији и брату Филипу који су ми увек давали подршку у свему што сам радио и били уз мене. Захваљујем се свим члановима своје породице на подршци и лекцијама које сам од њих добијао везаних за школу и за живот иначе. Хвала девојци Јовани која је са мном прошла кроз многе изазове које су моје студије носиле.

Свом ментору, професору др Милошу Цветановићу, захваљујем се на времену проведеном током истраживања и писања радова. Током рада на факултету и за време студија научио сам доста од њега. Поред професионалног, захваљујем му се на пријатељском и искреном односу који је са мном градио.

Многобројним пријатељима захваљујем се на дружењу још од детињства па до данас. Увек су употпуњавали моје време на леп начин и чинили да дани пролете. Поготову се захваљујем пријатељима који су ту још од мог најранијег детињства.

Хвала професору др Захарију Радивојевићу на времену проведеном током истраживања и писања радова и на знању које ми је пренео. Хвала Тамари у помоћи коју ми је пружала током рада на многобројним заједничким предметима и на пријатељском односу који имамо. Такође, хвала и осталим члановима катедре за Рачунарску технику и информатику на квалитетно и лепо проведеном времену.

Захвалност дугујем учитељима из своје основне школе, наставницима из своје средње школе, наставном особљу факултета као и свима осталима који су ме учили током живота. Верујем да ће ми стечено знање доста значити. Поготову су вредна пријатељства која сам током тог времена стекао.

Наслов докторске дисертације:

Језик за опис архитектуре софтверских система заснован на функционалној декомпозицији

Сажетак:

Описивање архитектура комплексних софтверских система користећи архитектуралне језике обично се врши кроз више гледишта што омогућава креирање погледа. Иако креирање погледа омогућава поделу интереса заинтересованих страна за системом и олакшава управљање, уводи проблем неконзистентности између погледа. Ова дисертација представља Анотирану функционалну декомпозицију (АФД), архитектурални језик који пружа интегрално моделовање као могуће решење проблема неконзистенција. Интегрално моделовање креира модел декомпоновањем система на његове функције, које су анотиране тако да симултано креирају више погледа. Имајући све креиране погледе доступним у исто време у моделу олакшава се управљање конзистенцијом. АФД пружа аутоматску детекцију неконзистентности и мануелну резолуцију неконзистентности. Штавише, АФД пружа аутоматско превођење погледа у одговарајуће UML дијаграме, што олакшава прилагођавање другим методолошким приступима. Према критеријумима који се користе у литератури за евалуацију 124 архитектурална језика, АФД пружа 9 од 12 захтева који су битни практикантима. Додатно, иницијални резултати квантитативне анализе показали су да АФД олакшава детекцију неконзистентности у поређењу са UML. Поред његове употребе у архитектури софтвера, АФД се може користити у образовању и уведен је да помогне студентима у превазилажењу изазова разумевања логике информационих система. АФД користи методолошке концепте из рачунарског размишљања и представља приступ за декомпозицију проблема. Квалитативне и квантитативне евалуације коришћења АФД током курса информационих система показале су да су студенти који су користили АФД постигли више просечне оцене од оних који су користили UML за решавање истих проблема, а штавише да студенти сматрају АФД лаким за разумевање и употребу.

Кључне речи: архитектурални језик, гледишта, конзистенција, интегрално моделовање, UML, рачунарско размишљање, функционална декомпозиција, информациони систем, систем дизајн

Научна област: Електротехника и рачунарство

Ужа научна област: Рачунарска техника и информатика

УДК број: 621.3:004.2

Doctoral dissertation title:

Language for description of an architecture of software systems based on functional decomposition

Abstract:

Describing architectures of complex software systems using architectural languages is usually done through multiple viewpoints that enable the creation of views. While the creation of views enables the separation of stakeholders' concerns with the system and eases manageability, it raises the problem of inconsistencies among the views. This dissertation presents Annotated Functional Decomposition (AFD), an architectural language that provides integral modeling as a possible solution to the problem of inconsistencies. Integral modeling creates a model by decomposing a system into its functions, which are annotated to simultaneously create multiple views. Having all created views available in the model at the same time facilitates inconsistency management. AFD supports automated inconsistency detection and manual inconsistency resolution. Moreover, AFD supports the automated translation of views to appropriate UML diagrams, which facilitates adaptation to other methodological approaches. According to the criteria used in the literature for the evaluation of 124 architectural languages, AFD provides nine out of 12 requirements that are important to practitioners. Additionally, initial results of quantitative evaluation showed that AFD facilitates inconsistency detection compared to the UML. Besides its use in software architecture, AFD can be used in education and is introduced to assist students in overcoming the challenge of understanding the logic of information systems. AFD leverages methodological concepts from computational thinking and represents a problem decomposition approach. Quantitative and qualitative evaluations of AFD usage during an information systems course revealed that students who used AFD achieved higher average grades than those who used UML for solving the same problems, and moreover that students perceived AFD as easy to understand and use.

Key words: architectural language, viewpoints, consistency, integral modeling, UML, computational thinking, functional decomposition, information system, system design

Scientific field: Electrical Engineering and Computing

Scientific subfield: Computer Engineering and Informatics

UDC number: 621.3:004.2

Садржај

Списак слика

Списак табела

1. Увод

Развијање великих софтверских система обично резултује комплексним архитектурама и током времена постала је пракса да се дизајнирају, граде и анализирају њихови описи архитектура са више гледишта [1]. Гледишта омогућавају различитим заинтересованим странама да се фокусирају на детаље у складу са својим интересима. Посматрање система са више гледишта резултује у креирању погледа. Велики скуп погледа може довести до тога да заинтересоване стране заврше са описом архитектуре којим се тешко управља. Штавише, опис додатно захтева да се обезбеди конзистенција између доста различитих погледа и тиме још више отежава управљање. Обезбеђивање конзистенције је комплексан процес који обухвата активности од детекције, преко разрешавања, све до праћења неконзистентности.

Архитектурални језици и њихови алати као средство за описивање архитектура, боре се са комплексношћу софтверских система на различите начине. Утврђивање предности и мана сваког архитектуралног језика може се урадити на основу захтева који су веома важни за практиканте [2]. Захтеви који се односе на дефинисање синтаксе и семантике језика разликују визуелне и текстуалне језике. Слично, начин на који језик омогућава спецификацију, модификовање и одржавање описа архитектура обухвата захтев који се односи на подршку за више гледишта. Додатно, оперативна употреба језика зависи од подршке за алатом која, између осталог, обухвата захтев који се односи на управљање конзистенцијом.

Иако је захтев за подршку за више гледишта веома пожељан јер омогућава раздвајање интереса, чини проблем одржавања погледа конзистентним још изазовнијим. Креирање погледа се обично ради независно на различитим местима, одвојеним дијаграмима или текстуалним описима. Већи број погледа чини дизајнирање и разумевање лакшим али инхерентно отежава проблем детекције и разрешавања неконзистентности између погледа.

Архитектурални језици се такође користе ради образовања студената у области архитектуре софтвера, као и ради припреме студената за решавање логичких проблема везаних за систем. Претходно истраживање је открило да након уводних курсева програмирања, многи студенти имају изазове у декомпозицији проблема, развоју планова, и имплементацији тих планова у програмским језицима како би решили програмерске проблеме [3]. Иако ове потешкоће могу произаћи из недостатка адекватне литературе или праксе за пренос знања, оне такође могу бити резултат употребе превише алата због очекивања тржишта у погледу овладавања професионалним програмским алатима од стране студената [4,5]. Слично томе, аутор је приметио да проблем постаје више очигледан код старијих студената који се углавном фокусирају на то како имплементирати решења, а мање на оно што се мора узети у обзир у оквиру решења. Ово искуство је посебно очигледно јер се од њих очекује да почну да користе професионалне алате током напредних курсева специфичног домена, као што су курсеви везани за информационе системе. Иако студенти користе одређене алате, недостаје им способност да се усредсреде на решавање проблема и схвате шире концепте дизајна информационог система, укључујући суштинске логичке провере и ограничења. Проблем се може интерпретирати као како увести методологије решавања проблема и логичког дизајна у курс који покрива дизајн и имплементацију информационих система. Решавање проблема може помоћи студентима да смање когнитивни јаз између логичког дизајна и имплементације информационог система.

Један могући методолошки приступ решавању проблема је рачунарско размишљање. Рачунарско размишљање описује решавање проблема као трансформацију из непожељног иницијалног стања у пожељено крајње стање превазилажењем баријере. Праћење овог процеса захтева рад на високим нивоима размишљања и расуђивања [6]. Jeannette Wing је дефинисала рачунарско размишљање као менталну активност за формулисање проблема који се може решити рачунарски [7,8]. Другим речима, ово обухвата мисаоне процесе који су укључени у формулацију проблема тако да је решење ефективно изражено онако како би машина могла да ради. Особа може извести рачунарско размишљање коришћењем четири технике, познате као стубови, за решавање проблема. Ова четири стуба се независно примењују и директно су релевантна за формулисање рачунарски одрживих решења. Рачунарско размишљање укључује идентификовање комплексног проблема и његово разбијање на под-проблеме којима је лакше управљати (који се назива стуб декомпозиције). Сваки резултујући под-проблем се онда може анализирати појединачно на већој дубини са намером да се идентификују слични проблеми који су претходно решени (стуб за препознавање образаца). Овај дубински преглед фокусира се само на релевантне детаље, а занемарује ирелевантне информације (стуб апстракције). Коначно, могу се успоставити једноставна правила за решавање сваког под-проблема (стуб алгоритма) [8].

Потенцијална методолошка парадигма дизајна је функционална декомпозиција. Смернице курикулума ACM и IEEE рачунарског друштва препознале су функционалну декомпозицију као препоручену тему у области знања о софтверском инжењерству. У последњем издању смерница курикулума рачунарске науке (CS2013), функционална декомпозиција се помиње као парадигма дизајна, названа структурирани дизајн, који изводи функционалну декомпозицију одозго према доле [9].

Овај рад представља Анотирану функционалну декомпозицију (АФД) која је текстуални архитектурални језик који омогућава креирање описа архитектура комплексних система на основу функционалне декомпозиције као парадигме за методолошки дизајн. Поврх функционалне декомпозиције АФД уводи анотације са циљем подршке за више гледишта и креирања погледа. Погледи у АФД се креирају заједно у истом текстуалном опису, што води ка интегралном моделу који би требало да олакша одређене активности управљања конзистенцијом. Поред овога, ова дисертација представља АФД алат који подржава оперативну употребу АФД не само за креирање интегралног модела већ и за раздвајање интегралног модела у погледе. Штавише, АФД алат омогућава трансформацију описа архитектуре датог у АФД у одговарајуће UML дијаграме.

Поред коришћења у сврхе креирања описа архитектуре комплексних софтверских система и увођења интегралног модела који би требало да олакша управљање конзистенцијом, ова дисертација уводи решење проблема превазилажења јаза између решавања проблема током дизајна информационих система и употребе професионалних алата током имплементације информационих система [10]. Предложено решење је засновано на рачунарском размишљању као приступу решавања проблема и функционалној декомпозицији као парадигми дизајна, тако да је рачунарско размишљање имплементирано у оквиру функционалне декомпозиције. Имплементација четири стуба рачунарског размишљања у оквиру функционалне декомпозиције се постиже кроз нови приступ који је овде уведен у оквиру архитектуралног језика АФД. Анотације у АФД омогућавају употребу декомпозиције, препознавања образаца,

апстракције и алгоритама. АФД алат, који омогућава рад са АФД, је имплементиран као веб апликација коју користе студенти током курса информационих система. Уграђивањем АФД алата у своје студије, студенти дизајнирају информационе системе узимајући у обзир потребне логичке провере и ограничења, што се директно своди на будући рад на дизајну који користи професионалне алате са фокусом на развој. Штавише, увођење рачунарског размишљања на курс софтверског инжењерства може га учинити доступнијим студентима којима то није ужа специјалност [11,12], што би у случају курса информационих система могле бити науке о менаџменту, истраживање операција или други.

Остатак дисертације организован је на следећи начин. Друго поглавље описује архитектуру комплексних софтверских система и уводи потребну терминологију. Треће поглавље описује особине архитектуралних језика и наводи неке од постојећих архитектуралних језика. Четврто поглавље дефинише проблем, предмет, циљ и значај рада и уводи полазне хипотезе. Пето поглавље описује архитектурални језик АФД. Шесто поглавље описује АФД алат. Седмо поглавље представља евалуацију и даје одговоре на уведене полазне хипотезе. Осмо поглавље даје смернице за даља истраживања. Девето поглавље закључује дисертацију.

2. Архитектура комплексних софтверских система

Архитектура је реч која је првобитно коришћена у областима које нису уско повезане са софтвером и софтверским системима. Архитектура се користи у области грађевине као грађевинска архитектура, или поморству као поморска архитектура. Према томе термин је преузет из других области у област софтвера и користи се као архитектура софтверских система.

Софтвер је данас неизоставни део многих уређаја, од сервера који се налазе у дата центрима, преко кућних рачунара, па до мобилних уређаја. Сваки софтверски систем чине његови основни елементи, то како они међусобно интерагују, као и како интерагују са њиховом околином. Елементи и интеракција елемената међусобно и са околином чине архитектуру једног система [13]. Широко прихваћена дефиниција архитектуре софтверских система може се наћи у стандарду ISO/IEC 42010 која гласи: „Архитектура система је скуп фундаменталних концепата или особина система у његовом окружењу, оличених у његовим елементима, односима и принципима његовог дизајна и еволуције.“ [14].

Фундаментални концепти и особине система манифестују се као екстерно видљиво понашање и квалитативне особине [13]. Екстерно видљиво понашање представља шта систем ради са становишта спољашњег посматрача. Систем се тада може посматрати као црна кутија јер није потребно знати како систем ради, већ какав излаз систем даје за одређени улаз. Пример екстерно видљивог понашања је ток података ка систему и од система. Квалитативне особине система представљају одређене нефункционалне карактеристике система које су видљиве споља. У такве карактеристике спадају перформансе система, енергетска потрошња, вероватноћа за кваром, сигурност и остале.

Елементи као фундаментални делови од којих се систем може конструисати и њихови односи могу представљати статичку и динамичку структуру софтверског система [13]. Елементи и односи између њих који представљају статичку структуру дефинишу композицију елемената система и могу бити класе, асоцијације, зависности, наслеђивања између њих, компоненте система, њихова композиција и зависности између њих, или чворови односно физичке машине на које се те компоненте распоређују. Са друге стране, елементи и односи између њих који представљају динамичку структуру дефинишу шта систем ради и могу бити акције, активности и њихова међусобна интеракција, животне линије и размена порука између њих, случајеви коришћења и интеракција спољашњих актера са њима као и остали.

Принципи дизајна и еволуције представљају моћан начин да се дефинише оквир за доношење одлука који ће довести до креирања добро структурираног система који се може одржавати. Имплементација оваквих система је конзистентна и прати дефинисане конвенције структуре на основу донетих принципа. Принципи дизајна могу олакшати креирање система за које постоје конфликтни или чак контрадикторни захтеви.

Софтверски систем се развија како би његовим корисницима био од користи. Међутим скуп категорија људи које имају везе са системом који се развија и користи, доста је шири од самих корисника система из разлога што се софтвер само не користи. Неке од активности које се раде над софтверским системом поред коришћења су његов развој, тестирање, затим над њим је потребно обављати одређене операције попут инсталирања, потребно је поправљати

грешке на њему, унапређивати га и потребно је да га неко плати. Из тог разлога постоји велики број заинтересованих страна за један систем поред самих корисника. Према томе, заинтересована страна у архитектуру система је индивидуа, тим, организација, или њихове класе, које имају интерес у реализацији таквог система [13].

Спецификација архитектуре система је прилика за заинтересоване стране да одреде смер развоја система на основу њихових интереса. Заинтересована страна може представљати особу, али доста често представља групу људи, односни тимове или организације. Генерализација интереса одређене групе људи, иако је практична, може представљати проблем јер интереси појединца могу бити незаступљени интересима групе. Одређени интереси могу бити заједнички различитим заинтересованим странама, али одређени интереси различитих заинтересованих страна могу бити међусобно у конфликту или чак контрадикторни. Разрешавање међусобних конфликата интереса представља изазов за архитекту таквог система.

Креирање система у потпуности се заснива на интересима и потребама заинтересованих страна. Заинтересоване стране заједно са архитектом формирају архитектуру система, а касније га развијају и обављају остале активности над системом. Добро креирана архитектура може бити креирана у случају да су задовољене потребе свих заинтересованих страна. У случају конфликтних интереса архитекта треба да пронађе баланс који је прихватљив.

Како би заинтересоване стране могле да сагледају архитектуру система потребно је прво креирати опис посматране архитектуре. Опис архитектуре је скуп модела који документују архитектуру на такав начин да заинтересоване стране могу да их разумеју, као и да демонстрирају да је архитектура испунила њихове интересе [13]. Опис архитектуре треба да прикаже суштину самог система али и да прикаже довољно детаља који су битни заинтересованим странама и на основу којих се систем касније може развити.

Иако сваки систем има своју архитектуру, нема сваки систем и опис архитектуре и то такав да може бити разумљив заинтересованим странама. Постоји доста техника, метода и модела који могу бити изабрани ради креирања описа архитектуре. Изабир техника, метода и модела је изазов сам по себи и највише зависи од скупа заинтересованих страна како би креирани опис њима био разумљив. У потпоглављу 2.1. се говори о комплексним софтверским системима и њиховим архитектурама. У потпоглављу 2.2. се говори о језицима за опис архитектуре. У потпоглављу 2.3. се описује функционална декомпозиција као потенцијална парадигма за методолошки дизајн.

2.1. Комплексни софтверски системи

Архитектуре комплексних софтверских система састоје се од великог броја елемената различитих типова и њихових међусобних веза. Елементи у комплексним системима могу представљати функције система, информације, имплементацију, софтверске и хардверске компоненте, физичке машине на којима се софтвер извршава, спољашње актере који интерагују са системом као и остале елементе који припадају многим другим аспектима система. Креирање описа комплексних архитектура на такав начин да оне буду разумљиве заинтересованим странама није лак задатак. Потребно је велики број елемената груписати у смислене целине и описати на одговарајући начин, тако да заинтересованим странама системи

буду разумљиви. Комплексне системе треба описати и тако да се њима може касније управљати.

У литератури широко коришћен приступ за описивање архитектура комплексних система води се идејом сагледавања проблема са више различитих гледишта која су уведена IEEE стандардима [15,14]. Гледиште је колекција шаблона, образаца и конвенција за конструисање делова описа архитектуре и дефинише заинтересоване стране чије интересе заступа [13]. Гледишта пружају оквир за прикупљање знања о архитектури које се може користити да усмерава креирање одређених делова описа архитектуре који су разумљиви дефинисаним заинтересованим странама. Конкретна гледишта и њихов број није стандардом одређен већ он зависи од посматране архитектуре и заинтересованих страна које гледишта заступају.

Моделовање комплексних софтверских система кроз више гледишта представља приступ за моделовање који хвали разнолик и комплексан научни корпус знања [16] у домену гледишта. Доприноси остварени у последњих неколико деценија сведоче о значају употребе вишеструких гледишта. 1995. Сони и остали [17] су увели концептуално, модул, извршно и код гледиште. Исте године Kruchtenr и остали [18] увели су четири главна гледишта: логичко, процесно, физичко, развојно и једно опционо гледиште: сценарији. 2000. године IEEE је креирао стандард „IEEE 1471-2000“ сада познат као „ISO/IEC 42010:2007“ [15], који је увео концепт гледишта како би обухватио заједничке описне оквире у многим системима. За разлику од приступа који прописују фиксни скуп гледишта, овај стандард заступа креирање скупа гледишта који најбоље служи заинтересованим странама и њиховим интересима везаним за систем. 2002. године Clements и остали [19] увели су 17 гледишта који су подељени у категорије: стилови модула, стилови компоненти и конектора, стилови алокација, хибридни стилови. 2002. године Garland и остали [20] увели су 14 гледишта који су подељени у категорије: концептуална и аналитичка гледишта, гледишта логичког дизајна и физичка гледишта. 2005. године Rozanski и остали [13] увели су гледишта контекста, функционалности, информације, конкурентности, развоја, распоређивања и операција. 2009. године Taylor и остали [21] увели су логичко, физичко, развојно гледиште, гледиште конкурентности и гледиште понашања. 2011. године IEEE је креирао стандард „ISO/IEC/IEEE 42010:2011(E)” [14] који је ревизија претходног стандарда и који и даље не прописује фиксни скуп гледишта. 2018. године Ozkaya [2] је увео логичко, информационо, физичко, развојно гледиште, гледиште понашања, гледиште конкурентности, гледиште развоја и гледиште операција.

Сагледавајући архитектуру са одређеног гледишта ствара се поглед који описује један део архитектуре система. Према томе, опис архитектуре комплексних система састоји се из више одвојених али међусобно повезаних погледа. На овај начин, погледи врше смислено груписање елемената комплексне архитектуре и њихових веза и тако представљају различите аспекте система. Погледи су као и гледишта дефинисани у оквиру истих IEEE стандарда али историја погледа у односу на гледишта датира још раније из радова које је написао David Parnas 1970их година. Међутим, широко коришћење погледа за описивање комплексних архитектура почело је након што је 1995. године Philippe Kruchten дефинисао одређен скуп погледа, као и сценарије којима се дефинисало њихово понашање. Погледе је дефинисао Rozanski [13] као репрезентацију једног или више аспеката архитектуре, која илуструје како архитектура остварује један или више интереса које заинтересоване стране имају у вези са системом.

Опис архитектуре комплексног софтверског система без коришћења погледа није разумљив заинтересованим странама и не може на задовољавајући начин да прикаже све аспекте система [13]. Креирање описа као скупа погледа омогућава задовољавајућу репрезентацију аспеката, води ка раздвајању интереса и одговорности и разумљив је заинтересованим странама. Међутим, погледи нису независни, већ су међусобно повезани и зависни, па мењање једног погледа може повући и мењање другог погледа. Врло често се дешава да погледи нису међусобно усклађени односно да су неконзистентни. Конзистенција постоји када су елементи који се појављују у једном погледу усклађени са елементима који се појављују у другим погледима или на другим местима у истом погледу [13]. Непостојање конзистенције може увести многобројне проблеме у различите фазе развоја као што су каснији дизајн на нижем нивоу или имплементација.

2.2. Архитектурални језици

Постоји велики број језика за опис архитектуре који се надаље скраћено зову архитектурални језици (енгл. ADL - architecture description language). Архитектуралним језицима се може описати архитектура софтверског система. Такви језици омогућавају креирање модела који заједно чине опис архитектуре. Они долазе из широког спектра језика, од оних који су опште намене и користе за креирање модела који не морају нужно да служе у описивању архитектуре, до језика који су специфични за одређени домен (енгл. DSL – domain specific language).

Један од језика опште намене је UML који се може прилагодити тако да служи одређеним потребама, у овом случају креирању описа. UML пружа нотацију за скуп дијаграма који укључује дијаграм класа, дијаграм компонената, дијаграм распоређивања који описују статичку структуру система, као и дијаграм активности, дијаграм секвенци, дијаграм случајева коришћења, дијаграм стања који описују динамичку структуру система. Поред набројаних и осталих дијаграма које UML пружа, он се може проширити коришћењем стереотипа и профила и тако се може користити у опште намене. С обзиром да је широко коришћен и познат, уколико се користи за описивање архитектуре заинтересоване стране обично неће имати проблема да разумеју његове моделе. Са друге стране језици опште намене укључујући UML нису прављени са намером да буду језици који треба да опишу архитектуру, већ је потребно дефинисати посебну интерпретацију модела или проширити језик како би се користио у те сврхе.

Мана већине језика који служе за опис архитектуре је што они не могу да се изврше а њихови модели нису део имплементације система. За потребе извршавања дефинисаних описа могу се користити језици који су извршни и специфични за одређени домен. Примери таквих језика су CSS, SQL, Groovy. Извршни језици специфични за одређени домен се извршавају у посматраном домену, па је потребно користити некада више различитих језика како би се они користили у сврху описа архитектуре, као и каснијег развоја и тестирања. Са друге стране, описи креирани на овакав начин обично нису лако разумљиви заинтересованим странама, посебно оним који немају знања и додира са изабраним језиком. Због своје специфичности овакви језици са собом носе и многа ограничења у смислу да могу да опишу само одређене делове архитектуре.

Поред језика опште намене и језика који су специфични за одређени домен постоје и језици направљени посебно за описивање архитектуре пружајући нотацију посебне намене чији је основни циљ креирање модела који описују архитектуру. Овакви језици на погодан начин описују велики број аспеката архитектуре тако да њихови модели садрже ниво детаља који је довољан за опис, разумљив заинтересованим странама, а не садрже превише детаља који су прикладнији у фази дизајна и имплементације система. Са друге стране, овакви језици се најчешће користе само у академским круговима, без подршке за алатима, извршавањем и интеграцијом у постојећа окружења која се користе у индустрији. Недостатак подршке доводи до тога да заинтересоване стране обично имају проблем са разумевањем нотације, и коришћењем оваквих језика у пракси. Примери познатијих језика који се користе за опис архитектуре система су Acme са Универзитета Carnegie Mellon и Rapide са Универзитета Stanford [13].

2.3. Функционална декомпозиција

Архитектура софтверског система може бити врло комплексна ствар [13], док креирање описа комплексних архитектура само по себи представља проблем који треба решити. Дакле, неопходно је да архитекте као особе које решавају проблем креирања описа архитектуре, поседују одређено знање, скуп вештина, као и начин размишљања који ће им омогућити да посматрани проблем реше. О решавању комплексних проблема, писала је Wing [12] и увела рачунарско размишљање као методолошки приступ у решавању проблема. Према Wing, рачунарско размишљање представља универзално примењив став и скуп вештина које би сви, а не само особе које се баве рачунарством, требало да науче и користе. Рачунарско размишљање укључује решавање проблема, дизајнирање система, разумевање људског понашања, ослањајући се на концепте фундаменталне за рачунарску науку. Рачунарско размишљање укључује низ менталних активности са циљем решавања проблема.

Решавање проблема описано је као трансформација из нежељеног иницијалног стања у жељено крајње стање превазилажењем баријере. Праћење овог процеса захтева рад на високим нивоима размишљања и расуђивања. Решавање проблема је у литератури обично описано као циклус од седам фаза а то су: препознавање или идентификација проблема, дефиниција и ментална репрезентација проблема, развој стратегија да би се проблем решио, организација знања везано за проблем, алокација менталних и физичких ресурса како би се проблем решио, праћење прогреса према циљу, и евалуација тачности решења [6]. Прва фаза у решавању проблема доста је значајна, јер се проблем мора препознати и идентификовати како би се изабрало најбоље решење. У контексту архитектуре софтвера резултат прве фазе решавања проблема, односно идентификације проблема може бити селекција архитектуралног језика који је пригодан за описивање архитектуре идентификованог система.

С обзиром да је процес решавања проблема доста сличан процесу рачунарског размишљања могу се извидети неке од подударности између решавања проблема и рачунарског размишљања. За потребе извиђања обично се користи теорија и одређени модели [22,23] везани за процесирање информација који између осталог могу симулирати људску обраду информација. Многи истраживачи тврде да рачунарско размишљање мења наш начин размишљања.

Допринос поређења рачунарског размишљања и решавања проблема, Labusch и остали [6] огледају у активностима које су идентификоване применом рачунарског размишљања у решавању проблема. Прва активност обухвата процес идентификације проблема који може бити комплексан и његову декомпозицију на под-проблеме који су мање сложени и самим тим решиви. Друга активност представља процес препознавања већ претходно решених проблема, односно препознавање образаца. Трећа активност односи се на процес апстракције проблема фокусирањем на најбитније детаље и занемаривањем небитних детаља што заједно води ка решењу проблема. Четврта активност представља решавање идентификованих проблема одређеним алгоритмима процесом алгоритамског размишљања.

Идентификација проблема наведена као активност пружена од стране рачунарског размишљања се такође наводи и као прва фаза у решавању проблема. Другим речима, потребно је прво идентификовати проблем и направити основни концепт како би уследило његово решавање. Идентификација проблема као прва фаза решавања проблема, обухвата и процену да ли је проблем рачунарски решив односно да ли се може решити рачунарским размишљањем. Идентификовани проблем обично је комплексан, па се не може лако решити. С тога, потребно је декомпоновати комплексан проблем на више мањих целина односно под-проблема који су лако решиви користећи преостале активности рачунарског размишљања. Процес декомпозиције представља основну активност рачунарског размишљања која води ка решавању комплексних проблема.

У процесу рачунарског размишљања могу се уочити одређене сличности између различитих проблема, док неки могу бити и идентични. У оваквим ситуацијама се процес решавања убрзава јер се већ постојећа решења могу поново употребити. Идентификовање истих образаца између проблема је доста честа појава приликом декомпозиције комплексних проблема. Идентификовање образаца са собом повлачи процесе у решавању проблема као што су обрада информација и коришћење постојећег знања везано за проблеме.

Приликом решавања проблема потребно је фокусирати се на детаље који су битни, а уклонити детаље који нису битни. Овај процес назива се процес апстракције и заснива се на увођењу генералних закључака који се изводе на основу посматрања и искустава [6]. Апстракције, као индуктивни процес, засноване на већем броју посматрања проблема и искустава са њима састојаће се од закључака који су тачнији, односно овакве апстракције ће боље направити разлику између детаља који јесу и који нису битни. Због постојања одређене мере несигурности у процес апстракције, рачунарско размишљање као и процес решавања проблема садржи компоненту евалуације, односно тестирање направљених решења како би се установило да ли су циљеви задовољени.

Поред идентификовања проблема, разбијања проблема на решиве целине, и уклањања небитних детаља процесом апстракције, рачунарско размишљање обухвата и алгоритамско размишљање. Циљ алгоритамског размишљања је решавање идентификованих проблема дефинисањем алгоритама као скупа акција које треба спровести у оквиру решења посматраног проблема. Креирање алгоритама је специфично, јер за разлику од осталих активности рачунарског размишљања, мора бити исказано језицима који нису двосмислени попут говорног језика, већ обично програмским језицима који обезбеђују детерминизам у њиховом раду.

Активности рачунарског размишљања називају се још и технике или стубови, односно: стуб декомпозиције, стуб за препознавање образаца, стуб апстракције и стуб алгоритама [8]. Иако то његов назив не сугерише, овакав начин размишљања не мора бити коришћен само у области рачунарства као на пример ради креирања описа архитектура софтверских система, већ у свим областима где се проблеми могу решити користећи поменута четири стуба. И поред широке примењивости и користи од рачунарског размишљања, у многим основним школама се оно не изучава у довољној мери или на начин на који би требало, али са друге стране расте идеја да коришћење вештина из рачунарства има образовне и економске бенефите [8].

Први стуб рачунарског размишљања, односно стуб декомпозиције доста је заступљен у рачунарској науци. Декомпозиција се у рачунарској науци односи на процес којим се комплексан проблем разбија, односно декомпонује на мање делове које је лакше замислити, разумети, програмирати и одржавати, док се оригинални проблем може поново компоновати од његових делова. У софтверском инжењерству се декомпозиција обично спомиње користећи термин функционална декомпозиција. Реч функција односи се на функције које систем обавља, а сама функционална декомпозиција има дугу историју. Функционалну декомпозицију увео је Dijkstra [24] који је дефинисао процес за разбијање програма на делове, где је сваки део представљен засебном рутином, блок структуром или петљом. Објектно-оријентисана парадигма такође спроводи декомпозицију система одозго према доле тако што систем прогресивно дели на мање целине представљене класама и објектима. Овако описана декомпозиција представља традиционални приступ, где се систем декомпонује одозго према доле на своје саставне делове.

Поред традиционалног приступа декомпозицији Jiang и остали [25] увели су свој приступ којим се декомпозиција врши хоризонтално и вертикално. Хоризонтална декомпозиција представља поделу система у логичке нивое сачињене од функција које имају исту улогу у оквиру система. Вертикална декомпозиција представља поделу система у модуле сачињене од функција које се односе на исти процес у оквиру система. Слојеви дефинисани хоризонталном поделом међусобно су зависни и остварују комуникацију једни са другима како би систем радио у целини. Дакле, хоризонтални слој не може представити посебан процес, и представља логичку поделу система што подсећа на традиционални приступ декомпозицији. Модули дефинисани вертикалном декомпозицијом, међусобно не морају да комуницирају већ могу радити као независне целине и на тај начин представљају посебне процесе у оквиру система. Дакле, модули су целине које се могу независно развијати, тестирати и извршавати.

Декомпозиција је обично мануелан процес, али у одређеним ситуацијама може бити аутоматизована поготово у системима са много корисника где грешке које се праве мануелним процесом могу бити скупе. Стога, развијене су аутоматизоване методе за функционалну декомпозицију, а као један од примера може се навести рачунарство у облаку где је битно да се под-системи могу независно развијати, тестирати и извршавати па се тако заснивају највише на принципима вертикалне декомпозиције [25].

Традиционални приступ декомпозицији не носи са собом идеју поделе система на независне подсистеме или процесе, али има велику примену у анализи непотпуно дефинисаних захтева за системом [25]. Одређени пројекти спроводе функционалну декомпозицију и дефинисање захтева у паралели, док су захтеви за системом, у случају и да постоје и пре декомпозиције система, обично разумљиви само искуснијим инжењерима.

Бенефит функционалне декомпозиције у недовољном познавању захтева за системом, је да су креиране функције обично генералне, не везујући се за имплементацију система, и поново употребљиве касније у декомпозицији.

Brimhall и остали [26] дефинисали су систематизован процес за функционалну декомпозицију у одсуству захтева који треба да задовољи три циља: покривеност, конзистенцију и поновну употребу. Покривеност треба да обезбеди да функције система покривају функционалности које систем треба да има и она укључује обим покривених функционалности, као и дубину, односно да постоје смислене атомске функције. Конзистенција треба да спречи контрадикторности у декомпозицији што укључује контрадикторности у хијерархији функција и да обезбеди компатибилност између структурних и динамичких аспеката система. Поновна употреба треба да обезбеди дефинисање функција без залажења у њихову имплементацију, како би оне касније могле бити поново употребљене, што доприноси и флексибилности у процесу декомпозиције. Предложени систематизован процес описан је акцијама и одлукама које треба спровести како би циљеви били задовољени, а заснива се на посматрању функционалности са више перспектива које чине и атрибуте креираних функција.

Поред потребе да се дефинишу функције, постоји и потреба да се дефинишу нефункционалне карактеристике система, као што су поузданост, перформансе, одрживост и остале, које би се касније могле користити у сврхе анализе. Функције система и њихова декомпозиција могу се описати различитим језицима али они нужно не морају да пруже дефинисање и нефункционалних захтева. Пример једног таквог језика који се дуго користи за декомпозицију система је графички језик за креирање блок дијаграма за ток функција (енгл. functional flow block diagram – FFBD) који на једноставан начин може дефинисати и декомпоновати функције, а може додатно дефинисати ток контроле између њих. Chourey и остали [27] представили су своје решење за представљање нефункционалних захтева на поменутим блок дијаграмима. Како се UML користи доста у архитектури софтверских система и са друге стране постоји потреба за функционалном декомпозицијом, Chourey и остали проширили су семантику UML како би омогућили превођење UML модела у блок дијаграме за ток функција. Додатно, увели су анотације у блок дијаграме за ток функција како би омогућили спецификацију нефункционалних захтева. На крају су приказали превођење анотираних блок дијаграма за ток функција у блок дијаграме на основу којих се може вршити анализа уведених нефункционалних захтева.

Потенцијална парадигма за методолошки дизајн је функционална декомпозиција. Смернице курикулума ACM и IEEE рачунарског друштва препознале су функционалну декомпозицију као препоручену тему у области софтверског инжењерства [10]. У последњем издању смерница курикулума рачунарске науке (CS2013), функционална декомпозиција се помиње као дизајн парадигма, названа структурирани дизајн, која спроводи функционалну декомпозицију одозго према доле (енгл. top-down functional decomposition) [9].

Према до сада наведеним, циљ креирања софтверског система је испуњавање интереса заинтересованих страна. Како би се имао увид у испуњеност интереса потребно је креирати опис архитектуре који је разумљив заинтересованим странама. Креирање описа комплексних софтверских архитектура врши се посматрајући систем са више гледишта и креирањем одговарајућих погледа. Додатно, креирање описа архитектуре представља проблем којем се

може методолошки приступити рачунарским размишљањем и парадигмом функционалне декомпозиције.

3. Постојећи архитектурални језици и њихове особине

За потребе описивања архитектура софтверских система може се користити велики број језика. Ови језици могу бити и говорни језици или то може бити једноставан скуп цртежа састављених од кутија који представљају елементе архитектуре и линија који представљају њихове везе. Природни језици и креирање цртежа имају велику подршку пружену многим текстуалним едиторима и графичким алатима. Међутим, поред велике подршке коју ови језици имају, њихова нотација није направљена са намером да се опишу архитектуре система иако се и за то могу користити. Природни језици и цртежи не могу прецизно описати елементе архитектуре, а њихови описи су готово увек двосмислени. Описи креирани овим језицима не могу лако дефинисати везу између архитектуре система и његове имплементације, па такви описи нису извршиви, не може се генерисати почетни дизајн или програмски код. С обзиром да говорни језици и цртежи нису примењиви у опису архитектура, јавила се потреба за креирањем посебних језика и нотација за описивање архитектура који ће решити наведене проблеме.

Историја архитектуралних језика датира још из 1980их година када су креирани једни од првих архитектуралних језика SARA [28] и Statemate [29]. Велико интересовање у архитектуралне језике почиње током 1990их година када је доста њих и настало. Нотација архитектуралних језика може бити опште намене тако да може да опише архитектуре у различитим доменима, као што је случај код језика Wright [30], Darwin [31], Rapide [32]. Супротно њима, нотације других језика омогућавају описивање архитектура у одређеним доменима, као што је случај код језика AADL [33] у домену уграђених система у реалном времену, π-net ADL [34] у домену система са више агената, или Mobis [35] у домену мобилних система. За разлику од говорног језика, архитектурални језици направљени су са намером креирања модела којим се могу формирати описи архитектура софтверских система и на тај начин имају већу примену у пракси.

С обзиром да се број архитектуралних језика доста повећао од 1990их година поставља се питање како да практиканти изаберу одређени архитектурални језик који одговара њиховим захтевима. У складу са тим у потпоглављу 3.1. су описане особине постојећих архитектуралних језика. У потпоглављу 3.2. су описани неки од постојећих архитектуралних језика за описивање комплексних архитектура софтверских система.

3.1. Особине постојећих архитектуралних језика

Током времена је урађено доста анкета како би се разумеле потребе и захтеви практиканата за архитектуралним језицима. На основу спроведених анкета које су дефинисале захтеве практиканата за језицима Ozkaya је урадио анализу постојећих архитектуралних језика, утврдио њихове предности и мане, и на тај начин покушао да помогне практикантима у изабиру језика, као и да пружи подршку развојним тимовима или појединцима који развијају нове архитектуралне језике у утврђивању предности и мана креираних језика у погледу подршке за постојећим захтевима [2].

Malavolta и остали [36] спровели су анкету над 48 практиканата из 40 компанија у области информационих технологија. У оквиру анкете од практиканата је добијен низ коментара у вези са њиховим очекивањима од архитектуралних језика. Овако дефинисане

захтеве практиканата за архитектуралним језицима Lago и остали [37] су поделили у три групе: дефиниција језика, карактеристике језика и подршка за алатом. С обзиром да су одређени захтеви били доста генерални, како би извршио анализу постојећа 124 језика, Ozkaya је одређене захтеве поделио на под-захтеве.

Дефиниција језика представља синтаксне и семантичке карактеристике језика као што су могућност представљања нефункционалних особина, његова нотација и постојаност формалне семантике. Представљањем нефункционалних особина, као што су перформансе, сигурност, поузданост и остале, омогућава се дефинисање квалитативних особина система. Језици могу специфицирати нефункционалне особине на неформалан или формалан начин. Неформално, нефункционалне особине могу се приказати текстуално или визуелно што зависи од нотације самог језика. Са друге стране, формална спецификација нефункционалних особина касније омогућава анализу система у циљу испитивања испуњености специфицираних нефункционалних захтева. Неке од формалних метода које се користе за анализу су: π-calculus [38], дизајн по уговору [39] и OCL [40]. Постоји 26 језика (21%) који пружају спецификацију нефункционалних особина, 16 језика (13%) неформално као што су ABACUS [41] и Primitive-C [42], а 10 (8%) формално као што су Con Moto [43] и Drems ML [44].

Нотација архитектуралног језика може бити текстуална или визуелна. Текстуална нотација омогућава компактније записане описе архитектура па је погодна за описивање великих и комплексних архитектура, а обично се користи од стране експерата ради бржег моделовања. Визуелна нотација омогућава креирање већег броја дијаграма који графички описују архитектуру и погодна је у ситуацијама у којима је циљ ефикасна комуникација решења, а обично се користи од стране практиканата који немају велико искуство у описивању архитектура. Обе нотације заступљене су у архитектуралним језицима, а њихова употреба зависи од искустава и потреба самих практиканата. Текстуална и визуелна нотација код језика који их пружају увек су међусобно зависне и статички аспекти система обично се описују визуелно, а динамички аспекти текстуално, што није погодно за практиканте који желе да направе избор у нотацији [2]. Постоји 49 језика (39%) који пружају само текстуалну нотацију, 33 (27%) само визуелну, а 42 (34%) и текстуалну и визуелну као што су новији језици Primitive-C [42], DPD [45], C3 [46].

Семантика језика може се дефинисати неформално или формално. Неформално дефинисана семантика се обично дефинише говорним језиком и може увести одређене двосмислености у њено тумачење. Формална семантика дефинише се математички базираним формалним методама у формама аксиома, денотација или операција. Формалне семантике неких језика дефинисане су језицима за формалну спецификацију као што су π-calculus [38], Петријеве мреже [47] и остали. Формално дефинисани језици омогућавају касније анализе софтверске архитектуре које су засноване на математичким доказима, иако се показује да их практиканти ретко користе [2]. Нешто мање од половине архитектуралних језика, тачније 60 језика, су формално дефинисани, 12 језика (10%) користи π-calculus за дефинисање формалне семантике а неки од њих релативно новијег датума су ACDL [48] и ScudADL [49].

Карактеристике језика односе се на описивање архитектуре и управљање описом а представљају га захтеви за више гледишта, проширивост и подешавање, и програмски оквир. Више гледишта баве се управљањем комплексних софтверских архитектура посматрајући их

са више гледишта како би се архитектура описала одговарајућим погледима. Користећи постојеће радове везане за гледишта и погледе, Ozkaya је дефинисао свој скуп гледишта за које је веровао да их практиканти често користе и то су гледишта: Логичко, Информације, Физичко, Распоређивање, Понашање, Конкурентност, Развој и Операције [2]. Логичко гледиште описује логичке компоненте и конекторе који заједно чине софтверски систем. Гледиште Информације описује како компоненте складиште, мењају и размењују њихове податке једне са другима. Физичко гледиште описује хардверске компоненте које су потребне ради спецификације уређаја у које ће софтверске компоненте бити лоциране и односа између тих уређаја. Гледиште Распоређивање описује мапирање између хардверских и логичких компоненти. Гледиште Понашање описује понашања системских компоненти и сложене механизме интеракције. Гледиште Конкурентност описује проблеме које се односе на конкурентност као што су нити и синхронизација између компоненти. Гледиште Развој описује како ће компоненте и конектори бити имплементирани и тестирани. Гледиште Операције описује како ће специфицирани систем бити покретан у свом окружењу, укључујући и планове за инсталацију система, конфигурацију, подршку за кориснике, надгледање система као и планове за креирање резервних копија и враћања система у претходно стање у случају квара. Свако дефинисано гледиште представља посебан под-захтев захтева за више гледишта. Постоји 113 језика (91%) који пружа гледиште Логичко. Гледишта Логичко и Информације пружа 96 језика (78%). Гледишта Логичко, Информације и Понашање пружа 59 језика (46%), када се на то дода гледиште Конкурентност 58 језика 45%, а када се на то дода и гледиште Развој 32 језика (26%). Гледишта Физичко и Распоређивање пружа 19 језика (15%). Сва гледишта пружа само 3 језика (2%): AADL [33], Palladio [50], Ambient-Prisma [51].

Проширивост и подешавање односи се на могућност проширивања језика како би он задовољио неке од нових захтева. Примери проширења су додавање нових гледишта, додавање могућности за аутоматску анализу и нефункционалних гледишта, проширивање подршке алата могућностима као што је генерисање кода, симулације и остало [2]. Може се радити синтаксно и семантичко проширивање. Синтаксно проширивање односи се на проширивање синтаксе језика увођењем нових архитектуралних елемената без мењања семантике језика, или увођењем нових синтаксних правила за већ постојеће архитектуралне елементе. Семантичко проширивање односи се на увођење нових гледишта, подршку за нефункционалним особинама и остале. За сврхе проширивања користе се различите технике а неке од њих који постојећи језици користе су наслеђивање језика, анотација особина, XML, аспект-оријентисан развој и остале. Постоји 20 језика (16%) који пружају проширивост и подешавање а неки од новијих су XAcme [52] и MontiArc [53].

Програмски оквир односи се на подршку за окружење за моделовање које се може користити за креирање описа архитектуре и његових погледа и управљање креираним описима. Програмски оквир такође може да омогући синтаксне, семантичке и друге провере као и генерисање софтверског кода за потребе имплементације и развоја [2]. Нешто више од половине архитектуралних језика пружа окружење за моделовање које омогућава креирање описа уз додатне синтаксне и семантичке провере. Аутоматизовано генерисање кода из описа архитектуре пружа око 45 језика 36%, док су најпопуларнији програмски језици чији код генеришу оваква окружења Java и C. Неки од новијих језика који пружају програмски оквир су AO-ADL [54] и MontiArc [53], где друго поменути пружа додатно и генерисање програмског кода.

Подршка за алатом односи се на особине алата које могу помоћи у опису архитектура и управљању описима. Подршка за алатом састоји се од захтева за аутоматску анализу, управљање великим погледима, подршку за колаборацију, подршку за верзионисање, подршку за управљање знањем и подршку за дизајн оријентисан на архитектуру софтвера. Аутоматска анализа односи се на аутоматске провере ради испуњења циљева комплетности, конзистенције, коректности и компатибилности, које је Ozkaya уједно и дефинисао као под-захтеве за архитектуралним језиком [2]. Аутоматска анализа односи се додатно и на и симулације, проверавање модела, анализе кориснички дефинисаних захтева и остале. Анализа комплетности се односи на проверу да ли опис архитектуре покрива све захтеве дефинисане за системом, као и да ли је ниво детаља описивања коректан. Анализа конзистенције бави се провером правила конзистенције која могу бити дефинисана између архитектуралних елемената. Провера конзистенције је значајна у описивању архитектура комплексних система и често се користи у ситуацијама када се описи архитектура састоје из више погледа. Примери конзистенције би били да је порт конектора компоненте повезан на захтевани порт конектора друге компоненте, да је захтевано понашање компоненте исто као пружено понашање компоненте, или да статичка структура система описана класама одговара динамичкој структури система описаној дијаграмом секвенце. Анализа коректности односи се на провере да ли су захтеване особине система задовољене описом архитектуре. Пример захтеване особине система може бити постојаност или учесталост застоја у систему. Анализа компатибилности се односи на проверу да ли опис архитектуре одговара одређеном стилу за описивање архитектуре или одређеним смерницама за дизајн. Од постојећих архитектуралних језика ниједан не подржава све под-захтеве односно циљеве аутоматске анализе. Анализа конзистенције је подржана од стране 24 архитектуралних језика (19%), док су анализа комплетности, коректности и компатибилности засебно пружене од стране 7 до 11 језика (6 до 9%). Дакле, анализа конзистенције је један од водећих циљева у оквиру аутоматске анализе архитектуралних језика. Од постојећих језика само језици XCD [55] и Stratus ML [56] пружају три од четири циља аутоматске анализе и то анализу комплетности, конзистенције и коректности, а уз то ови језици су и релативно новијег датума [2].

Поред четири циља или под-захтева аутоматске анализе одређени архитектурални језици додатно пружају још неке могућности у оквиру аутоматске анализе а то су симулација, провера модела, провера застоја, претрага досега, анализа кориснички дефинисаних особина система. Симулација омогућава анализу понашања система под одређеним околностима и може дати увид у интеракцију архитектуралних елемената. Провера модела бави се представљањем описа архитектуре, обично превођењем, у одговарајуће математичке моделе који се могу исцрпно проверавати. За разлику од симулације која пружа проверу понашања система кроз ограничен број путања у оквиру простора који дефинишу стања система и њихове транзиције, провера модела омогућава проверу свих путања овог простора. Провера застоја у систему односи се на утврђивање и анализу постојања ситуација у којима одређени део система чека на одговор од другог дела система који се неће никада догодити. Анализа досега односи се на претрагу мртвих стања, односно оних у које систем никада неће ући. Последње, могу се увести одређене под-нотације којима се практикантима омогућава да дефинишу особине система које ће се касније аутоматски проверавати. Симулацију пружа 25 језика (20%) и ти језици обично пружају и проверу модела. Провера модела је најучесталија додатна могућност анализе коју пружа 35 језика (28%). Проверу застоја пружа 28 (23%) језика и сви они то раде

кроз проверу модела. Претрага досега најмање је заступљена и пружена је од стране 16 (13%) језика. Последње, анализу кориснички дефинисаних особина система пружа 24 (19%) језика [2]. Неки од језика релативно новијег датума који пружају већи број могућности за додатну анализу су Con Moto [43] и East-ADL [57] који пружају све набројане додатне могућности осим анализе кориснички дефинисаних особина система.

Подршка за управљање великим погледом односи се на технике представљања великог броја елемената и веза великих и комплексних система посебним погледом на разумљив начин. Управљање великим погледима може се омогућити користећи одређену технику. Технике које се користе за ове сврхе су: структурна композиција компонената, структурна композиција конектора, наслеђивање, композиција понашања и аспектно-оријентисана спецификација. Структурна композиција компонената дефинише компоненте и њихову композицију у погледу под-компоненти и њихових интеракција. Структурна композиција конектора дефинише комплексне протоколе интеракције кроз једноставне механизме интеракције. Наслеђивање позајмљује идеју објектно-оријентисаног програмирања и омогућава компоненти да наследи и касније прошири, уколико то језик дозволи, структуру и понашање друге компоненте. Композиција понашања представља спецификацију понашања компонената кроз понашања других компоненти. Аспектно-оријентисана спецификација омогућава дефинисање аспеката који описују архитектурална решења за комплексне и опште заступљене проблеме софтверских система као што су сигурност, нефункционалне особине и остале. Најзаступљенија техника за представљање великог погледа је структурна композиција компонената коју пружа 70 (56%) архитектуралних језика [2], а неки од њих нешто новији су MontiArc [53] и LISA [58].

Подршка за колаборацију односи се на заједнички рад више заинтересованих страна у креирању описа архитектуре са циљем да се смањи потребно време за креирање описа, као и да се повећа квалитет описа архитектуре. Колаборација може бити синхрона или асинхрона и у оба случаја допушта заинтересованим странама да буду на различитим физичким локацијама. Синхрона и асинхрона колаборација представљају под-захтеве за језиком. Синхрона колаборација захтева да различите заинтересоване стране раде на истом опису архитектуре у исто време, док је асинхрона колаборација релаксиранија у смислу да допушта заинтересованим странама да заједнички раде на истом опису архитектуре у различито време. Постоји 10 језика (8%) који пружају могућност колаборације. Сви они пружају асинхрону колаборацију обично користећи репозиторијуме на које постављају описе архитектура, али само језици AADL [33] и UML [59] пружају додатно и синхрону колаборацију кроз одређене алате [2].

Подршка за верзионисање односи се на могућност чувања детаља везаних за различите верзије описа архитектура или неких његових делова. Верзионисање се обично користи заједно са репозиторијумима тако да се верзије одређених делова описа архитектуре чувају на репозиторијуму, па им се касније одатле може поново приступити. Постоји 18 језика (15%) који пружају верзионисање, а неки новији језици су Drems ML [44] и X-MAN [60].

Подршка за управљање знањем представља материјале везане за архитектуралне језике, и начине дистрибуције поменутих материјала заинтересованим странама. Материјални могу бити књиге, радови, алати, упутстава за коришћење, а њихова дистрибуција обично се ради путем веб сајтова језика. Постоји 29 језика (23%) који имају свој веб сајт, а материјали који су

код готово свих заступљени су увод, радови, алат и контакт информације. На веб сајтовима најмање је заступљено постојање форума, мејл листа и чланстава [2].

Подршка за дизајн оријентисан на архитектуру софтвера бави се алатима за интеграцију процеса креирања описа и самих описа са осталим фазама развоја софтвера као што су спецификација захтева и њихова анализа, дизајн на ниском нивоу и имплементација софтвера. Коришћење описаних архитектура у осталим фазама може се радити аутоматском анализом описа архитектуре, аутоматским генерисањем софтверског кода, поновним коришћењем описа архитектура кроз репозиторијуме [2]. Одређени језици пружају могућност дефинисања функционалних и нефункционалних захтева и каснију аутоматску анализу описа према дефинисаним захтевима. Само језици Palladio [50] и Drems ML [44] омогућавају сву наведену подршку за дизајн оријентисан на архитектуру софтвера.

Захтеви и под-захтеви за архитектуралним језицима ће овде бити заједно набројани, редом пролазећи кроз сваку групу захтева. Захтеви које су практиканти дефинисали су подељени у групе: дефиниција језика, карактеристике језика и подршка за алатом. У оквиру групе дефиниција језика налазе се захтеви за нотацијом, специфицирањем нефункционалних особина и формална семантика, док је захтев за нотацијом рашчлањен на под-захтеве за текстуалном и визуелном нотацијом. Група карактеристике језика обухвата захтеве за више гледишта, проширивост и подешавање, и оквир за програмирање. Захтев за више гледишта рашчлањен је на под-захтеве за гледиштима: логичко, информације, физичко, распоређивање, понашање, конкурентност, развој и операције. Захтев за проширивост и подешавање рашчлањен је на под-захтеве за синтаксно и семантичко проширивање. Захтев за оквир за програмирање рашчлањен је на под-захтеве за едитор за моделовање, и за генератор софтверског кода који обухвата и генерисање самог дизајна. Захтеви у оквиру групе подршка алата су аутоматизована анализа, верзионисање, колаборација, управљање знањем, дизајн оријентисан на архитектуру софтвера и управљање великим погледом. Захтев аутоматизована анализа подељен је на под-захтеве конзистенција, комплетност, коректност и компатибилност. Захтев колаборација подељен је на под-захтеве за синхрону и асинхрону колаборацију. Захтеви и под-захтеви заједно чине оквир који се може користити у развоју будућих архитектуралних језика са циљем да се задовоље потребе практиканата, односно заинтересованих страна и тако повећа употребљивост развијених архитектуралних језика. Додатно, анализа постојећих архитектуралних језика кроз дефинисан оквир [2] пружа добар увид у њихове особине и могућности, и тако помаже у њиховом избору.

Испуњење захтева за више гледишта омогућава поделу интереса и брига везано за систем између заинтересованих страна. Развој заснован на моделима истиче да је подела брига везано за систем приликом моделовања врло битна како би се управљало комплексним софтверским системима [61]. Дакле, задовољење захтева за више гледишта од стране архитектуралних језика посебно је битно у описивању комплексних софтверских архитектура.

Подела брига и моделовање са више гледишта су добро истражене теме, и постоји најмање сто студија у области моделовања кроз више гледишта. Antonio и остали [16] селектовали су међу 8600 студија, 40 најрелевантнијих студија у области моделовања софтвера са више гледишта и дефинисали таксономију којом су описали постојећа решења у овој области. Постојећа решења описана су механизмима за моделовање са више гледишта,

артефактима за моделовање са више гледишта, преписка, управљањем конзистенцијом и подршком алата.

Први механизам моделовања са више гледишта је ортографски којим погледи представљају пројекције јединственог модела који има све информације везане за систем. Механизмом синтезе сваки поглед је представљен засебним мета-моделом, а опис се добија синтезом информација из постојећих погледа. Механизам одвојених погледа карактеришу погледи који су такође представљени засебним мета-моделима, али су погледи одвојени једни од других у смислу да они представљају своје описе система и не може се урадити њихова синтеза. Механизмом пројекције погледи су виртуелни у смислу да их дефинишу скупови концепата, селектованих из једног заједничког мета-модела, који се односе на одговарајућа гледишта. Хибридни механизам представља компромис између механизма синтезе и пројекције којим се омогућава дефинисање погледа над заједничким мета-моделом, а сваки поглед представљен је делом заједничког мета-модела. Највише приступа за моделовање кроз више гледишта подржава механизам синтезе (14 од 40) и механизам одвојених погледа (12 од 40), док остали нису у толикој мери заступљени.

Артефакти за моделовање могу бити одређеног типа. Типови артефаката идентификовани у литератури су гледиште, поглед, перспектива, модел и брига односно интерес. Поред гледишта, погледа, модела и интереса који су дефинисани и спомињани до сада, само један од 40 радова уводи перспективе и то као виртуелне погледе који представљају агрегацију већ постојећих погледа. Иако се ови типови артефаката често користе у литератури они немају у потпуности усклађене дефиниције. Од анализираних приступа, 21 користи погледе, а 18 користи гледишта као тип артефаката, што указује на њихову велику заступљеност. Број артефаката који се користе у већини приступа је промењив, где 25 њих подржава промењив, а 15 фиксан скуп артефаката. Од постојећих приступа 7 подржава могућност креирања нових артефаката. Може се омогућити и прилагођавање артефаката потребама архитектуре која се описује. Пример тога је да се могу додати или уклонити погледи који ће чинити опис. Додатно, постоји проблем адаптације механизама за очување конзистенције и механизма синтезе на овакве промене. Четири приступа нису специфицирала да ли подржавају прилагођавање артефаката, док 21 приступ подржава прилагођавање [16].

Артефакти се дефинишу одређеним језицима користећи своју нотацију. Неке нотације базиране су на UML језику, где се артефакти као што су погледи могу дефинисати UML дијаграмима. Нотације базиране на мета-моделима могу дефинисати артефакте уколико се претходно дефинише мета-модел. Постоје и нотације које користе формалне језике за дефинисање артефаката. Артефакти се дефинишу нотацијама базираним на UML језику у 16 приступа, нотацијама базираним на метамоделима у 15 приступа, док се остали типови нотација користе у мањој мери [16]. Језици су дефинисани и својом конкретном синтаксом која може бити текстуална, визуелна, док неки језици користе комбинују текста и дијаграма. Приступа који подржавају визуелну синтаксу, 33 од 40, има више него оних који подржавају текстуалну.

Постоје приступи који подржавају синтезу артефаката мануелно или аутоматски. Мануелна синтеза се делимично обавља ручно, у одређеним ситуацијама када треба решити одређене неконзистентности или када треба разрешити одрешене комплексне случајеве. Аутоматска синтеза обавља се од стране алгоритама који самостално врше синтезу

информација које долазе из различитих артефаката. Аутоматска синтеза може бити оперативна, она која решава ограничења, или аспектно-дефинисана. Оперативна омогућава синтезу алгоритмима који се базирају на преписками односно преклапањима између артефаката. Други механизам аутоматске синтезе се базира се на решавању проблема ограничења која постоје над артефактима. Аспектно-дефинисана аутоматска анализа заснива се на операцијама које дефинишу аспекти. Од анализираних приступа 12 подржава синтезу, од чега 10 аутоматску, док 23 приступа не подржава синтезу [16].

Преклапања односно преписке између артефаката могу бити релационе, моделоване, имплицитне, или ограничења. Релационе преписке су преписке између артефаката дефинисане релацијама које морају постојати између одговарајућих елемената. Моделоване преписке су дефинисане посебним артефактима наменски направљеним за те сврхе и то обично у посебним језицима. Имплицитне преписке се посебно не дефинишу, већ се успостављају на основу конвенција као што су именовања, идентификатори и остало. Као ограничења, преписке се могу дефинисати на основу правила над гледиштима што ће заузврат дефинисати преписке над одговарајућим елементима. Већина приступа, тачније 31 приступ, подржава експлицитне преписке, 7 имплицитне, док се за два приступа није могло утврдити [16]. Број приступа који подржавају релационе, моделоване или преписке као ограничења је поприлично подједнак.

Експлицитне преписке се описују језицима различитих нотација. Ове нотације могу бити базиране на UML језику, специфицирајући преписке UML дијаграмима и елементима. Нотације базиране на трансформационим језицима дефинишу преписке кроз трансформациона језичка правила, као што могу бити правила између различитих гледишта. Нотације базиране на мета-моделима користе се обично да кроз креирани мета-модел дефинишу преписке. Нотације засноване на формалној спецификацији користе формалне приступе да представе преписке као моделе, релације или ограничења. Око половина приступа, 18 од 40, не пружа информацију о нотацијама које се користе за спецификацију преписки. Код осталих приступа највише су заступљене нотације засноване на UML језику и нотације засноване на мета-моделима. Синтакса ових језика може бити текстуална, визуелна, или комбинација, где се визуелна репрезентација преписки врши више од текстуалне.

Управљање конзистенцијом представља битан аспект управљања описима комплексних архитектура које се обично дефинишу кроз више погледа што уводи проблеме конзистенције између њих. Велики број приступа моделовању кроз више гледишта, 37 од 40, подржава управљање конзистенцијом. Управљање конзистенцијом може се поделити на управљање конзистенцијом унутар или између артефаката. Око четвртина приступа, 11 од 40, подржава управљање конзистенцијом унутар артефаката (на пример погледа). Велики број приступа, 36 од 40, подржава управљање конзистенцијом између артефаката [16].

Детекција неконзистентности је прва активност управљања конзистенцијом. Детекција неконзистентности може бити мануелна, где се неконзистентности детектују од стране заинтересованих страна, или аутоматска, где се користе аутоматизовани механизми. Од 40 приступа 8 не даје информацију везану за начин детекције, а од преосталих само 1 подржава мануелну, док остали подржавају аутоматску детекцију неконзистентности. Постоји неколико идентификованих механизама за аутоматско детектовање неконзистентности. Оперативни механизам представља коришћење оперативне семантике, као што су алгоритми ради детектовања неконзистентности. Други механизам, евидентира операције измене на такав

начин да се измене урађене над артефактима чувају у посебним лог фајловима, а детекција неконзистентности врши анализом тих фајлова. Формалне методе могу бити механизам којим се детекција врши извршавањем формалних верификација као што су дедуктивни докази. Решавање проблема скупа ограничења која постоје над артефактима такође се користи као посебан механизам где су проблеми који немају решења индикација постојања неконзистентности. Оперативни механизми највише се користе, у 12 од 40, затим формалне методе у 7 од 40 приступа [16].

Разрешавање неконзистентности је друга активност управљања конзистенцијом. Разрешавање неконзистентности може бити мануелно, где се разрешавање делегира заинтересованим странама, или аутоматско, где се користе аутоматизовани механизми. Разрешавање неконзистентности је заступљено у 19 од 40 приступа, од чега 16 приступа то ради аутоматизованим механизмима. Један од аутоматизованих механизама је заступник (енгл. proxy), који ради тако што се измене урађене на артефактима, обично погледима, чувају на једном месту који представља заступника који даље пројектује измене на остале погледе. Разрешавање се може радити оперативним механизмима, као што је коришћење алгоритама. Додатно, постоји механизам разрешавања неконзистентности решавањем проблема скупа ограничења. Највише приступа, 12 од 40, користи оперативне механизме [16]. Постојећи приступи не дају велику значајност томе када треба урадити разрешавање неконзистентности. Мали број приступа разрешавање ради након сваке измене над артефактима, док остали разрешавање раде касније, обично након завршетка фазе за измене.

Моделовање кроз више гледишта у таксономији је описано и кроз подршку алата. Подршка алата подржана је од стране 28 од 40 приступа. Са друге стране, иако постоји велика подршка, постојећи приступи наводе да постоје одређена ограничења алата у погледу аутоматских механизама за креирање и синтезу артефаката као и управљање конзистенцијом.

Antonio и остали систематском претрагом литературе идентификовали су неке од најрелевантнијих приступа у моделовању заснованом на више гледишта. Таксономија коју су дефинисали формира још један оквир који се може користити за класификацију постојећих архитектуралних језика заснованих на моделовању са више гледишта. Додатно, даје смернице за развој нових језика чији је циљ креирање описа архитектура комплексних софтверских система.

3.2. Постојећи архитектурални језици

Током времена постала је пракса да се комплексне архитектуре система описују и анализирају са више гледишта [1]. Стога, у наставку ће бити описана четири најновија језика из скупа језика које су утврдили Malavolta и остали, а који описују архитектуру софтверских система са више гледишта. Језици који су у наставку описани су Drems ML, Stratus ML, X-MAN и AO-ADL и наведени су од новијих ка старијим.

Један од архитектуралних језика који се користи за опис архитектура комплексних система је Drems ML [44]. Drems ML је графички језик који подржава процес развоја заснован на компонентама. Различите фазе процеса развоја описане су посебним под-језицима језика Drems ML. На почетку моделовања могу се одвојено моделовати софтвер, хардвер и сам систем, а ова подела постоји ради одвајања интереса. На почетку моделовања софтвера дефинишу се компоненте и њихове имплементације, тако што се прво дефинишу типови

података за све атрибуте и интерфејсе које компоненте имају, као и интерфејси које компонента пружа и користи. Типови података и интерфејси дефинишу се посебним под-језиком. Потреба за посебан под-језик постоји јер се могу дефинисати кориснички типови поред унапред дефинисаних примитивних типова као и интерфејси. Интерфејси представљају колекције потписа метода које могу користити дефинисане типове као параметре. Типови података могу се дефинисати текстуалном или графичком нотацијом а додатно се спроводе и синтаксне провере.

Интеракција компоненти може бити остварена директно између две компоненте или путем објаве и претплате. Директна интеракција између две компоненте остварује се позивом методе. Интеракција путем објаве и претплате заснива се на темама. Тема садржи идентификатор и тип података. Одређена компонента објављује податке на тему које конзумирају све компоненте које су претплаћене на ту тему. На овај начин допушта се и да постоји више тема са истим типом података. За специфицирање тема користи се посебан под-језик којим се врши креирање тема и њихово повезивање са претходно дефинисаним типовима података.

Поред модела који описују типове података и теме, Drems ML даље омогућава описивање компоненти и то њихових дефиниција и имплементација. Дефиниције компоненти представљају апстрактне јединице функционалности система које могу међусобно интераговати. Дефиниција компоненте креира се посебним под-језиком и састоји се од интерфејса које она пружа и захтева, портова за објаве и портова за претплате које поседује и атрибута. Свака метода интерфејса може имати специфициране одређене особине као што је време одговора у најгорем случају. Дефиниција компоненте разликује се од имплементације и једна дефиниција компоненте може имати више различитих имплементација. Додатно, може се описати склоп компонената које групишу компоненте тако да пруже функционалност као композицију једноставнијих функција система. Имплементације већ дефинисане компоненте се креирају посебним под-језиком. Имплементације се састоје од артефаката као што су дељене библиотеке, које су потребне компоненти у време извршавања. Имплементација компоненте садржи и тајмере, који се такође могу користити за позивање одређених операција компонената. Имплементација може променити подразумеване особине портова дефинисане компоненте, као што је максимално време чекања (енгл. timeout) на одређену функцију [44].

Следеће, омогућено је описивање такозваних апликација посебним под-језиком који се састоји од проширених елемената претходних под-језика. На овај начин апликације се састоје од једне или више дефиниција компонената, које су понекад груписане у склопове, где је свака компонента повезана са својом одређеном имплементацијом. Дефиниције компонената се додатно могу доделити актерима, који чине основну јединицу за планирање у оквиру система. Апликације се састоје и од веза између портова компонената. Порт једне компоненте који користе одређени интерфејс може бити повезан на порт друге компоненте који пружа одређени интерфејс. Портови који објављују и портови који су претплаћени повезани су на одговарајуће теме, што омогућава да портови који су претплаћени на тему могу конзумирати податке које су послали портови који објављују на ту тему. На основу модела апликације може се генерисати план распоређивања, који је представљен конфигурационим фајлом који описује локације свих фајлова и осталих артефаката које су потребне за покретање апликације.

Drems ML омогућава даље повезивање апликација софтверским пакетима посебним под-језиком. Софтверски пакет дефинише интеракције портова између различитих апликација којим се омогућава да компонента из једне апликације користи интерфејс који пружа компонента из друге апликације. Софтверски пакет дефинише домене које компоненте користе за њихове портове за објаве и портове за претплате, којима се дефинише простор у којем су подаци видљиви. Софтверски пакет дефинише и план актера у оквиру апликација, када се одређени планови могу придружити актерима у одређеном временском периоду, или се оперативни систему може делегирати дефинисање планова за актере.

Поред моделовања софтвера, Drems ML омогућава моделовање хардвера. На почетку моделовања хардвера посебним под-језиком се дефинишу платформе које систем може да користи. Дефиниција платформе састоји се од уређаја, мреже, модула и њихових екстерних интерфејса. С обзиром да софтвер има строге захтеве у вези са уређајима и мрежом коју може да користи, уређаји и мрежа се описују посебним моделима, који се касније користе у конфигурацијама платформи.

Платформе се могу користи при раду система, тако што се модули, уређаји и мрежа дефинисани у оквиру њих међусобно повезују и чине одређену конфигурацију платформе. Конфигурација платформе описује се посебним под-језиком и састоји се од скупа модула и веза између интерфејса модула и мрежа као и интерфејса модула и уређаја [44]. Током времена одређене везе између модула и мрежа као и мрежне адресе интерфејса могу да се промене или одређени уређаји да се покваре, па се конфигурација платформе мења. Стога, могуће је дефинисати више конфигурација за једну платформу где свака представља посебно физичко стање система у датом тренутку. Drems ML додатно омогућава генерисање конфигурационог фајла из описа конфигурације платформе, који се може користити током распоређивања софтвера [44].

Поред моделовања софтвера и хардвера потребно је извршити њихову интеграцију у један систем. Drems ML пружа интеграцију распоређивањем софтвера на хардвер. Распоређивање софтвера посебним под-језиком описује како се софтверски пакети мапирају на одређене конфигурације платформи. Распоређивање дефинише мапирање планова на хардверске модуле на којима ће се извршавати, чиме се врши и мапирање актера на хардверске модуле. Такође, дефинише мапирање домена за објаве и претплате на мрежу која ће преносити податке.

Након описивања архитектуре користећи наведене моделе, Drems ML омогућава додатне анализе ради верификације. За потребе анализа могу се специфицирати ограничења над моделима, сигурност везана за рад хардвера, ограничења мреже и друге. Алат за моделовање може омогућити анализу ограничења дефинисаних над моделом, као што је ограничење да назначена искоришћеност процесора од стране свих компоненти буде мања од 100%.

Drems ML пружа алат за анализу ресурса мреже који ради на основу захтеваних ресурса од стране портова компоненти и пружених ресурса мреже. За портове компоненти могу се захтевати ресурси као што су минимални проток и максимално кашњење у одређеном временском интервалу. За мрежу се могу специфицирати пружени ресурси, такође минимални проток и максимално кашњење у одређеном временском интервалу [44]. Након агрегације

захтеваних и пружених ресурса и интерпретације модела система врши се анализа која утврђује да ли систем задовољава захтеве свих апликација, као што су квалитет сервиса који мрежа пружа свакој од апликација као и преостали ресурси мреже.

Пружена је анализа планова која осигурава да они могу бити извршени од стране модула којима су придружени. Планови се извршавају периодично и имају одређено трајање. С обзиром да се апликација може састојати од доста планова придружених модулима омогућавање извршавања великог броја планова периодично није тривијалан задатак. Уколико решење не постоји за извршавање дефинисаних планова алат шаље информацију да захтеви не могу бити испуњени.

Drems ML пружа временску анализу операција компоненти. Компоненте комуницирају захтевајући операције које су видљиве кроз интерфејсе других компоненти. Комуникација се на реалном систему извршава тако да се захтеви за операцијама компоненте стављају у ред за чекање одређене компоненте одакле се обрађују. Компоненте се извршавају као нити у оквиру оперативног система, а редослед извршавања операција је битан, како би се задовољили рокови за извршавање дефинисани за сваку операцију. За ове потребе алат омогућава временску анализу у време дизајна, како би обезбедио да свака компонента изврши своје операције у захтеваном временском року.

Drems ML је архитектурални језик који омогућава креирање описа архитектуре кроз више погледа дефинисаним посебним под-језицима и користи се за описивање комплексних архитектура софтверских и хардверских система. Креирање модела омогућено је у оквиру едитора за моделовање којим се могу цртати елементи описа. Додатно, постоји подршка за алате којима се врше анализе интегрисаног система ради његове реалне употребе.

Пример другог архитектуралног језика који се користи за опис архитектура комплексних система је Stratus ML [56]. Stratus ML је графички језик који служи за описивање дистрибуираних апликација које треба да буду распоређене у облаку. Раздвајање интереса заинтересованих страна ради се кроз слојеве.

Stratus ML користи пет слојева а то су: језгро које дефинише композицију сервиса, адаптација која дефинише динамичко понашање и еластичност, доступност које дефинише дистрибуцију и репликацију, провајдер који дефинише ресурсе и цене, и последњи слој перформансе. Сваки слој описан је посебним мета-моделом. Приказивањем или сакривањем одређених слојева у оквиру едитора за моделовање заинтересованим странама се пружају различити погледи описа архитектуре [56].

Слој језгро је централни модел који представља концепте већине провајдера и описује распоређивања кроз задатке и интеракције, где се задатак односи на софтверски модул који је сачињен од активности. Слој језгро омогућава дефинисање сервиса и иницијалних конфигурација. Сервис је сачињен од неколико компоненти које се могу груписати тако да се одређене особине као што је ограничење локације могу доделити одређеним компонентама или њиховим групама. Остали слојеви представљају надоградњу слоја језгра.

Слој адаптације се користи да дефинише правила адаптације и акције за сваки задатак или групу задатака дефинисаних у језгру. Постоје предефинисане, као што је акција скалирања, и кориснички дефинисане акције. Акције се покрећу на основу ограничења, као што је највећи

и најмањи број инстанци који је дозвољен у одређено време, или правила адаптације. Правила адаптације су динамичка ограничења која се заснивају на одређеним параметрима перформанси као што је искоришћење процесора.

Слој доступности пружа компоненте које су неопходне како би се језгро инстанцирало и његове инстанце дистрибуирале на различите географске локације. Слој провајдер дефинише шаблоне за сервисе и цене. Слој перформансе је остављен за каснију имплементацију па није разрађиван.

Модел описан кроз наведене слојеве је платформски независтан. Stratus ML омогућава даљу валидацију платформски независног модела посебним доменски специфичним језицима. Валидација се заснива на дефинисању правила која могу бити тврда правила односно она која едитор проверава и која корисник не може да прекрши и мека правила која корисник може да прекрши али едитор исписује упозорења и грешке [56]. Валидација омогућава проверу структуре модела посебним структурним тврдим правилима која проверава едитор за моделовање. Пример стуктурног правила је да је празан модел невалидан, односно да се мора састојати од најмање једне компоненте. Поред валидације структуре омогућена је семантичка валидација модела семантичким тврдим или меким правилима као што је да сваки задатак мора имати јединствен назив. Додатно, омогућено је дефинисање услова под којим ће се провера семантичког правила обавити. Последњи вид валидације који се користи је валидација трансформација, односно дефинишу се правила која служе за проверу валидности платформски независинсног модела при трансформацији у платформски специфични модел.

Stratus ML омогућава даљу трансформацију валидног платформски независног модела у платформски специфичне моделе користећи шаблоне који се састоје од правила за трансформацију. Специфични модел добија се заменом референци које постоје у правилима шаблона елементима који долазе из платформски независног модела. Након трансформације врши се генерисање артефаката, који представљају код, конфигурационе фајлове и остало, који служе за покретање апликације на циљну платформу.

Stratus ML је архитектурални језик који омогућава креирање описа архитектуре кроз више слојева дефинисаним посебним метамоделима. Сакривањем или приказивањем слојева пружају се различити погледи што омогућава описивање комплексних архитектура. Stratus ML се користи за описивање архитектура апликација у облаку. Креирање модела омогућено је у оквиру едитора за моделовање којим се могу цртати елементи описа. Постоји подршка за валидацију модела што омогућава управљање конзистенцијом и трансформацију платформски независног модела у модел специфичан за платформу као и креирање артефаката ради покретања на циљној платформи. Додатно, дефинисањем платформски независног модела Stratus ML омогућава и лаку миграцију између платформи.

Пример трећег архитектуралног језика који се користи за опис комплексних система је X-MAN [60]. X-MAN је графички језик који служи за описивање комплексних софтверских система и заснован је на компонентама вођеним развојем којим се компоненте креирају и постављају на репозиторијум из којег се касније могу преузети и поново користити. На почетку развоја заснованом на компонентама дефинишу се компоненте у одређеном домену проблема како би оне биле независне од конкретног система. Тако би компоненте могле поново да се употребе у свим системима који се описују у посматраном домену. Затим се врши валидација

и верификација компоненти и оне се постављају на репозиторијум. Описивање система врши се касније, након постављања потребних компоненти на репозиторијум. Дефинишу се кориснички захтеви, врши се спецификација система на основу које се користе компоненте из репозиторијума ради описивања система.

У језику X-MAN компоненте су јединице структуре и понашања. Структура је дефинисана композицијом компоненти где се компоновање врши посебним конекторима. Понашање се дефинише кроз интерфејсе компоненти које она пружа другим компонентама. Свака компонента која није композиција других је атомска компонента и она има своју имплементациону јединицу која имплементира методе које компонента може пружити својим интерфејсима и тако дефинише понашање. Компоненте које компонују неку већу компоненту међусобно не позивају методе друге компоненте већ се њихова комуникација обавља преко конектора за композицију. X-MAN пружа посебан поглед на атомске компоненте [60].

Поред дефинисања композиције компоненти, омогућено је дефинисање тока контроле. Конектор може да омогући секвенцирање компоненти тако да користи понашање једне па понашање друге компоненте или селектовање тако да омогући њихово рачвање. Такође, постоје посебни конектори који се користе ради дефиниције понашања компоненте у петљи или под одређеним условом. X-MAN пружа посебан поглед на ток контроле [60].

X-MAN дефинише ток података каналима за податке који повезују одређене компоненте. Ток података може бити хоризонталан или вертикалан. Хоризонталан ток података може постојати између под компонената одређене компоненте. Вертикалан ток података може постојати између композитне компоненте и њених под-компоненти. Додатно, канал за податак може бити иницијализован одређеном вредношћу. X-MAN пружа посебан поглед на ток података [60].

Компонентама вођен дизајн X-MAN пружа првобитним креирањем компоненти у одређеном домену. Компоненте се постављају на репозиторијум и касније користе у развоју система. У фази развоја система компоненте се инстанцирају, иницијализују и прилагођавају. Компоненте у оквиру свог интерфејса могу дефинисати и податке који се описују њиховим типом, подразумеваном вредношћу и дозвољеним вредностима. Иницијализација компоненте представља постављање вредности за податак у њеној инстанци. Прилагођавање компоненти представља изабир одређених метода из компоненте које ће бити присутне у њеној инстанци.

X-MAN пружа алат који се састоји од дизајнера компоненти и дизајнера система [60]. Дизајнер компоненти омогућава описивање компоненти, њихову валидацију, постављање на репозиторијум. Описивање компоненти започиње се описивањем атомских компоненти тако што се дефинише назив компоненте, њена имплементациона јединица за коју се може посебно везати код написан у одређеном програмском језику и интерфејс који компонента пружа. Затим се врши описивање композитних компоненти кроз атомске компоненте, конекторе, интерфејсе и вертикалне и хоризонталне токове података. Валидација композитних компоненти остварује се проверама исправности токова контроле и токова података у композицији. Постављање одређених компонената на репозиторијум спроводи се уз њихову претходну валидацију. Репозиторијум омогућава одређене функционалности као што су претрага, уклањање, сортирање, извожење и категоризација компоненти.

Дизајнер система омогућава описивање система, валидацију, симулацију и верификацију. Описивање система спроводи се инстанцирањем, иницијализацијом, прилагођавањем компоненти са репозиторијума као и њиховом композицијом користећи конекторе. Валидација система остварује се слично као валидација компоненти, проверама исправности токова контроле и токова података у композицији инстанци. Симулација система спроводи се евалуацијом излаза система користећи скуп тест примера изведених из захтева система, како би се посматрало понашање система. Последње, верификација система се спроводи ради провере задовољења дефинисаних захтева.

X-MAN је архитектурални језик заснован на компонентама вођеном развоју који омогућава креирање описа архитектуре кроз више погледа и користи се за описивање комплексних архитектура софтверских система. Креирање модела омогућено је у оквиру едитора за моделовање којим се могу дефинисати компоненте, њихова структура и понашање, ток контроле, ток података и компоненте поставити на репозиторијум ради поновног коришћења. Компоненте се могу поново користити у описивању конкретног система њиховим инстанцирањем и композицијом. Додатно, постоји подршка за валидацију компонената и описа архитектуре система што омогућава управљање конзистенцијом као и подршка за симулацију и верификацију описа архитектуре система.

Пример четвртог архитектуралног језика који се користи за опис архитектура комплексних система је AO-ADL [54]. AO-ADL је текстуални и графички језик који је заснован на аспектно оријентисаном приступу, при чему компонентама поред структуре и понашања система које ћемо звати основним компонентама, описује и аспекте који се прожимају кроз читав систем, као што су сигурност, право приступа, руковање грешкама, синхронизација, које ћемо звати аспектуалним компонентама. Како би и аспекте описао компонентама AO-ADL проширује класичну семантику конектора којим уводи аспекте у интеракцију компонената.

AO-ADL пружа описивање компонената њиховом називом и интерфејсима. Интерфејси се дефинишу на посебним местима и они се могу поново употребити у оквиру више различитих компонената као интерфејси које компонента пружа или захтева. Интерфејсима се могу доделити типови који описују семантику операција. Тип интерфејса може бити порука која описује синхронизовану интеракцију између компонената, догађај који описује асинхрону интеракцију између компонената, атрибут који описује манипулацију стања компоненте, изузетак који описује бацање или руковање изузецима од стране компонената, фабрика која описује креирање нових инстанци компонената. Интерфејсима се на крају додељују листе операција чиме се њихова дефиниција завршава. Дефинисане компоненте могу се структурно компоновати у композитне компоненте које се поред навођења под-компонената на исти начин описују својим називом и скупом пружених и захтеваних интерфејса. Компоненте се као и интерфејси описују на посебним местима па их композитне компоненте могу касније поново употребити.

AO-ADL пружа описивање конектора прво произвољним бројем улога које компоненте могу имати у међусобној интеракцији, а улоге могу бити захтеване, пружене и аспектуалне [54]. У случају основних компоненти интерфејс који компонента пружа може бити повезан са улогом коју конектор захтева, док интерфејс који компонента захтева може бити повезан са улогом који конектор пружа. У случају аспектуалних компоненти, интерфејс који компонента пружа може бити повезан са аспектуалном улогом конектора. Улоге конектора могу имати

типове као што су порука, догађај и остали који ограничавају број интерфејса који се са њима могу повезати. На овај начин конектори дефинишу произвољан број компонената које имају своју улогу у композицији понашања. AO-ADL пружа могућност повезивања улога конектора са интерфејсима компонената и кроз џокер обрасце којима се може дефинисати скуп интерфејса који могу да обављају одређену улогу конектора. Конектори се описују засебно, па се тиме као и употребом џокер образаца повећава могућност њихове поновне употребе.

Конектори даље описују композиције понашања компонената кроз везивања основних компонената и везивања аспекатуалних компонената. Везивања основних компонената се остварују индиректно везивањем захтеваних и пружених улога, док везивања аспектуалних компонената специфицирају како се аспектулане компоненте преплићу са основним компонентама, а њихово дефинисање заснива се на тачкама пресека [54]. Тачке пресека омогућавају пресретање интеракције између компонената и дефинишу се над пруженим улогама, захтеваним улогама и над везама између пружених и захтеваних улога. Тачке пресека дефинисане над пруженим улогама омогућавају пресретање позива операција од стране компонената које су повезане на пружене улоге. Тачке пресека дефинисане над захтеваним улогама омогућавају пресретање пријема одговора од операција које су позване над компонентама повезаним на захтеване улоге. Тачке пресека дефинисане над везама између пружених и захтеваних улога омогућавају пресретање операција приликом интеракције компонената које су повезане на одговарајуће улоге. Дакле, приликом описивања тачака пресека специфицирају се улоге и додатно операције које оне пресрећу.

Везивање аспектуалне компоненте остварује се индиректно везивањем аспектуалних улога са тачкама пресека између пружених и захтеваних улога. На овај начин интеракција основних компонената може бити допуњена аспектуалним компонентама такозваним саветима. Савети се дефинишу над тачкама пресека самим тим над операцијама које се пресрећу. Савети представљају аспектуално понашање које може бити додато пре или после операције, као и уместо или конкурентно са самом операцијом. Спецификација савета састоји се од одређивања тачке пресека, аспектуалне улоге конектора, дефинисања када се аспектуално понашање додаје у тачку пресека, као и дефинисање аспектуалног понашања у виду операције аспектуалне компоненте која ће бити додата [54]. Додатно, могу бити дефинисани услови у којима се додају аспектуална понашања. У случају да је дефинисано више савета над једном тачком пресека, потребно је дефинисати редослед којим ће савети бити додавани. Овим је дефиниција конектора завршена.

AO-ADL пружа алат за рад са репозиторијумом, алат за спецификацију, алат за генерисање дизајна и алат за генерисање кода. Специфицирани елементи архитектуре, компоненте и њихове композиције као и конектори могу бити складиштени у репозиторијуму. Репозиторијум пружа функционалности складиштења, брисања, претраге архитектуралних елемената у оквиру посебних фолдера. Елементи складиштени на репозиторијуму могу се поново користити при описивању других архитектура.

Алат за спецификацију састоји се од неколико едитора за моделовање који омогућавају описивање архитектура кроз компоненте, интерфејсе, конекторе. Могу се користити нови или користећи постојећи елементи из репозиторијума. Алат може урадити проверу да ли се опис састоји од добро формираних елемената, као и да ли је опис комплетан уз омогућено дефинисање делимично специфицираних описа. Додатно, пружена је могућност да се сагледа

комплетна листа компонената које учествују у конектору посебним погледом Конектори, као и листа конектора у којима учествује једна компонента посебним погледом Компоненте [54].

Постоји посебан алат који служи за генерисање детаљног дизајна на основу креираног описа. Генерисани дизајн представља почетну верзију која се може дорадити додавањем нових аспеката који се могу наћи у фази детаљног дизајна, и увођењем нових детаља у постојеће елементе. Могуће је генерисати аспектно оријентисани дизајн представљен језиком Theme/UML или дизајн који није аспектно оријентисан и представљен језиком UML [54]. Генерисање скелета детаљног дизајна је омогућено претходним дефинисањем одговарајућих трансформација из AO-ADL у Theme/UML или UML посебним језиком. Аспектуалне информације се у језику Theme/UML моделују аспектно-оријентисаним механизмима језика Theme/UML. Компоненте и њихова статичка композиција дефинисана у AO-ADL се у UML описује дијаграмима компонената, док се основна и аспектуална понашања у UML описују дијаграмима секвенце. Предност генерисања детаљног дизајна из описа архитектуре огледа се у томе да су аспекти дефинисани у фази описивања архитектуре присутни и касније у фази детаљног дизајна, односно дизајна на ниском нивоу.

Постоје два приступа за генерисање софтверског кода. Први приступ је генерисање кода на основу већ генерисаног детаљног дизајна користећи постојеће трансформације. Стога, већ генерисани дизајн у Theme/UML или у UML се може превести у софтверски код, што значи да интереси дефинисани у опису архитектуре могу бити задржани и на нивоу имплементације. Други приступ је генерисање кода директно из описа архитектуре и њега пружа последњи алат AO-ADL језика. Конкретно, омогућено је генерисање AspectJ или JBoss AOP кода на основу AO-ADL описа [54].

AO-ADL је архитектурални језик заснован на аспектно-оријентисаном приступу који омогућава креирање описа кроз поглед Компоненте и поглед Конектори. За разлику од осталих језика заснованим на компонентама као основним архитектуралним елементима уводи аспектуалне компоненте и омогућава симетрично моделирање којим се основни и аспектуални интереси моделују на исти начин користећи компоненте. AO-ADL поред едитора за моделовање пружа алат за коришћење репозиторијума као и алате за генерисање детаљног дизајна и софтверског кода.

4. Поставка проблема

Процес и приступ у креирању софтверских система зависи доста од намене и области у којој се системи користе, као и од њихове величине и комплексности. Намена софтверских система између осталог може бити за потребе индустрије као и потребе образовања. Софтверски системи могу садржати велики број комплексних функционалности или бити доста једноставни и обухватити само одређен број функционалности. Генерално процес креирања софтвера састоји се од следећих фаза: дефинисање захтева, креирање описа архитектуре, дизајн на ниском нивоу, имплементација [2], са тим да се ове фазе не морају увек радити једна за другом или у овом редоследу, већ се неке од њих у пракси врло често раде заједно. Пример заједничког обављања фаза био би дефинисање функција система у оквиру фазе креирања описа архитектуре при недостатку дефинисаних захтева за системом [26]. Неке фазе у процесу креирања софтверског система могу бити прескочене, обично у случају једноставнијих система.

Дефинисање захтева, као прва фаза процеса креирања софтверског система, може бити једноставно записивање захтева у текстуалном едитору, или записивање посебним језицима у алатима за те намене, а као резултат даје скуп функционалних и нефункционалних захтева који софтверски систем треба да поседује. Креирање описа архитектуре, као наредна фаза, има задатак да дефинисане функционалне и нефункционалне захтеве адресира представљајући их на одређен начин у опису архитектуре. Додатно, пре креирања самог описа потребно је идентификовати све заинтересоване стране за системом и пронаћи одговарајући архитектурални језик и алат који ће се користити. Следећа фаза је дизајн на ниском нивоу при чему се архитектурални елементи даље разрађују у целине које се могу имплементирати. Имплементација обухвата писање програмског кода који ће реализовати идентификоване целине система. Поред описаних фаза софтверски системи се у великој мери и тестирају.

Разлог за описивање архитектуре софтверског система је представљање различитих аспеката система заинтересованим странама као и да прикаже да ли су њихови интереси за системом испуњени. Како би представљени аспекти система били разумљиви они морају бити представљени на одговарајући начин тако да се детаљи који нису од интереса сакрију, а прикажу детаљи који заступају интересе заинтересованих страна. Представљање аспеката комплексних система може резултовати комплексним описом који није разумљив, па је потребно користити посебан приступ.

Креирање описа архитектуре комплексних софтверских система ради се са више гледишта што омогућава креирање погледа као делова описа [13]. Велики број постојећих архитектуралних језика (91%) пружа подршку за више погледа [2]. На тај начин заинтересоване стране за софтверски систем као што су архитекте, инжењери, администратори, менаџери и остали имају могућност да систем сагледају из своје перспективе [37]. Са друге стране, описивање архитектуре погледима уводи проблем конзистенције међу њима [13]. Стога је управљање конзистенцијом саставни део многих архитектуралних језика који описују комплексне софтверске системе. Јавља се потреба да се детектују и забележе све неконзистентности између погледа или у оквиру погледа. Такође, постоји потреба за дефинисањем правила која представљају начин да се изразе, забележе, спроведу и анализирају

конзистентности. Не захтева се разрешавање неконзистености одмах након детекције, већ је у одређеним ситуацијама чак добро толерисати их све док развој не достигне одређену фазу [16].

Неколико оквира за управљање конзистенцијом који су дефинисани током година су обједињени од стране Spanoudakis и осталих [62] као скуп активности који прецизније одражавају операционализацију процеса за управљање конзистенцијом различитим техникама и методама које су развијене да га подрже. Ове активности су детекција преклапања, детекција неконзистентности, дијагноза неконзистентности, руковање неконзистентностима, праћење неконзистентности и спецификација и примена политике за управљање неконзистентностима, и неке од њих се могу урадити одређеном техником или методом. Antonio и остали [16] у њиховом систематском прегледу литературе идентификовали су сличан скуп активности процеса за управљање неконзистентностима који је праћен њиховим скупом техника и метода. Штавише, Antonio и остали тврде да када год је више погледа креирано одвојено постизање конзистентности архитектуралног описа се заснива на механизмима резолуције који су посебно дефинисани над паровима погледа. Замршеност постизања конзистенције представљена је неопходношћу дефинисања доста бинарних релација конзистентности и одговарајућих процедура обнављања, а могуће је и појављивање ефекта таласања који може довести до неконфлуентног процеса. Проблем n-арних релација конзистентности остаје неразрешен јер поседује озбиљне потешкоће и са теоријске и са практичне стране.

У анализи постојећих архитектуралних језика Ozkaya [2] је евалуирао и визуелне и текстуалне језике и показао да већина њих пружа подршку за више гледишта. Међутим, само део анализираних језика се бави управљањем конзистенцијом. Постојећи архитектурални језици који пружају креирање описа кроз више гледишта, то обично раде погледима који се креирају одвојено. Погледи су тада представљени одвојеним текстуалним описима или одвојеним дијаграмима. Погледи који су на једном месту могу отежати разумевање описа заинтересованим странама [13], док погледи на одвојеним местима могу отежати одређене активности управљања конзистенцијом. Уколико би се омогућило креирање интегралног модела, који би погледе представљао на једном месту, на пример у једном текстуалном опису, али тако да заинтересоване стране имају доступне погледе који су њима од интереса, то би могло да олакша одређене активности за управљање конзистенцијом. Према томе, уводе се полазне хипотезе: „Могуће је користити интегрално моделирање за креирање описа архитектуре софтверског система кроз више погледа.“, као и: „Коришћење новог архитектуралног језика може олакшати уочавање неконзистентности у опису архитектуре у поређењу са архитектуралним језиком који се тренутно користи у пракси.“.

Како би интегрални модел направљен у оквиру фазе описивања архитектуре софтверског система могао да се користи у осталим фазама процеса развоја софтвера јавља се проблем интеграције са другим језицима и решењима у оквиру посматраног процеса. Уколико би било могуће превести интегрални модел у одговарајући модел другог архитектуралног језика који се користи у пракси и који има добру подршку у процесу развоја софтвера омогућила би се примена интегралног модела. Додатно, практичној примени би допринела аутоматизација процеса превођења интегралног модела у модел који се користи у пракси посебним алгоритмима. Према томе уводи се полазна хипотеза: „Опис креиран предложеним архитектуралним језиком је могуће аутоматски превести у одговарајући опис за архитектурални језик који се тренутно користи у пракси.“.

Интегрални модел у себи садржи више погледа на опис архитектуре који заступају интересе заинтересованих страна са циљем олакшавања каснијег управљања конзистенцијом, али са друге стране сагледавање свих погледа заједно без сакривања детаља који заинтересованој страни нису битни може бити проблем. Уколико би постојало средство, у виду алата које би омогућило да се интегрални модел посматра са више различитих гледишта, тако да заинтересована страна може да сакрије погледе који јој нису битни, а да се фокусира на погледе који су јој од интереса то би водило решењу проблема. Додатно, решењу проблема допринело би и постојање методе која у случају приказивања више погледа омогућава да се сваки од њих лако идентификује и прати.

Одређени архитектурални језици, посебно они који су опште намене односно који се користе за креирање модела који не морају нужно да служе у описивању архитектуре [13] садрже велики број елемената који може отежати разумевање. Постојање језика који су намењени конкретно за описивање архитектуре система може водити ка једноставнијем мета-моделу као и лакшем разумевању од стране практиканата. Поставља се полазна хипотеза: „Коришћење новог архитектуралног језика може студентима бити лак за разумевање у поређењу са архитектуралним језиком који се тренутно користи у пракси.“ која би се проверила у домену образовања.

У пракси архитектурални језици користе и текстуалну и графичку нотацију ради описивања комплексних система, међутим текстуална нотација је погоднија за описивање комплексних архитектура, док графичка доприноси ефикасној комуникацији решења [2]. Њихова употреба највише зависи од искустава и потреба практиканата. Уколико би се интегрални модел описао текстуалном нотацијом што би допринело приближавању погледа то би могло да допринесе лакшем описивању комплексних система, па се уводи полазна хипотеза: „Коришћење новог архитектуралног језика може студентима олакшати креирање описа архитектуре у поређењу са архитектуралним језиком који се тренутно користи у пракси.“.

Поред коришћења у индустрији архитектурални језици се могу користити у образовању како би се поред упознавања студената са основним концептима архитектуре софтверских система и практичне примене алата у том домену, студентима приближио логички дизајн, декомпозиција и решавање проблема. Током уводних курсева програмирања студенти решавају проблеме и имплементирају их програмским језицима. Након уводних курсева програмирања студентима и даље изазов представљају декомпозиција проблема, развој планова и њихова имплементација како би те проблеме решили [3]. Поред непостојања адекватне литературе или пракси за адекватан пренос знања, разлог постојања овог проблема може бити и превелика употреба алата од стране студената због очекивања тржишта у погледу овладавања професионалним алатима [4,5]. Пракса коришћења великог броја алата нарочито је заступљена на напредним курсевима специфичног домена, као што су курсеви везани за информационе системе. На пример, циљ курса Информациони системи 1 на Електротехничком факултету Универзитета у Београду је да се предају концепти дизајна система, анализе и имплементације. Током овог курса студенти уче традиционални и објектно-оријентисани дизајн система и анализу засновану на UML и развој система заснован на Java Enterprise. Иако студенти уче UML и пратеће алате, још увек им недостаје способност да се усредсреде на решавање проблема и схвате шире концепте дизајна информационог система, укључујући суштинске логичке провере и ограничења, односно постоји јаз између логичког дизајна и имплементације

софтверских система. Проблем се може интерпретирати као како увести методологије решавања проблема и логичког дизајна у курс који покрива дизајн и имплементацију софтверских система.

Уколико би се у архитектурални језик имплементирао одређени методолошки приступ решавању проблема можда би се студентима олакшало решавање логичких проблема. Jeannette Wing је увела нови методолошки приступ у решавању проблема под називом рачунарско размишљање као менталну активност за формулисање проблема који се може решити рачунарски [7]. Рачунарско размишљање чине четири стуба или технике који се независно користе у решавању проблема. Први стуб назван декомпозиција, представља идентификовање проблема и декомпозицију комплексних проблема на мање целине односно под-проблеме који су решиви. Други стуб назван препознавање образаца, представља идентификовање истих проблема на различитим местима у декомпозицији и поновно коришћење њихових решења. Трећи стуб, назван апстракција, односи се на идентификовање само релевантних детаља, њиховог укључивања у декомпозицију, док се остали детаљи занемарују. Четврти стуб назван алгоритми, чине решења једноставних проблема. Додатно, декомпозиција система на његове функције у последњем издању смерница курикулума рачунарске науке (CS2013), се помиње као парадигма дизајна, названа структурирани дизајн, који изводи функционалну декомпозицију одозго према доле [9]. Прво питање је да ли је могуће имплементирати рачунарско размишљање у архитектурални језик. Стога се уводи полазна хипотеза: „Могуће је предложити нов архитектурални језик који би подржао све принципе рачунарског размишљања.“. Друго питање је да ли овакав језик олакшава креирање описа па тако и решавање логичких проблема. Овај одговор се може добити тестирањем уведене хипотезе: „Коришћење новог архитектуралног језика може студентима олакшати креирање описа архитектуре у поређењу са архитектуралним језиком који се тренутно користи у пракси.“.

Предмет дисертације су архитектурални језици за опис софтверских система. Дисертација обухвата анализу постојећих архитектуралних језика и преглед њихових приступа кроз постојеће оквире и особине архитектуралних језика који су дефинисани у литератури [2,16]. Дисертација даје преглед неколико најновијих архитектуралних језика који креирају описе архитектура софтверских система са више гледишта и тако могу да се користе у описивању архитектура комплексних система, преглед њихових приступа у креирању погледа, приступа за управљање конзистенцијом, као и преглед одређених функционалности њихових алата и едитора за моделовање.

Циљ дисертације је предлог новог архитектуралног језика. Предложени архитектурални језик поред креирања описа архитектуре, омогућава подршку за више погледа на опис архитектуре и тиме омогућава креирање описа архитектуре комплексних софтверских система. Подршку за решавање проблема конзистенције између погледа предложени архитектурални језик омогућава интегралним моделовањем. Како би предложени језик био лак за разумевање и како би олакшао решавање логичких проблема и креирање описа архитектуре у поређењу са архитектуралним језиком који се користи у пракси језик је заснован на принципима рачунарског размишљања и функционалне декомпозиције. Ради интеграције предложеног језика у процес развоја софтвера језик омогућава превођење својих описа архитектуре у описе постојећег архитектуралног језика који се користи у пракси. Додатно, предложени језик пружа алат како би се повећала његова практична употреба, олакшало

управљање описима и конзистенцијом међу погледима и олакшало превођење описа предложеног језика у опис архитектуралног језика који се тренутно користи у пракси.

Значај дисертације, поред прегледа неколико постојећих, огледа се у предлогу новог архитектуралног језика за описивање комплексних софтверских система који својим интегралним моделом уводи посебан приступ креирању описа архитектуре са више гледишта. Овакав приступ олакшава управљање конзистенцијом у поређењу са архитектуралним језиком који се користи у пракси. Рачунарско размишљање као методолошки приступ решавању проблема имплементирано је у предложени језик заснован на функционалној декомпозицији као методолошкој парадигми дизајна и тиме се омогућава олакшано креирање описа и решавање логичких проблема, као и лако разумевање предложеног језика у поређењу са архитектуралним језиком који се користи у пракси. Такође, значај се огледа интеграцијом језика у процес развоја софтвера могућношћу превођења описа предложеног језика у опис архитектуралног језика који се користи у пракси. Додатно језик пружа алат којим се олакшава управљање описом, у смислу селекције, сакривања и разликовања одређених погледа, олакшава превођење описа и повећава његову практичну употребу у софтверској индустрији и образовању.

Све до сада уведене полазне хипотезе овде су написане обједињено. Прва полазна хипотеза је: „Могуће је предложити нов архитектурални језик који би подржао све принципе рачунарског размишљања.“. Друга полазна хипотеза је: „Могуће је користити интегрално моделирање за креирање описа архитектуре софтверског система кроз више погледа.“. Трећа полазна хипотеза је: „Опис креиран предложеним архитектуралним језиком је могуће аутоматски превести у одговарајући опис за архитектурални језик који се тренутно користи у пракси.“. Четврта полазна хипотеза је; „Коришћење новог архитектуралног језика може студентима олакшати креирање описа архитектуре у поређењу са архитектуралним језиком који се тренутно користи у пракси.“. Пета полазна хипотеза је: „Коришћење новог архитектуралног језика може студентима бити лак за разумевање у поређењу са архитектуралним језиком који се тренутно користи у пракси.“. Шеста полазна хипотеза је: „Коришћење новог архитектуралног језика може олакшати уочавање неконзистентности у опису архитектуре у поређењу са архитектуралним језиком који се тренутно користи у пракси.“.

Како би се тестирало да ли су полазне хипотезе тачне прво је направљен архитектурални језик заснован на функционалној декомпозицији назван АФД. Затим је направљен алат за коришћење АФД. На крају су спроведене одговарајуће евалуације ради тестирања полазних хипотеза над имплементираним језиком и његовим алатом.

5. Архитектурални језик заснован на функционалној декомпозицији - АФД

Функционална декомпозиција је процес разлагања функције на њене саставне делове, односно под-функције, на начин да се оригинална функција може поново компоновати од ових делова кроз функционалну композицију. Недавно истраживање функционалне декомпозиције евалуирало је могуће сценарије примене. На пример, због практичног ограничења да захтеви нису увек доступни пре функционалне декомпозиције, процес се може применити у паралели са дефинисањем захтева [26]. Када се нефункционални захтеви морају евалуирати, може се применити функционална декомпозиција проширивањем семантике UML [27]. У оквиру дељених окружења апликација у облаку са више закупаца, функционална декомпозиција се може применити са намером да се одвојено анализира, дизајнира, развија и одржава више пословних процеса [25]. Поред ових примена функционалне декомпозиције, функционални блок дијаграми (енгл. FFBD) се дуго користе у декомпозицији комплексних система у различитим доменима [10].

Као графички језик који дефинише односе свих функција које систем обавља, FFBD декомпонује комплексне функције и представља токове контроле кроз дијаграме [63]. Проширену верзију FFBD (EFFBD) карактерише додатна могућност да прикаже токове података кроз дијаграме. EFFBD не решава проблем двосмислености података, па је неопходан додатан рад на креирању концептуалног модела података, именован као речник података, који је повезан са улазима и излазима функције [64]. Додатно проширење EFFBD именовано xFFBD, исказало је потребу за креирањем „самосталних“ модела који обухватају и повезују више дисциплина, од фундаменталних као што су механика, електроника, термодинамика до попречних као што су интегрисана логистичка подршка, управљање ланцем снабдевања и људски фактори [65]. Све тренутно доступне имплементације функционалне декомпозиције, у форми FFBD или EFFBD, уколико се користе у домену информационих система, могу представити декомпозицију функција, ток контроле, ток података, и поновну употребу функција али им недостаје могућност да прикажу имплементационе аспекте [10].

Анотирана функционална декомпозиција (АФД) је текстуални архитектурални језик који омогућава креирање описа архитектуре комплексних софтверских система. Идеја која стоји иза АФД је да се користи функционална декомпозиција као методолошка парадигма за дизајн са циљем да се олакша разумевање сагледавањем система кроз његове саставне делове. Да би се олакшало разумевање анотације су представљене у пет нивоа декомпозиције. Први ниво је обавезан, док су остала четири ортогонална па тако и опциона. Први ниво описује декомпозицију, други ток контроле, трећи ток података, четврти поновну употребу, а пети описује имплементацију. Сваки ниво ће бити даље објашњен у примерима који су дати на Сликама 1-5 које приказују поједностављен процес плаћања у продавници.

Први ниво декомпозиције представља основну структуру система дефинишући функције система (Слика 1). Функције система могу се даље декомпоновати на под-функције. Функције дефинисане у овом нивоу могу бити анотиране у другим нивоима декомпозиције.

```
1   Payment
2         Input
3         CreatePayment
4         TryPayment
5               GetOrderSum
6                     GetItemsForOrderId
7                           GetDatabasePersistenceManager
8                           GetOrder
9                           GetItems
10                    InitialSum
11                    GetSumForItems
12                          GetItemPrice
13                          GetItemCount
14                          IncreaseSum
15              Transaction
16              TransactionDone
17                    PaymentSuccesful
18                          WriteSuccessToLog
19                          ChangeToSuccesful
20                    PaymentFailed
21                          WriteFailureToLog
22                          ChangeToFailed
23        RecordPayment
24              GetDatabasePersistenceManager
25              SavePayment
26              GetItemsForOrderId
27              UpdateInventory
28        Output
```

Слика 1 Први ниво декомпозиције - Функције за пример Процес плаћања у продавници.

Први ниво декомпозиције дефинише основну структуру функције увлачењем њених под-функција. Функција Payment прима улазне податке, агрегира улазне податке иницијализацијом плаћања, покушава плаћање, може забележити плаћање и даје податке као резултат. Како би се покушало плаћање, израчуна се сума дохватањем броја и цене артикала, обави се трансакција и евидентира исход трансакције. Функција која евидентира плаћање дохвата менаџера перзистенције, евидентира плаћање и уклања наручене артикле. Након дефинисања основне структуре система могу се увести остали нивои декомпозиције.

Други ниво декомпозиције представља скуп анотација које дефинишу ток контроле система (Слика 2). Ток контроле дефинише редослед извршавања функција, условно извршавање, извршавање петље и паралелно извршавање. Редослед извршавања функција дефинисан је додељивањем редних бројева функцијама. Условно извршавање функције и извршавање у петљи постиже се додељивањем услова функцији и обележавањем функције за условно извршавање или извршавање у петљи респективно. Група анотација може бити обележена ради паралелног извршавања.

Други ниво декомпозиције дефинише редослед функција писањем редног броја пре назива функције. Функција у линији 3 прима улазне податке, након чега функција у линији 4 иницијализује плаћање. Функција у линији 10 декомпозиције израчунава суму за артикле у петљи која итерира кроз скуп претходно дохваћених артикала. Петља се дефинише писањем симбола звездице (*) након редног броја и писањем услова након дефиниције функције. Функције у линијама 16 и 19 које евидентирају исход плаћања су ексклузивне и њихови услови су дисјунктни. Ексклузивна функција се дефинише писањем слова е након редног броја.

```
1   1 Payment
2         1 Input
3         2 CreatePayment
4         3 TryPayment
5               1 GetOrderSum
6                     1 GetItemsForOrderId
7                           1 GetDatabasePersistenceManager
8                           2 GetOrder
9                           3 GetItems
10                    2 InitialSum
11                    3* GetSumForItems /item in items
12                          1 GetItemPrice
13                          2 GetItemCount
14                          3 IncreaseSum
15              2 Transaction
16              3 TransactionDone
17                    1e PaymentSuccesful /status==200 AND (Initialized)
18                          1 WriteSuccessToLog
19                          2 ChangeToSuccesful
20                    1e PaymentFailed /status!=200 AND (Initialized)
21                          1 WriteFailureToLog
22                          2 ChangeToFailed
23        4? RecordPayment /status==200
24              1 GetDatabasePersistenceManager
25              2p SavePayment
26              2p GetItemsForOrderId
27              3 UpdateInventory
28        5 Output
```

Слика 2 Други ниво декомпозиције – Ток контроле за пример Процес плаћања у продавници.

Функција у линији 22 евидентира плаћање само у случају да је плаћање успешно. Функција која евидентира плаћање у линији 24 и функција која дохвата артикле у линији 25 су обележене да се изврше у паралели писањем слова p након редног броја.

Трећи ниво декомпозиције представља скуп анотација које дефинишу ток података система (Слика 3). Ток података дефинише улазне објекте података, излазне објекте података, улазне токове и излазне токове. Објекат података може имати делове који су објекти података. Улазни ток може бити декомпонован на скуп објеката података, док излазни ток може бити компонован од скупа објеката података. Улазни и излазни токови су репрезентација података који теку кроз систем на вишем нивоу апстракције у декомпозицији.

Трећи ниво декомпозиције дефинише ток података у округлим заградама након имена функције. Улазни подаци су обележени симболом веће (>), излазни подаци су обележени симболом мање (<). Додавањем симбола једнако (=), улазни токови (<=) и излазни токови (=>) су обележени. У првој линији функција плаћања добија улазни ток података и даје излазни ток података. У линији 2 улазни ток података је декомпонован на објекте података који представљају број наруџбине и број кредитне картице. У линији 3 објекат података плаћања је креиран коришћењем објеката података број наруџбине и број кредитне картице. У линији 4 функција за покушај плаћања поседује плаћање као улазни и излазни објекат података што значи да се његова вредност може променити. Покушај плаћања даје статус као резултат. У линији 6 објекат података број наруџбине је део објекта података плаћања и улаз је у функцију. У последњој линији излазни ток је компонован од података статус.

```
1   Payment(=>PaymentInput,<=PaymentOutput)
2           Input(=>PaymentInput,<ordId,<CCNum)
3           CreatePayment(<payment,>ordId,>CCNum)
4           TryPayment(<>payment,<status)
5                   GetOrderSum(>payment,<sum)
6                           GetItemsForOrderId(>payment.ordId,<items)
7                                   GetDatabasePersistenceManager(<dbPersistenceManager)
8                                   GetOrder(>dbPersistenceManager,>ordId,<order)
9                                   GetItems(>order,<order.items)
10                          InitialSum(<sum)
11                          GetSumForItems(>payment,>items,<>sum)
12                                  GetItemPrice(>item,<price)
13                                  GetItemCount(>item,<count)
14                                  IncreaseSum(>payment,>price,>count,<>sum)
15                  Transaction(>payment.CCNum,>sum,<status)
16                  TransactionDone(<>payment,>status)
17                          PaymentSuccesful(<>payment)
18                                  WriteSuccessToLog(>payment)
19                                  ChangeToSuccesful(<>payment)
20                          PaymentFailed(<>payment)
21                                  WriteFailureToLog(>payment)
22                                  ChangeToFailed(<>payment)
23          RecordPayment(>payment)
24                  GetDatabasePersistenceManager(<dbPersistenceManager)
25                  SavePayment(>dbPersistenceManager,>payment)
26                  GetItemsForOrderId(>payment.ordId,<items)
27                  UpdateInventory(>items)
28          Output(>status,<=PaymentOutput)
```

Слика 3 Трећи ниво декомпозиције - Ток података за пример Процес плаћања у продавници.

Четврти ниво декомпозиције представља скуп анотација које омогућавају поновну употребу већ дефинисаних функција (Слика 4). Функција може бити означена за поновну употребу. Друге функције могу бити означене ради употребе функције која је означена за поновну употребу.

Четврти ниво декомпозиције дефинише поновну употребу хеш симболом (#). Функција у линији 6, која дохвата артикле на основу броја наруџбине, означена за поновну употребу и њено име је резервисано писањем хеш симбола (#) као суфикса имена функције. Функција у линији 25 је поновна употреба функције у линији 6 јер је хеш симбол (#) написан као префикс имена функције.

Пети ниво декомпозиције представља скуп анотација који дефинише имплементациони аспект система. Слика 5, поред имплементационог аспекта приказује преостала четири нивоа декомпозиције ради комплетности примера. Имплементациони аспект дефинише извршиоце функција, стања објеката података, типове објеката података, компоненте, чворове, ресурсе и улоге. Извршилац је део система који је одговоран за извршавање функције. Објекти података могу да имају дефинисана стања и транзиције стања. Тип података може бити дефинисан за објекат података, и хијерархија типова података може бити дефинисана. Типови података могу бити груписани у компоненте система. Извршавање функције ради се на чвору. Током извршавања, функција може обавити операцију над ресурсом. Функцији могу бити додељени актери. Актери су ентитети ван система који могу користити функције система.

```
1   Payment
2           Input
3           CreatePayment
4           TryPayment
5                   GetOrderSum
6                           GetItemsForOrderId#
7                                   GetDatabasePersistenceManager
8                                   GetOrder
9                                   GetItems
10                          InitialSum
11                          GetSumForItems
12                                  GetItemPrice
13                                  GetItemCount
14                                  IncreaseSum
15                  Transaction
16                  TransactionDone
17                          PaymentSuccesful
18                                  WriteSuccessToLog
19                                  ChangeToSuccesful
20                          PaymentFailed
21                                  WriteFailureToLog
22                                  ChangeToFailed
23          RecordPayment
24                  GetDatabasePersistenceManager
25                  SavePayment
26                  #GetItemsForOrderId
27                  UpdateInventory
28          Output
```

Слика 4 Четврти ниво декомпозиције - Поновна употреба на примеру Процес плаћања у продавници.

Пети ниво декомпозиције дефинише извршиоца функције (Слика 5). Функција може имати извршиоца који може бити метода класе, метода објекта, или сервис. Извршилац је дефинисан унутар угластих заграда након дефиниције функције. Функција у првој линији се извршава методом payment класе Store. Подразумевано, име методе је исто као име функције. Функција у линији 2 нема дефинисаног извршиоца па се она извршава као део окружујуће методе што је метода payment класе Store. Функција у линији 3 се извршава методом cPayment класе Payment. Функција у линији 4 се извршава методом tryPayment објекта payment. Уколико је извршилац методе објекат, назив извршиоца мора бити исти као назив првог објекта податка функције. Извршилац функције у линији 14 је сервис банке.

Пети ниво декомпозиције дефинише стања података и транзиције између стања. Стање се пише унутар округлих заграда након назива податка. У линији 3 објекат податка payment је у стању Initialized. Након што се трансакција обави, стање објекта податка payment се мења у Successful у линији 18 у случају да је трансакција успешна и објекат податка payment је у стању Initialized. Након што се трансакција обави, стање објекта податка payment се мења у Failed у линији 21 у случају да трансакција није успешна и објекат податка payment је у стању Initialized.

Пети ниво декомпозиције дефинише типове података. Тип може бити класа или примитиван тип. У линији 2 број наруџбине и број кредитне картице су примитивног типа integer. У линији 3 објекат податка за плаћање је класе Payment. У линији 7 објекат податка

```
1  1 Payment(=>PaymentInput,<=PaymentOutput)[C:Store(store.jar:System.Store)]^SystemAdmin@StoreServer:www.store.com
2        1 Input(=>PaymentInput,<ordId:int,<CCNum:int)
3        2 CreatePayment(<payment(Initialized):Payment,>ordId,>CCNum)[C:Payment:cPayment]
4        3 TryPayment(<>payment,<status:int)[O:payment]^FinanceSector@FinanceServer:www.finance.com
5              1 GetOrderSum(>payment,<sum)
6                    1 GetItemsForOrderId#(>payment.ordId,<items)^DBAdmin,DBStoreOwner,DBPrivilegedUser
7                          1 GetDatabasePersistenceManager(<dbPersistenceManager:StoreDatabasePersistenceManager-
   >PersistenceManager(persistence.jar:Persistence))[C:PersistenceManagerFactory]
8                          2 GetOrder(>dbPersistenceManager,>ordId,<order:Order(:Persistence))[O:dbPersistenceManager]
9                          3 GetItems(>order,<order.items:{}Item(persistence.jar:Persistence))
10                   2 InitialSum(<sum)
11                   3* GetSumForItems(>payment,>items,<>sum:int) /item in items
12                         1 GetItemPrice(>item,<price:float)[O:item]
13                         2 GetItemCount(>item,<count:int)[O:item]
14                         3 IncreaseSum(>payment,>price,>count,<>sum)[O:payment]
15             2 Transaction(>payment.CCNum:int,>sum,<status:int)[S:BankService]@BankServer:www.bank.com
16             3 TransactionDone(<>payment,>status)[O:payment(payment.jar:System.Payment)]
17                   1e PaymentSuccesful(<>payment) /status==200 AND (Initialized)
18                         1 WriteSuccessToLog(>payment)[O:payment(payment.jar:System.Payment)]
19                         2 ChangeToSuccesful(<>payment(Succesful))
20                   1e PaymentFailed(<>payment) /status!=200 AND (Initialized)
21                         1 WriteFailureToLog(>payment)[O:payment:writeFailToLog]
22                         2 ChangeToFailed(<>payment(Failed))
23       4? RecordPayment(>payment) /status==200^DBAdmin,DBStoreOwner
24             1 GetDatabasePersistenceManager(<dbPersistenceManager:StoreDatabasePersistenceManager)[C:PersistenceManagerFactory]
25             2p SavePayment(>dbPersistenceManager,>payment)[O:dbPersistenceManager](I-PAYMENT:TABLE@DatabaseServer:www.db.com)
26             2p #GetItemsForOrderId(>payment.ordId:int,<items)
27             3 UpdateInventory(>items)[C:Store]
28       5 Output(>status,<=PaymentOutput)
```

Слика 5 Пети ниво декомпозиције – Имплементациони аспект на примеру Процес плаћања у продавници.

који представља менаџера перзистенције је класе StoreDatabasePersistenceManager који проширује класу PersistenceManager.У линији 9 објекат података који представља артикле је колекција објеката класе Item.

Пети ниво декомпозиције дефинише компоненте. Компоненте су логичке групе класних типова. Физички артефакт може манифестовати логичку компоненту. Након писања назива класе, компонента којој припада може бити дефинисана у округлим заградама. Компонента се дефинише физичким именом и логичким именом компоненте које су одвојене симболом двотачка (:). У првој линији класа Store припада компоненти Store која је физички смештена у фајлу store.jar. Компонента Store је под-компонента компоненте System. У линији 8 класа Order припада компоненти Persistence чије физичко име још увек није дефинисано. У линији 15 класа Payment објекта payment припада компоненти Payment која је физички смештена у фајлу payment.jar. Компонента Payment је под-компонента компоненте System.

Пети ниво декомпозиције дефинише чворове. Чвор је генеричка машина на коју се распоређују артефакти и на којој се извршавају функције. Чвор може имати своју инстанцу која представља постојећу машину. Чвор се дефинише писањем симбола ет (@) који је праћен логичким именом чвора и физичким именом инстанце чвора. Функција за плаћање у првој линији означена је да буде извршена на чвору StoreServer на постојећој машини идентификованом преко веб адресе www.store.com. Функција на линији 4 је означена да буде извршена на чвору FinanceServer на постојећој машини идентификованом веб адресом www.finance.com. Функција на линији 14 означена је да буде извршена на чвору BankServer на постојећој машини идентификованом веб адресом www.bank.com.

Пети ниво декомпозиције дефинише употребу ресурса. Употреба ресурса се дефинише као операција коју функција обавља над ресурсом. Тип ресурса може бити дефинисан за ресурс. Употреба ресурса пише се у витичастим заградама након дефиниције функције. У линији 24 плаћање се евидентира у табели Payment која представља ресурс, користећи SQL операцију уноса insert. Табела Payment налази се на чвору DatabaseServer на постојећој машини са веб адресом www.db.com. Дефиниција чвора на којем се налази ресурс је опциона.

Пети ниво декомпозиције дефинише актере. Актери који интерагују са функцијом могу бити дефинисани писањем посебног симбола (^) који је праћен називима актера. Функција за плаћање на првој линији може експлицитно бити коришћена од стране администратора система. Функција у линији 4 која покушава плаћање може експлицитно бити искоришћена од стране финансијског сектора. Функција која дохвата артикле на линијама 6 и 25 може експлицитно бити коришћена од стране администратора базе података, власника базе података и привилегованог корисника базе података. Евидентирање плаћања у линији 22 може се експлицитно користити од стране администратора базе података и власника базе података. Уколико администратор система експлицитно користи функцију плаћања у првој линији, све под-функције функције плаћања могу бити на тај начин имплицитно искоришћене од стране администратора система.

Све претходно описане анотације су дефинисане у АФД кроз његову бесконтекстну граматику која се састоји од 41 правила. Преглед неких анотација које су коришћене у примеру Процес плаћања у продавници је дат у Табели 1. За сваки ниво декомпозиције, Табела 1 даје анотације и њихове локације у примеру што је пропраћено и кратким објашњењима. Правила и покривеност нивоа декомпозиције правилима су дати у Табели 2.

Табела 1 Преглед неких анотација које су коришћене у примеру Процес за плаћање у продавници.

Ниво декомпозиције	**Анотација**	**Линија у примеру**	**Објашњење**
1. ниво – Декомпозиција	TryPayment	4	Функција.
2. ниво – Ток контроле	5 * ? e p /item in items /status==200	27 10 22 16 24 10 22	Редни број функције. Функција ће бити извршена у петљи. Функција ће бити условно извршена. Функција је експлузивна. Функција ће бити извршена у паралели. Услов петље за извршавање функције. Услов за извршавање функције.
3. ниво – Ток података	>ordId <sum <>sum =>PaymentInput <=PaymentOutput payment.ordId	3 5 10 1 1 6	Улазни податак. Излазни податак. Улазни и излазни податак. Улазни ток података. Излазни ток података. Део података.
4. ниво – Употреба функција	GetItemsForOrderId# #GetItemsForOrderId	6 25	Функција која се може употребити. Употреба функције.
5. ниво – Имплементација	C:Payment:cPayment O:payment:writeFailToLog Initialized Payment payment.jar:System.Payment StoreServer:www.store.com I-PAYMENT:TABLE DBAdmin	3 20 3 3 15 1 24 6	Метода класе која извршава функцију. Метода објекта која извршава функцију. Стање објекта података. Тип података. Физичка и логичка компонента. Чвор и инстанца чвора. Операција над ресурсом. Актер.

#	Правило	1. ниво Декомпозиција	2. ниво Ток контроле	3. ниво Ток података	4. ниво Употреба функција	5. ниво Имплементација
1	Function ::= FunctionDefinition FunctionDecompositionEntry FunctionList FunctionDecompositionExit | FunctionDefinition;	X				
2	FunctionDecompositionEntry ::= INDENT;	X				
3	FunctionDecompositionExit ::= DEDENT;	X				
4	FunctionList ::= FunctionList Function | Function;	X				
5	FunctionDefinition ::= FunctionDefinitionWithoutNewLine NEWLINE;	X				
6	FunctionDefinitionWithoutNewLine ::= FunctionPrefix FunctionName DataFlows ImplementationFlows Resources Condition Roles Node;	X	X	X	X	X
7	FunctionPrefix ::= NUMBER SPACE | EXECUTION_TYPE SPACE | CONDITION_TYPE SPACE | NUMBER EXECUTION_TYPE SPACE | NUMBER CONDITION_TYPE SPACE | EXECUTION_TYPE CONDITION_TYPE SPACE | NUMBER EXECUTION_TYPE CONDITION_TYPE SPACE | ;		X			
8	FunctionName ::= NAME | NAME HASH | HASH NAME;				X	
9	Condition ::= SPACE SLASH BoolExpression | SPACE SLASH LOOP_CONDITION | ;		X			
10	BoolExpression ::= BoolOperand | BoolOperand Separation BOOL_OPERATOR Separation BoolExpression | LBRACE BoolExpression RBRACE | LBRACE BoolExpression RBRACE Separation BOOL_OPERATOR Separation BoolExpression;		X			
11	BoolOperand ::= NAME RelationalOperation NAME | NAME RelationalOperation Constant | Constant RelationalOperation NAME | StateDefinition | Constant;		X			
12	RelationalOperation ::= RELATIONAL_OPERATION | DIRECTION;		X			
13	Separation ::= SPACE Separation | SPACE;					
14	Roles ::= ROLES_START RoleList | ;					X
15	RoleList ::= RoleList COMMA Role | Role;					X

Табела 2 (први део) Нивои декомпозиције покривени правилима АФД језика.

#	Правило	1. ниво Декомпозиција	2. ниво Ток контроле	3. ниво Ток података	4. ниво Употреба функција	5. ниво Имплементација
16	Role ::= NAME;					X
17	Constant ::= NUMBER | STRING_CONSTANT | BOOLEAN_CONSTANT;					
18	DataFlows ::= LBRACE DataFlowList RBRACE | ;			X		
19	ImplementationFlows ::= LSBRACE ImplementationFlowList RSBRACE | ;					X
20	DataFlowList ::= DataFlowList COMMA DataFlow | DataFlow;			X		
21	DataFlowType ::= LCBRACE RCBRACE DataFlowTypeHierarchy Component | DataFlowTypeHierarchy Component;					X
22	DataFlowTypeHierarchy ::= DataFlowTypeElem INHERITANCE LBRACE DataFlowTypeHierarchies RBRACE | DataFlowTypeElem INHERITANCE DataFlowTypeHierarchy | DataFlowTypeElem;					X
23	DataFlowTypeHierarchies ::= DataFlowTypeHierarchies COMMA DataFlowTypeHierarchy | DataFlowTypeHierarchy;					X
24	DataFlowTypeElem ::= NAME;					X
25	DataFlow ::= DIRECTION NAME State | DIRECTION NAME State COLON DataFlowType | DIRECTION NAME State EQUAL Constant | DIRECTION NAME State COLON DataFlowType EQUAL Constant;			X		X
26	State ::= StateDefinition | ;					X
27	StateDefinition ::= LBRACE NAME RBRACE | LBRACE NAME Entry RBRACE | LBRACE NAME Exit RBRACE | LBRACE NAME Entry Exit RBRACE;					X
28	Entry ::= VERTICAL_BAR ENTRY SPACE FunctionDefinitionWithoutNewLine;					X
29	Exit ::= VERTICAL_BAR EXIT SPACE FunctionDefinitionWithoutNewLine;					X
30	ImplementationFlowList ::= ImplementationFlowList COMMA ImplementationFlow | ImplementationFlow;					X

Табела 2 (други део) Нивои декомпозиције покривени правилима АФД језика.

Табела 2 (трећи део) Нивои декомпозиције покривени правилима АФД језика.

#	Правило	1. ниво Декомпозиција	2. ниво Ток контроле	3. ниво Ток података	4. ниво Употреба функција	5. ниво Имплементација
31	ImplementationFlow ::= NAME COLON NAME Component Implementation;					X
32	Implementation ::= COLON NAME | ;					X
33	Component ::= LBRACE ArtifactName COLON ComponentName RBRACE | ;					X
34	ArtifactName ::= NAME | ;					X
35	ComponentName ::= NAME | ;					X
36	Resources ::= LCBRACE ResourceList RCBRACE | ;					X
37	ResourceList ::= ResourceList COMMA Resource | Resource;					X
38	Resource ::= ResourceDefinition Node;					X
39	ResourceDefinition ::= NAME HYPHEN NAME ResourceType | NAME ResourceType;					X
40	ResourceType ::= COLON NAME | ;					X
41	Node ::= AT NAME | AT COLON NAME | AT NAME COLON NAME | ;					X

АФД поред функционалне декомпозиције имплементира и друге методологије ради управљања описима архитектура. У потпоглављу 5.1. је описана имплементација рачунарског размишљања у АФД. У потпоглављу 5.2. је представљено креирање погледа интегралним моделовањем и управљање конзистенцијом у АФД. У потпоглављу 5.3. су приказани и описани алгоритми превођења АФД описа у UML описе архитектура.

5.1. Имплементација рачунарског размишљања у АФД

За решавање комплексних проблема дизајнирања великих софтверских система, АФД се ослања на методолошким концептима уведеним у рачунарском размишљању које бира одговарајућу репрезентацију проблема или моделира релевантне аспекте проблема како би га учинио решивим. Моделирање у контексту рачунарског размишљања подразумева поверење да се сигурно користи, модификује и утиче на велики комплексан систем без разумевања свих детаља, модуларизовање система у очекивању вишеструких употреба или претходног преузимања, и кеширање у очекивању будуће употребе [7]. Да би се постигле ове намере рачунарско размишљање користи четири технике, такође познате као стуб декомпозиције, стуб апстракције, стуб алгоритама и стуб препознавања образаца [8], који су имплементирани у АФД пратећи принципе доброг дизајна [66].

Као што је претходно описано, рачунарско размишљање идентификује комплексан проблем тако што га разлаже у веће и мање под-проблеме којима је лакше управљати. Приступ стуба декомпозиције је инхерентно својство функционалне декомпозиције, тако да се инкорпорира у АФД природно кроз имплементацију као први ниво декомпозиције, што је, поново, декомпозиција комплексне функције у мање функционалне компоненте.

Пратећи стуб апстракције рачунарског размишљања који се фокусира само на најбитније детаље, приликом моделовања проблема и већих под-проблема функцијама, само токови контроле и токови података функција на вишем нивоу су битни. Сходно томе, АФД имплементира овај стуб апстракције кроз други и трећи ниво декомпозиције дефинишући токове контроле и токове података.

Креирање корака или једноставних правила како би се решили мањи под-проблеми кроз стуб алгоритама је примењен над условним извршавањима и извршавањима у петљи унутар функционалне декомпозиције, као и објектима података које користе функције на нижем нивоу. АФД имплементира овај стуб алгоритама такође кроз његов други и трећи ниво декомпозиције.

Коначно, стуб препознавања образаца који разматра решења сличних проблема за решавање идентификованих већих или мањих под-проблема се види у декомпозицији функције која је поново искоришћена као део декомпозиције друге функције. Дакле, АФД имплементира овај стуб препознавања образаца у оквиру четвртог нивоа декомпозиције кроз поновну употребу функције. На овај начин АФД имплементира све принципе рачунарског размишљања [10].

5.2. Интегрално моделовање и управљање конзистенцијом у АФД

Оно што карактерише АФД је могућност за моделовање комплексних софтверских система његовим декомпоновањем кроз пет нивоа декомпозиције. Сваки ниво декомпозиције представљен је одређеним АФД анотацијама. У декомпозицији све анотације се пишу једна уз другу и на тај начин креирају интегрални модел читавог софтверског система. Имајући интегрални модел који садржи све информације о систему не ограничава управљање архитектуре јер и даље подржава раздвајање интереса везаних за систем јер одређени нивои декомпозиције могу бити мапирани у одговарајућа гледишта која су идентификована у литератури.

Током година различити скупови гледишта су уведени ради моделовања комплексних система. 1995. године Soni и остали [17] увели су гледишта Концепт, Модул, Извршавање и Код. Овај скуп гледишта представља први скуп гледишта из литературе. Гледиште Концепт описује систем у погледу његових главних елемената дизајна и односа између њих. Гледиште Модул обухвата две ортогоналне структуре: функционалну декомпозицију и слојеве. Гледиште Извршавање описује динамичку структуру система. Гледиште Код описује како су изворни код, бинарни фајлови и библиотеке организовани у оквиру развојног окружења.

1995. године Kruchtenr и остали [18] увели су четири главна гледишта: Логичко, Процес, Физичко, Развој и једно описно гледиште: Сценарији. Овај скуп гледишта представља други скуп гледишта из литературе. Логичко гледиште описује дизајн система кроз модел објеката када се користи објектно оријентисана метода. Како би се дизајнирао систем који је доста заснован на подацима, могу се користити алтернативни приступи ради дизајнирања логичког погледа, као што је дијаграм ентитета и односа. Гледиште Процес описује конкурентне и синхронизационе аспекте дизајна. Физичко гледиште описује мапирање софтвера на хардвер и тако одражава дистрибуирани аспект. Гледиште Развој описује статичку организацију софтвера у развојном окружењу. Гледиште Сценарији показује како елементи из четири гледишта раде заједно. Гледиште Сценарији представљају апстракцију најважнијих захтева.

2002. године Clements и остали [19] увели су 17 гледишта који су подељени у категорије: стилови модула, стилови компоненти и конектора, стилови алокација, хибридни стилови. Овај скуп гледишта представља трећи скуп гледишта из литературе. Модули су примарни елементи стилова модула. Модул представља имплементациону јединицу која пружа повезан скуп одговорности. Стилови компоненти и конектора изражавају извршно понашање. Они су описани на основу компоненти и конектора. Компонента је један од главних јединица за процесирање у оквиру извршног система. Стилови алокација описују мапирање софтверских јединица на елементе окружења у оквиру којих се софтвер развија или извршава. Окружење може бити хардвер, фајл систем који даје подршку за развој или распоређивање или организације за развој.

2002. године Garland и остали [20] увели су 14 гледишта који су подељени у категорије: концептуална и аналитичка гледишта, гледишта логичког дизајна и физичка гледишта. Овај скуп гледишта представља четврти скуп гледишта из литературе. Концептуална и аналитичка гледишта пружају сажетак граница система и спољашњих ентитета који интерагују са системом као и апстрактни скуп ентитета фокусираних на моделовање проблема пре него на решење. Гледишта логичког дизајна пружају мапирање логичких извршних структура, њихове функционалности, и њихове међусобне комуникације, визуализацију зависности између подсистема и интерфејса, апстрактни поглед на све подсистеме и опис модела података дељених између компоненти. Физичка гледишта пружају приказују мапирање хардвера и софтвера дистрибуираних система, илуструју структуре физичког распоређивања база података, приказују извршне нити система и често мапирање на компоненте и приказују динамичка стања процеса.

2005. године Rozanski и остали [13] увели су гледишта Контекст, Функционалности, Информације, Конкурентност, Развој, Распоређивање и Операције. Овај скуп гледишта представља пети скуп гледишта из литературе. Гледиште Контекст описује односе, зависности и интеракције између система и окружења. Гледиште Функционалности описује системске

извршне функционалне елементе, њихове одговорности, интерфејсе и главне интеракције. Гледиште Информације описује на који начин систем складишти, манипулише, одржава и дистрибуира информације. Гледиште Конкурентност описује конкурентну структуру система и мапира функционалне елементе на конкурентне јединице како би јасно идентификовао делове система који могу да буду извршени конкурентно и како се то координише и контролише. Гледиште Развој описује архитектуру која подржава процес развоја софтвера. Гледиште Распоређивање описује окружење у које ће систем бити распоређен и зависности које систем има од његових елемената. Гледиште Операције описује како ће се системом управљати, администрирати и одржавати када ради у свом производном окружењу.

2009. године Taylor и остали [21] увели су гледишта Логичко, Физичко, Развој, Конкурентност и Понашање. Овај скуп гледишта представља шести скуп гледишта из литературе. Логичко гледиште обухвата логичке ентитете система и како су они међусобно повезани. Физичко гледиште обухвата физичке ентитете система и како су они међусобно повезани. Гледиште Развој показује како су физички ентитети мапирани на физичке ентитете. Гледиште Конкурентност показује како се управља конкурентношћу и нитима у систему. Гледиште Понашање обухвата очекивано понашање система.

2018. године Ozkaya [2] је увео гледишта Логичко, Информације, Физичко, Распоређивање, Понашање, Конкурентност, Развој и Операције. Овај скуп гледишта представља седми скуп гледишта из литературе. Логичко гледиште описује логичке компоненте и конекторе који заједно чине софтверски систем. Гледиште Информације описује како компоненте складиште, мењају и размењују њихове податке једне са другима. Физичко гледиште описује хардверске компоненте које су потребне ради спецификације уређаја у које ће софтверске компоненте бити лоциране и односа између тих уређаја. Гледиште Распоређивање описује мапирање између хардверских и логичких компоненти. Гледиште Понашање описује понашања системских компоненти и сложене механизме интеракције. Гледиште Конкурентност описује проблеме које се односе на конкурентност као што су нити и синхронизација између компоненти. Гледиште Развој описује како ће компоненте и конектори бити имплементирани и тестирани. Гледиште Операције описује како ће специфицирани систем бити покретан у свом окружењу, укључујући и планове за инсталацију система, конфигурацију, подршку за кориснике, надгледање система као и планове за креирање резервних копија и враћања система у претходно стање у случају квара.

Скупови гледишта у литератури се разликују једни од других у броју, именовању и значењу гледишта. Са друге стране, постојећи стандард не прописује фиксан скуп гледишта, па у сврху демонстрације коришћења гледишта у АФД, пример скупа гледишта садржи следећа гледишта: Функционално, Извршавање, Информације, Имплементација, Стање података, Компоненте, Распоређивање, Контекст и Ресурси. Гледиште Функционално описује систем у смислу његових основних функција и односа између њих. Гледиште Извршавање описује ток контроле система. Гледиште Информације описује податке, односе између података и ток података. Гледиште Имплементација описује типове података, односе између типова података, и делове система који извршавају функције. Гледиште Стање података описује стање података и транзиције између стања. Гледиште Компоненте описује систем као скуп логичких компоненти и њихових физичких манифестација. Гледиште Распоређивање дефинише како су софтверске компоненте распоређене на хардвер. Гледиште Контекст дефинише актере који су

изван система и како актери интерагују са системом кроз случајеве употребе. Гледиште Ресурси дефинише операције на логичким и физичким ресурсима које систем користи, а такође се може користити за описивање нефункционалних захтева [67,68] или за представљање ресурса који су потребни за процесе развоја и операција. Мапирање између примера скупа гледишта и гледишта у литератури је приказано у Табели 3. Сваки скуп гледишта из Табеле 3 бави се сличним проблемима везаним за систем.

Уведена гледишта су мапирана на нивое декомпозиције и АФД анотације које су приказане у Табели 4. Постојећи односи између нивоа декомпозиције су разлог за постојање односа између гледишта и одговарајућих погледа. Креирање интегралног модела почиње са функцијама па се тако креира поглед Функционално. Остали погледи се креирају касније, и

Табела 3 Мапирање примера гледишта на гледишта из литературе.

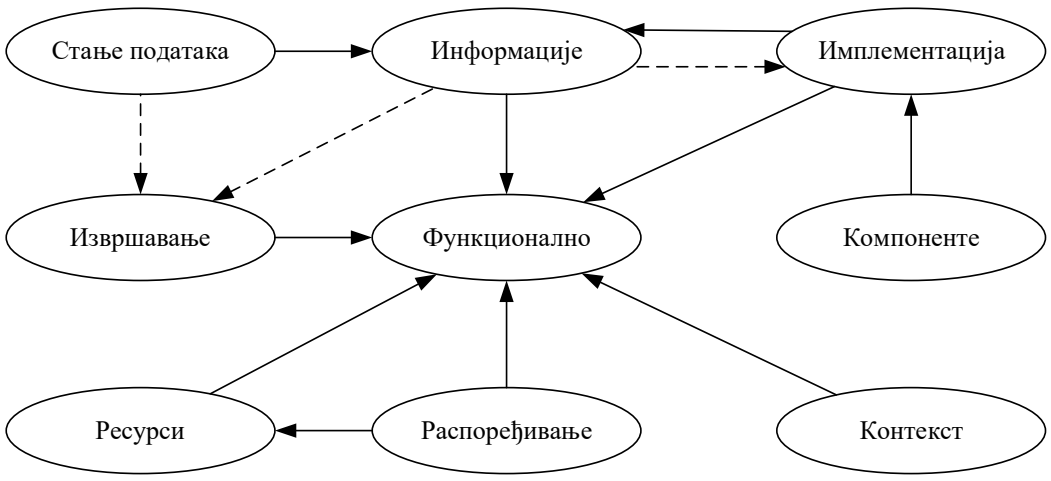
Скуп гледишта #	Гледишта у литератури	Функционално	Извршавање	Информације	Имплементација	Стање података	Компоненте	Распоређивање	Контекст	Ресурси
1	Концепт								X	
	Модул	X								
	Извршавање		X	X	X	X				
	Код						X	X		X
2	Логичко	X		X	X	X				X
	Процес		X							
	Физичко						X	X		X
	Развој				X					X
	Сценарији								X	
3	Стилови модула	X		X					X	X
	Стилови компоненти и конектора		X	X			X			X
	Стилови алокација				X	X	X	X		X
	Хибридни стилови	X	X	X	X	X	X	X	X	X
4	Концептуална и аналитичка гледишта								X	
	Гледишта логичког дизајна	X	X	X	X	X	X			X
	Физичка гледишта						X	X		X
5	Контекст								X	
	Функционалности	X								
	Информације			X	X	X				X
	Конкурентност		X							
	Развој		X	X	X	X	X			X
	Распоређивање						X	X		X
	Операције									X
6	Логичко	X	X	X	X	X	X		X	X
	Физичко							X		X
	Развој						X	X		X
	Конкурентност		X							
	Понашање		X	X	X	X				
7	Логичко	X					X		X	
	Информације			X		X				
	Физичко							X		
	Распоређивање							X		
	Понашање		X							
	Конкурентност		X							
	Развој				X					X
	Операције									X

они увек директно или индиректно анотирају поглед Функционално као што је приказано на Слици 6. Подаци представљени погледом Информације могу бити део услова који је дефинисан у погледу Извршавање. Поглед Имплементација анотира поглед Информације, међутим подаци представљени погледом Информација могу бити део извршиоца који је дефинисан погледом Имплементација. Поглед Стање података анотира поглед Информације и може представљати стања која могу бити део услова који је дефинисан погледом Извршавање. Поглед Компоненте анотира поглед Имплементација, и поглед Распоређивање анотира поглед Ресурси.

Креирање модела као скупа погледа може да направи проблеме конзистентности, уколико правила дефинисана за гледишта нису задовољена. Дакле, настаје потреба за управљањем конзистенцијом. Antonio и остали, анализирали су 40 студија и дефинисали таксономију за карактеризацију решења за моделовање кроз више погледа [16] и навели су

Табела 4 Мапирање примера гледишта на нивое декомпозиције и на АФД анотације.

Примери гледишта	**1. ниво - Декомпозиција**	**2. ниво - Ток контроле**	**3. ниво - Ток података**	**4. ниво - Употреба функција**	**5. ниво - Извршавање**	**5. ниво - Стања**	**5. ниво - Типови**	**5. ниво - Компоненте**	**5. ниво - Чворови**	**5. ниво - Употреба ресурса**	**5. ниво - Актери**
Функционално	X			X							
Извршавање		X									
Информације			X								
Имплементација					X		X				
Стање података						X					
Компоненте								X			
Распоређивање									X		
Контекст											X
Ресурси										X	



Слика 6 Односи између гледишта. Пуна стрелица – однос анотације, испрекидана стрелица – „може бити део“ однос.

активности које се односе на управљање конзистенцијом као што су: спецификација преклапања, детекција неконзистентности и разрешавање неконзистентности. Још шири скуп активности наведен је у оквиру које су дефинисали Spanoudakis и остали [62] који се поред спецификације и примене политике управљања конзистенцијом састоји од активности: детекција преклапања, детекција неконзистентности, дијагноза неконзистентности, руковање неконзистентностима, и праћење неконзистентности. Пружањем интегралног моделовања АФД олакшава активности повезане са детекцијом преклапања, детекцијом неконзистентности, и руковањем неконзистентностима и користи неке од техника и метода које су споменули Spanoudakis и Antonio. Дијагноза неконзистентности и праћење неконзистентности није подржано у АФД.

Детекција преклапања врши се идентификовањем преклапања између погледа. Детекција неконзистентности се врши провером кршења правила конзистентности од стране погледа. Дијагноза неконзистентности бави се идентификацијом извора, узрока и утицаја неконзистентности. Руковање неконзистентностима бави се идентификацијом могућих акција ради разрешавања неконзистентности, проценом ризика и бенефита акција, проценом ризика који би настали уколико се не би разрешиле неконзистентности и избором акција за извршавање. Праћење неконзистентности бави се праћењем онога што се догодило у процесу. Спецификација и примена политике управљања неконзистентностима дефинише агента који би требало да буде искоришћен да идентификује преклапања између погледа, правила конзистентности које треба проверити над погледима, околности које треба да покрену детекцију преклапања и неконзистентности, механизме који треба да буду коришћени ради дијагнозе неконзистентности и околности које треба да покрену ову активност, механизме које треба користити ради процене утицаја неконзистентности и околности које треба да покрену ову активност, механизме које треба користити ради процењивања цене, бенефита и ризика везаних за различите опције руковања неконзистентностима и заинтересоване стране које би биле одговорне за руковање неконзистентностима [1].

Детекција преклапања у АФД ради се коришћењем технике Анализа сличности. Према Spanoudakis анализа сличности се врши аутоматским упоређивањем између погледа. Елементи у АФД су описани углавном њиховим називом, па се анализа сличности врши упоређивањем назива елемената у погледима који припадају гледиштима који могу имати елементе који се преклапају. Примери гледишта који могу имати елементе који се преклапају дати су у Табели 5. Елементи који се преклапају су последица односа „може бити део“ између гледишта као што је приказано на Слици 6. Према Antonio, АФД пружа имплицитну детекцију преклапања која се успоставља коришћењем конвенција као што су именовања, идентификатори и остало.

Детекција неконзистентности у АФД врши се провером испуњености одређених правила конзистентности, односно коришћењем Специјалних облика анализе. Spanoudakis је идентификовао категорије правила конзистентности која се могу дефинисати језиком и која могу, између осталих, бити у категорији правила добре формираности или у категорији правила идентитета описа. Правила у категорији добре формираности су правила која морају бити задовољена погледима како би они били легитимни погледи у језику у којем су дефинисани. АФД дефинише правила добре формираности за свако гледиште као скуп семантичких провера. Пример правила конзистентности за гледиште Функционално је: „Функција која се употребљава мора претходно бити дефинисана као функција која се може

Табела 5 Парови гледишта која могу имати елементе који се преклапају.

	Функционално	**Извршавање**	**Информације**	**Имплементација**	**Компоненте**	**Стање података**	**Контекст**	**Ресурси**	**Распоређивање**
Функционално	/	НЕ	НЕ	НЕ	НЕ	НЕ	НЕ	НЕ	НЕ
Извршавање		/	ДА	НЕ	НЕ	ДА	НЕ	НЕ	НЕ
Информације			/	ДА	НЕ	НЕ	НЕ	НЕ	НЕ
Имплементација				/	НЕ	НЕ	НЕ	НЕ	НЕ
Компоненте					/	НЕ	НЕ	НЕ	НЕ
Стање података						/	НЕ	НЕ	НЕ
Контекст							/	НЕ	НЕ
Ресурси								/	НЕ
Распоређивање									/

поново употребити. Употреба функције означена је знаком # пре назива функције.“. Пример правила конзистентности за гледиште Имплементација је: „Не сме да постоји циклус у наслеђивању класа.“. Како се у АФД погледи пишу једни уз друге, правила конзистентности у категорији која дефинише добру формираност су такође дефинисана над комбинацијама погледа. Пример правила конзистентности за комбинацију погледа Функционално и Извршавање је: „Функција мора имати редни број у префиксу.“. Пример правила конзистентности за комбинацију погледа Функционално и Информације је: „Функција која користи другу функцију мора имати исти број објеката података као функција која се користи.“. Правила конзистентности у категорији опис идентитета су правила која захтевају да различити елементи погледа који се преклапају, имају исте описе. У случају АФД, описи елемената који се преклапају су њихова имена, дакле уколико се два елемента преклапају њихови описи су исти. Оваква врста правила су увек задовољена, па нису експлицитно дефинисана. Правила конзистентности наведена су касније. Према Antonio, АФД пружа аутоматску детекцију неконзистентности оперативном семантиком. Аутоматску детекцију неконзистентности АФД пружа аутоматским механизмима који интерпретирају преклапања и детектују неконзистентности. Оперативна семантика су алгоритми који се користе за детекцију неконзистентности.

Руковање неконзистентностима у АФД ради се синоптичком техником. Према Spanoudakis, синоптичка техника представља укључивање заинтересованих страна у генерисање решења за руковање неконзистентностима. Неконзистентности, у погледима креираним у АФД, могу бити означене како би их заинтересоване стране разрешиле. Дакле, разрешавање неконзистентности је мануелан процес који раде заинтересоване стране. Према Antonio, АФД пружа мануелно разрешавање неконзистентности, које се делегира заинтересованим странама.

Правила конзистентности која постоје у АФД дата су у наставку. Правила су нумерисана и груписана су по гледиштима и комбинацијама гледишта. За одређена гледишта не постоје специфична правила.

Правила за гледиште Функционално:

1. У декомпозицији може постојати највише једна функција која нема над-функцију.
2. Функција која се може поново употребити има знак # после свог назива и мора имати јединствен назив у декомпозицији, у смислу да на другим местима може постојати само функција која је употребљава.
3. Функција која се употребљава мора претходно бити дефинисана као функција која се може поново употребити. Употреба функције означена је знаком # пре назива функције.

Правила за гледиште Имплементација:

4. У хијерархији класа не може да се нађе примитивни тип.
5. Не сме да постоји циклус у наслеђивању класа.
6. Примитивни типови могу бити: int, float, bool. Класни типови су остали типови који нису примитивни.
7. Тип податка не може бити дефинисан над током података.

Правила за гледиште Компоненте:

8. Компонента може бити непосредни део највише једне компоненте.

Правила за гледиште Стање података:

9. Не могу се користити називи стања Initial и Final.
10. Стање не може да буде дефинисано над примитивним податком или над током података.

Правила за комбинацију гледишта Функционално и Извршавање:

11. Функција мора имати редни број у префиксу.
12. Непосредно пре или после функције која је означена за паралелно извршавање мора постојати бар једна функција која је исто увучена и означена за паралелно извршавање и има исти редни број.
13. Непосредно пре или после функције која је означена за ексклузивно извршавање мора постојати бар једна функција која је исто увучена и која је означена за ексклузивно извршавање и има исти редни број.
14. Функција која је означена за секвенцијално извршавање мора имати редни број који је за један већи од редног броја функције која јој претходи и исто је увучена. У случају да функција нема функцију која је исто је увучена и претходи јој, њен редни број мора бити 1.
15. Функција може имати само једну од ознака: e, p, *, ?.
16. Функција која је означена са * мора имати дефинисан услов за петљу. Важи и обрнуто.
17. Функција која је означена са ? или e мора имати дефинисан логички услов за извршавање. Важи и обрнуто.

Правила за комбинацију гледишта Функционално и Информације:

18. Функција која користи другу функцију мора имати исти број објеката података као функција која се користи.
19. Смерови објеката података у функцији која користи другу функцију се морају поклопити са смеровима објеката података у функцији која се користи.
20. Уколико је објекат података улазни у функцију он мора бити излазни објекат података из функције која је исто увучена и која јој претходи, улазни објекат података у првој над-функцији, или итератор у услову за петљу прве над-функције посматране функције уколико она има дефинисан услов за петљу.
21. Уколико је објекат података излазни из функције он мора бити излазни податак из неке од његових директних под-функција уколико оне постоје.
22. Токови података функције која нема над-функцију морају бити такви да је прво наведен улазни ток, а други наведен излазни ток података и то морају бити једини токови података ове функције.
23. Прва под-функција функције која нема над-функцију мора имати наведен прво улазни ток података, а у наставку могу бити наведени само излазни објекти података.
24. Последња под-функција функције која нема над-функцију може имати наведене прво улазне објекте података, док последње наведен мора бити излазни ток података.
25. Уколико је ток података улазни у функцију он мора бити излазни ток података из функције која је исто увучена и која јој претходи, или улазни ток података у првој над-функцији. Ово правило не важи за функцију која нема над-функцију.
26. Уколико је ток података излазни из функције он мора бити излазни ток података из неке од његових директних под-функција уколико оне постоје.

Правила за комбинацију гледишта Функционално и Имплементација:

27. Функција која нема над-функцију мора имати дефинисаног извршиоца који је статичка метода класе.

Правила за комбинацију гледишта Информације и Имплементација:

28. Објекат података мора имати дефинисан тачно један тип података.

Правила за комбинацију гледишта Функционално, Извршавање и Информације:

29. Ако је објекат података део логичког услова функције, тада тај објекат података мора бити излазни објекат података из функције која је исто увучена и која јој претходи, улазни објекат података у првој над-функцији, или итератор у услову за петљу прве над-функције посматране функције уколико она има дефинисан услов за петљу.

Правила за комбинацију гледишта Функционално, Информације и Имплементација:

30. Објекат података који представља колекцију над којим се итерира у оквиру услова за петљу функције мора бити улазни објекат података у ту функцију.
31. Функција чији је извршилац метода објекта мора имати први улазни објекат података који је и објекат методе извршиоца.

Правила за комбинацију гледишта Извршавање, Информације и Имплементација:

32. Објекат података чија је метода извршилац мора имати дефинисан тачно један тип података који није колекција.
33. Објекат података над којим се итерира мора имати дефинисан тачно један тип података који је колекција података одређеног типа.
34. Објекат података који је итератор мора имати дефинисан тачно један тип који је исти као тип елемента колекције над којом се итерира. Имплицитно је дефинисано типом објекта података над којим се итерира.
35. Објекат података који је операнд у логичком изразу за извршавање мора имати дефинисан тачно један тип података.
36. У логичком услову операнди за бинарне релационе операције (==, !=, <>, >=, >, <=, <=) морају бити истог типа. Операнд може бити објекат података или константа. Константе постоје за тип int и то су цели бројеви (пример: 20), за тип float и то су разломљени бројеви (пример: 20.5 или 20), за тип bool и то су TRUE или FALSE , а за класу String константа се наводи под наводницима (примери: “200“ или “tekst“).

Правила за комбинацију гледишта Функционално, Извршавање, Информације, Имплементација и Компоненте:

37. Уколико се у оквиру једне функције безусловно извршава друга функција, не може се посматрана функција безусловно извршавати у оквиру друге функције.

5.3. Алгоритми превођења из АФД у UML

Опис архитектуре који је у конзистентном стању, може бити преведен у UML дијаграме, а UML је широко признат и коришћен језик за описивање дизајна и архитектуре софтверских система. Превођење АФД модела у UML моделе је приступ интегрисања АФД у рану фазу процеса развоја софтвера. АФД алат пружа превођење у седам UML дијаграма који највише одговарају АФД језику: дијаграм класа, компоненти, распоређивања, активности, секвенце,

Табела 6 Мапирање примера гледишта на UML дијаграме.

	Класе	**Компоненте**	**Распоређивање**	**Активности**	**Секвенце**	**Стања**	**Случајеви коришћења**
Функционално	X	X	X	X	X	X	X
Извршавање				X	X	X	X
Информације	X	X	X	X	X	X	
Имплементација	X	X	X		X	X	
Компоненте		X	X				
Стање података						X	
Контекст							X
Ресурси			X				
Распоређивање			X				

стања и случајева коришћења. UML дијаграми класа, компоненти и распоређивања описују структуру система, док дијаграми активности, секвенце, стања и случајева коришћења описују понашање система. Превођење из АФД у UML ради се превођењем одређеног броја АФД погледа у UML дијаграме, односно нису сви погледи неопходни ради превођења у одређени UML дијаграм. Табела 6 приказује која гледишта се користе у превођењу у одређени тип UML дијаграма.

UML дијаграм класа генерише се на основу гледишта Функционално, Информације и Имплементација. Псеудокод алгоритма за генерисање дијаграма класа на основу описа архитектуре писаног у АФД је приказан на Слици 7. На почетку процеса генерисања функција на највишем нивоу описа архитектуре се процесира извршавањем процедуре BuildClassDiagram. Уколико је дефинисан извршилац за функцију закључује се назив типа за извршиоца. Уколико је извршилац објекат података, назив типа може бити дефинисан у његовом изворном објекту података, па се у том случају његов тип мора закључити. Уколико је закључени тип класа, тада се класа са пронађеним називом додаје на дијаграм класа. Сваки објекат података у оквиру функције се може састојати од делова који су одвојени симболом „.“. Називи типова се закључују за сваки део објекта података. Уколико је закључени тип дела класа, она се додаје на дијаграм класа. Зависности се додају од класе извршиоца ка класама делова објеката података. Сваки део података десно од „.“ додаје се као поље класе објекта података лево од „.“. Тип додатог поља је тип дела објекта података. Асоцијације се додају од сваке класе објекта података лево од „.“ Ка класи објекта података десно од „.“. Хијерархија наслеђивања класа додаје се на дијаграм за сваку хијерархију типова дефинисану над објектима података. Метода се додаје класи која је извршилац функције. Метода је статичка уколико је извршилац класа. Метода није статичка уколико је извршилац објекат. Назив методе може се експлицитно дефинисати након назива извршиоца. Уколико се назив методе не дефинише експлицитно, назив методе је исти као назив функције. Аргументи методе су улазни подаци функције. Повратни тип методе је тип излазног података функције. За сваку под-функцију функције понавља се иста процедура, односно нови елементи могу бити додати на дијаграм класа за остале функције описа архитектуре. Остали детаљи алгоритма могу се пронаћи на Слици 7.

UML дијаграм класа генерисан коришћењем описаног алгоритма над описом архитектуре система за плаћање у продавници, приказан је на Слици 8. Статичке методе су подвучене на генерисаном дијаграму класа. Поред поља и метода класа, на дијаграму су приказане зависности између класа као и наслеђивање класа.

UML дијаграм компоненти генерише се на основу гледишта Функционално, Информације, Имплементација и Компоненте. Псеудокод алгоритма за генерисање дијаграма компоненти за опис архитектуре написан у АФД је дат на Слици 9. Компонента може бити дефинисана за класу извршиоца и за класу података као пар назив компоненте и назив артефакта који представља манифестацију компоненте. Назив компоненте може имати префикс који се састоји од назива над-компоненти којима припада одвојених симболом „.“. Компонента десно од „.“ је под-компонента компоненте лево од „.“. На почетку процеса генерисања процесира се функција на највишем нивоу извршавањем процедуре BuildComponentDiagram. Уколико је класа дефинисана као тип извршиоца функције или као тип података функције, за који је дефинисана компонента, процесира се назив компоненте. За сваки назив над-компоненте који се налази у префиксу назива компоненте, додаје се компонента на дијаграм уколико претходно

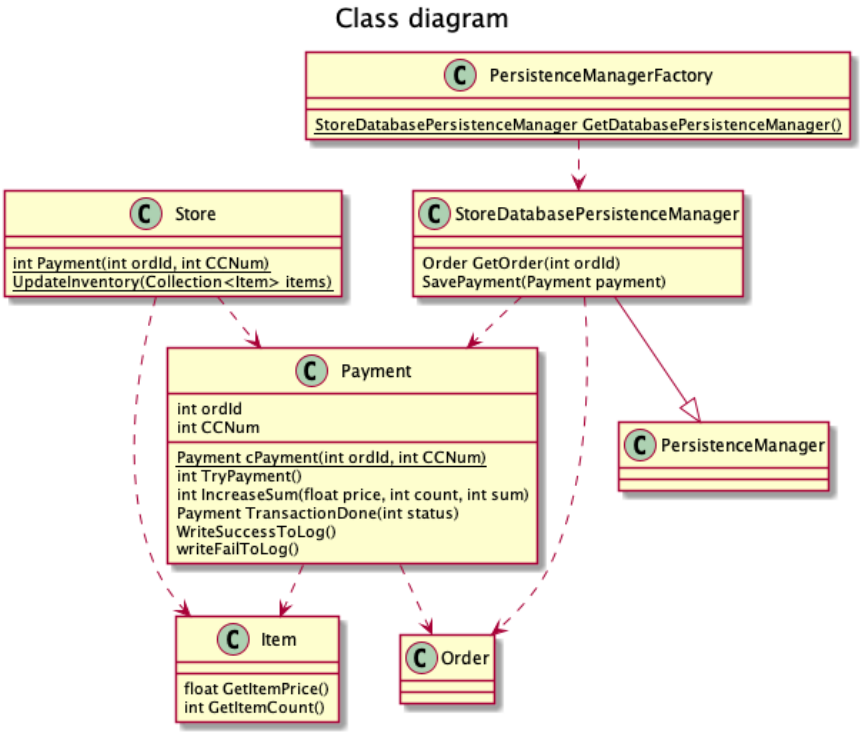
Algorithm Build Class Diagram - Part 1

```
1: procedure BuildClassDiagram(functionality)
2:     BuildFunctionality(functionality)

3: procedure BuildFunctionality(functionality)
4:     if ExecutorExists(functionality) = true then
5:         executorClassName ← ConcludeClassName(Executor(functionality))
6:         executorClass ← AddClass(executorClassName)
7:     for each dataFlow in DataFlows(functionality) do
8:         dataFlowPartClass ← null
9:         for each dataFlowPart in DataFlowParts(dataFlow) do
10:            previousDataFlowPartClass ← dataFlowPartClass
11:            dataFlowPartClassName ← ConcludeClassName(dataFlowPart)
12:            if dataFlowPartClassName = null then
13:                continue
14:            dataFlowPartClass ← AddClass(dataFlowPartClassName)
15:            if executorClassName ≠ null AND dataFlowPartClassName ≠ null AND dataFlowPartClassName ≠ executorClassName then
16:                AddDependency(executorClassName, dataFlowPartClassName)
17:            if previousClass ≠ null then
18:                fieldName ← Name(dataFlowPart)
19:                AddField(previousClass, fieldName)
20:                if classNode ≠ null then
21:                    AddAssociation(previousDataFlowPartClass, dataFlowPartClass)
22:        dataFlowClassName ← ConcludeClassName(dataFlow)
23:        AddSuperClasses(dataFlowClassName)
24:    if ExecutorExists(functionality) = true then
25:        AddMethod(functionality)
26:    for each subFunctionality in GetSubfunctionalities(functionality) do
27:        BuildFunctionality(subFunctionality)
```

Algorithm Build Class Diagram - Part 2

```
28: procedure AddMethod(functionality)
29:     class ← GetClass(functionality)
30:     isStatic ← ExecutorIsClass(Executor(functionality))
31:     methodName ← MethodName(Executor(functionality))
32:     method ← AddMethod(class, methodName, isStatic, returnClass)
33:     for each dataFlow in DataFlows(functionality) do
34:         if IsInput(dataFlow) then
35:             AddArgument(method,dataFlow)
36:         if IsOutput(dataFlow) then
37:             AddReturnType(method,dataFlow)

38: procedure AddSuperClasses(dataFlowClassName)
39:     for each inheritedClassName in InheritedDataFlowClassNames(dataFlowClassName) do
40:         class ← AddClass(dataFlowClassName)
41:         superClass ← AddClass(inheritedClassName)
42:         AddSuperClass(class, superClass)
43:         AddSuperClasses(inheritedClassName)
```

Слика 7 Псеудокод алгоритма за генерисање UML дијаграма класа.



Слика 8 Генерисани UML дијаграм класа за пример Процес плаћања у продавници.

није додата. У случају додавања под-компоненте на дијаграм, под-компонента се додаје унутар над-компоненте. Артефакт се додаје на дијаграм као манифестација компоненте. У случају да за класу назив компоненте није експлицитно дефинисан, назив компоненте се закључује за класу и компонента се додаје на дијаграм. Уколико је компонента дефинисана за класу извршиоца и за класу података функције додаје се зависност од компоненте извршиоца ка компоненти података функције. Уколико је компонента дефинисана за класу која је извршилац над-функције као и за класу која је извршилац функције, додаје се зависност од компоненте дефинисане за класу извршиоца над-функције до компоненте дефинисане за класу извршиоца функције. Под-функције функције процесирају се на исти начин. Овим се завршава креирање дијаграма компонената. Остали детаљи везани за алгоритам могу се пронаћи на Слици 9.

UML дијаграм компонената, генерисан коришћењем описаног алгоритма за опис архитектуре система за плаћање у продавници, приказан је на Слици 10. Компонента System је написана као префикс компоненте Payment, дакле компонента Payment се на дијаграму налази у оквиру компоненте System. Над класом извршиоца функције GetDatabasePersistenceManager дефинисана је компонента Persistence, док је над класом извршиоца функције Payment која је над-функција функције GetDatabasePersistenceManager дефинисана компонента Store. Према томе, зависност је додата од компоненте Store ка компоненти Persistence. Артефакт store.jar је дефинисан за компоненту Store и одговарајућа манифестација је додата на дијаграм.

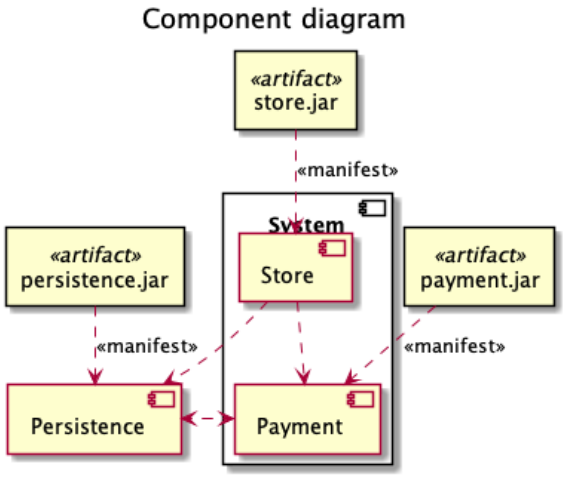
UML дијаграм распоређивања заснован је на гледиштима Функционално, Информације, Имплементација, Компоненте, Ресурси и Распоређивања. Псеудокод алгоритма за генерисање дијаграма распоређивања за опис архитектуре написан у АФД је дат на Слици 11. Чвор може бити дефинисан за одређену функцију или ресурс. Дефиниција чвора састоји се од назива чвора и назива појаве чвора. Уколико је први чвор дефинисан за одређену функцију, а други чвор дефинисан за њену над-функцију, први и други чвор међусобно комуницирају. Уколико је први чвор дефинисан за ресурс који користи одређена функција, а други чвор за ту функцију,

први и други чвор међусобно комуницирају. Уколико је први чвор дефинисан за ресурс одређене функције, а други чвор дефинисан за над-функцију посматране функције, први и други чвор међусобно комуницирају. Уколико је први чвор дефинисан за одређену функцију, сви артефакти дефинисани за извршиоца и податке те функције и свих функција ниже у хијерархији до оне функције за коју је дефинисан други чвор, се распоређују на први чвор. У оквиру процеса генеришу се два дијаграма, први за чворове и њихове појаве и други за комуникацију између чворова и распоређивање артефаката. На почетку процеса генерисања функција на највишем нивоу декомпозиције процесира се извршавањем процедуре BuildDeploymentDiagram. Уколико функција има дефинисан назив чвора, чвор се додаје на први дијаграм. Уколико функција има дефинисан назив појаве чвора, нова појава чвора се

Algorithm Build Component Diagram

```
1: procedure BuildComponentDiagram(functionality)
2:     BuildFunctionality(functionality)

3: procedure BuildFunctionality(functionality)
4:     executor ← Executor(functionality)
5:     executorComponent ← null
6:     if executor ≠ null then
7:         if ComponentDefined(executor) then
8:             componentPartNames ← GetComponentPartNames(executor)
9:             artifactName ← GetArtifactName(executor)
10:            executorComponent ← AddComponentsAndArtifact(componentPartNames, artifactName)
11:        else
12:            componentName ← ConcludeComponentName(executor)
13:            executorComponent ← AddComponent(componentName)
14:    for each dataFlow in DataFlows(functionality) do
15:        dataFlowComponent ← null
16:        if ComponentDefined(dataFlow) then
17:            componentPartNames ← GetComponentPartNames(dataFlow)
18:            artifactName ← GetArtifactName(dataFlow)
19:            dataFlowComponent ← AddComponentsAndArtifact(componentPartNames, artifactName)
20:        else
21:            componentName ← ConcludeComponentName(dataFlow)
22:            dataFlowComponent ← AddComponent(componentName)
23:        if executorComponent ≠ null AND dataFlowComponent ≠ null AND executorComponent ≠ dataFlowComponent then
24:            AddDependency(executorComponent, dataFlowComponent)
25:    if executorComponent ≠ null then
26:        executorComponentInSuperFunc ← GetExecutorComponentInSuperFunc(functionality)
27:        if executorComponentInSuperFunc ≠ null AND executorComponentInSuperFunc ≠ executorComponent then
28:            AddDependency(executorComponentInSuperFunc, executorComponent)
29:    for each subFunctionality in GetSubfunctionalities(functionality) do
30:        BuildFunctionality(subFunctionality)
```

Слика 9 Псеудокод алгоритма за генерисање UML дијаграма компоненти.



Слика 10 Генерисани UML дијаграм компоненти за пример Процес плаћања у продавници.

додаје на први дијаграм. Уколико су дефинисани називи чвора и појаве чвора, појава чвора се повезује са чвором на првом дијаграму. Иста додавања се врше у случају да су чвор или појава чвора дефинисани за ресурс. Уколико се у опису архитектуре пронађе комуникација између два чвора, чворови и комуникација између њих се додају на други дијаграм. Сваки пронађен артефакт, чвор и распоређивање артефакта на чвор у опису архитектуре, додају се на други дијаграм. Процесирање под-функција врши се на исти начин. Овим се завршава креирање UML дијаграма распоређивања. Остали детаљи алгоритма могу се наћи на Слици 11.

Два UML дијаграма распоређивања, генерисана користећи описане алгоритме за опис архитектуре система за плаћање у продавници, приказани су на Слици 12. Чворови и појаве чворова додају се на први дијаграм, док се комуникације између чворова и распоређивање артефаката додају на други дијаграм. Чвор StoreServer и појава чвора www.store.server су дефинисани за функцију Payment, па су чвор и појава чвора додати на први дијаграм. Чвор FinanceServer је дефинисан за функцију TryPayment. Функција TryPayment је ниже у хијерархији у односу на функцију Payment, с тога је на други дијаграм додата комуникација између чворова FinanceServer и StoreServer. Артефакт persistence.jar је дефинисан за класе PersistenceManager и Item. Објекат dbPersistenceManager класе PersistenceManager се користи у функцији GetDatabasePersistenceManager која је ниже у хијерархији у односу на функцију TryPayment над којом је дефинисан чвор FinanceServer. Према томе, артефакт persistence.jar је распоређен на чвор FinanceServer што је и додатно на други дијаграм. Колекција артикала, која се састоји од објеката класе Item, користи се у оквиру функције UpdateInventory која је ниже у хијерархији од функције Payment над којом је дефинисан чвор StoreServer. Дакле, артефакт persistence.jar је распоређен на чвор StoreServer што је и додато на други дијаграм.

UML дијаграми активности генеришу се на основу гледишта Функционално, Извршавања и Информације. Псеудокод алгоритма за генерисање дијаграма активности за опис архитектуре написан у АФД је дат на Слици 13. На почетку процеса генерисања функција на највишем нивоу описа архитектуре се процесира извршавањем процедуре BuildActivityDiagram. На почетку креирања дијаграма активности иницијални и крајњи елемент се додају на дијаграм. Између иницијалног и крајњег елемента одређена додавања на дијаграм се врше за функцију. Уколико је функција означена за извршавање у паралели елемент за паралелно рачвање се додаје. Уколико се функција извршава у петљи или се условно извршава елемент за рачвање се додаје. Уколико је извршилац дефинисан за функцију и ако

функција има под-функцију, додаје се активност на дијаграм и креира се нови дијаграм активности за ту функцију. Уколико извршилац није дефинисан за функцију и функција нема под-функцију, додаје се акција на дијаграм. Уколико је функција означена за поновну употребу додаје се активност на дијаграм и креира се нови дијаграм активности за функцију. Уколико функција представља поновну употребу функције активност се додаје на дијаграм. У осталим случајевима ни акција ни активност се не додају на дијаграм за функцију. Након што се заврше додавања на дијаграм за одређену функцију, под-функције се процесирају на исти начин. Након што се обави процесирање под-функција финална додавања се врше за функцију. Уколико функција представља петљу, петља се на дијаграму затвара додавањем контроле тока ка чвору

Algorithm Build Deployment Diagram - Part 1

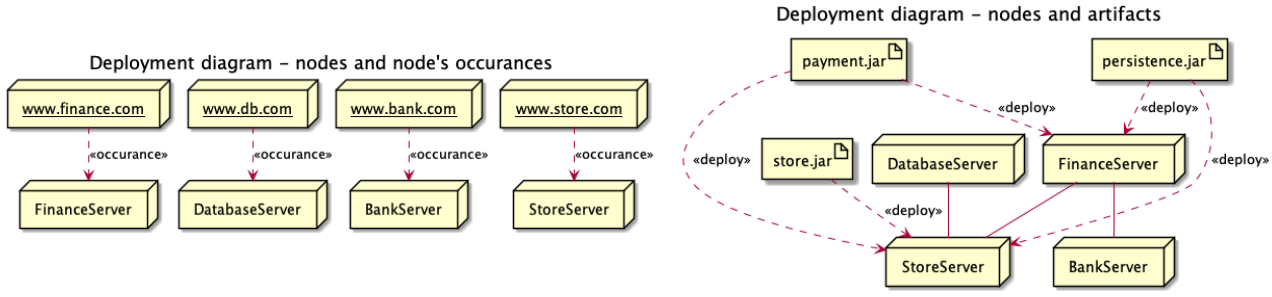
```
1:  procedure BuildDeploymentDiagram(functionality)
2:      BuildFunctionality(functionality)

3:  procedure BuildFunctionality(functionality)
4:      nodeName ← LogicalNodeName(functionality)
5:      nodeOccuranceName ← PhysicalNodeName(functionality)
6:      node ← null
7:      if nodeName ≠ null then
8:          node ← AddNode(nodeName)
9:      if nodeOccuranceName ≠ null then
10:         nodeOccurance ← AddNodeOccurance(nodeOccuranceName)
11:     nodeInSuperFunc ← GetNodeInSuperFunc(functionality)
12:     if nodeName ≠ null then
13:         if nodeOccuranceName ≠ null then
14:             AddOccurance(node, nodeOccurance)
15:         artifacts ← GetArtifactsUntilNextNode(functionality)
16:         for each artifact in artifacts do
17:             AddArtifactDeployment(node, artifact)
18:         if nodeInSuperFunc ≠ null AND nodeInSuperFunc ≠ node then
19:             AddCommunication(node, nodeInSuperFunc)
20:     nodesForResources ← GetNodesForResources(functionality)
21:     for each nodeForR in nodesForResources do
22:         nodeNameForResource ← LogicalNodeName(nodeForR)
23:         nodeOccuranceNameForResource ← PhysicalNode-
    Name(nodeForR)
24:         nodeForResource ← null
25:         if nodeNameForResource ≠ null then
26:             nodeForResource ← AddNode(nodeNameForResource)
27:             if nodeOccuranceNameForResource ≠ null then
28:                 nodeOccuranceForResource ← AddNodeOccu-
    rance(nodeOccuranceNameForResource)
29:                 AddOccurance(nodeForResource,
    nodeOccuranceForResource)
30:         if nodeName ≠ null then
31:             AddCommunication(node, nodeForResource)
32:         else if nodeInSuperFunc ≠ null then
33:             AddCommunication(node, nodeInSuperFunc)
34:     for each subFunctionality in GetSubfunctionali-
    ties(functionality) do
35:         BuildFunctionality(subFunctionality)
```

Слика 11 (први део) Псеудокод за генерисање UML дијаграма распоређивања.

Algorithm Build Deployment Diagram - Part 2

```
36: procedure GetArtifactsUntilNextNode(functionality)
37:     nodeName ← LogicalNodeName(functionality)
38:     artifacts ← []
39:     GetArtifactsUntilNextNode(functionality, artifacts, nodeName, false)
40:     return artifacts

41: procedure GetArtifactsUntilNextNode(functionality, artifacts, currentNodeName, stop)
42:     executor ← Executor(functionality)
43:     if executor ≠ null then
44:         artifactName ← ConcludeArtifactName(executor)
45:         if artifactName ≠ null then
46:             artifact ← AddArtifact(artifactName)
47:             AddNewArtifactToSet(artifacts, artifact)
48:     for each dataFlow in DataFlows(functionality) do
49:         artifactName ← ConcludeArtifactName(dataFlow)
50:         if artifactName ≠ null then
51:             artifact ← AddArtifact(artifactName)
52:             AddNewArtifactToSet(artifacts, artifact)
53:     if stop = false then
54:         for each subFunctionality in GetSubfunctionalities(functionality) do
55:             nodeName ← LogicalNodeName(functionality)
56:             if nodeName = null OR nodeName = currentNodeName then
57:                 GetArtifactsUntilNextNode(subFunctionality, artifacts, currentNodeName, false)
58:             else if nodeName ≠ null AND nodeName ≠ currentNodeName then
59:                 GetArtifactsUntilNextNode(subFunctionality, artifacts, currentNodeName, true)
```

Слика 11 (други део) Псеудокод за генерисање UML дијаграма распоређивања.



Слика 12 Генерисани UML дијаграми распоређивања за пример Процес плаћања у продавници.

за рачвање који је претходно био додат. Уколико се функција извршава условно, елемент за спајање се додаје. Уколико је функција означена за извршавање у паралели елемент за паралелно спајање се додаје. Остали детаљи алгоритма се могу пронаћи на Слици 13.

UML дијаграми активности генерисани користећи описани алгоритам за опис архитектуре система за плаћање у продавници, су приказани на Слици 14. На дијаграмима активности, акције су приказане жутом бојом, док су активности приказане светло плавом бојом. Функције TryPayment и TransactionDone имају извршиоца и под-функцију па су оне представљене активностима и одговарајући дијаграми активности су креирани. Функција

Algorithm Build Activity Diagram - Part 1

```
1: procedure BuildActivityDiagram(functionality)
2:     AddBegin
3:     BuildFunctionality(functionality)
4:     AddEnd

5: procedure BuildFunctionality(functionality)
6:     BuildOnFunctionalityEnter(functionality)
7:     for each subFunctionality in SubFunctionalities(functionality) do
8:         BuildFunctionality(subFunctionality)
9:     BuildOnFunctionalityExit(functionality)
```

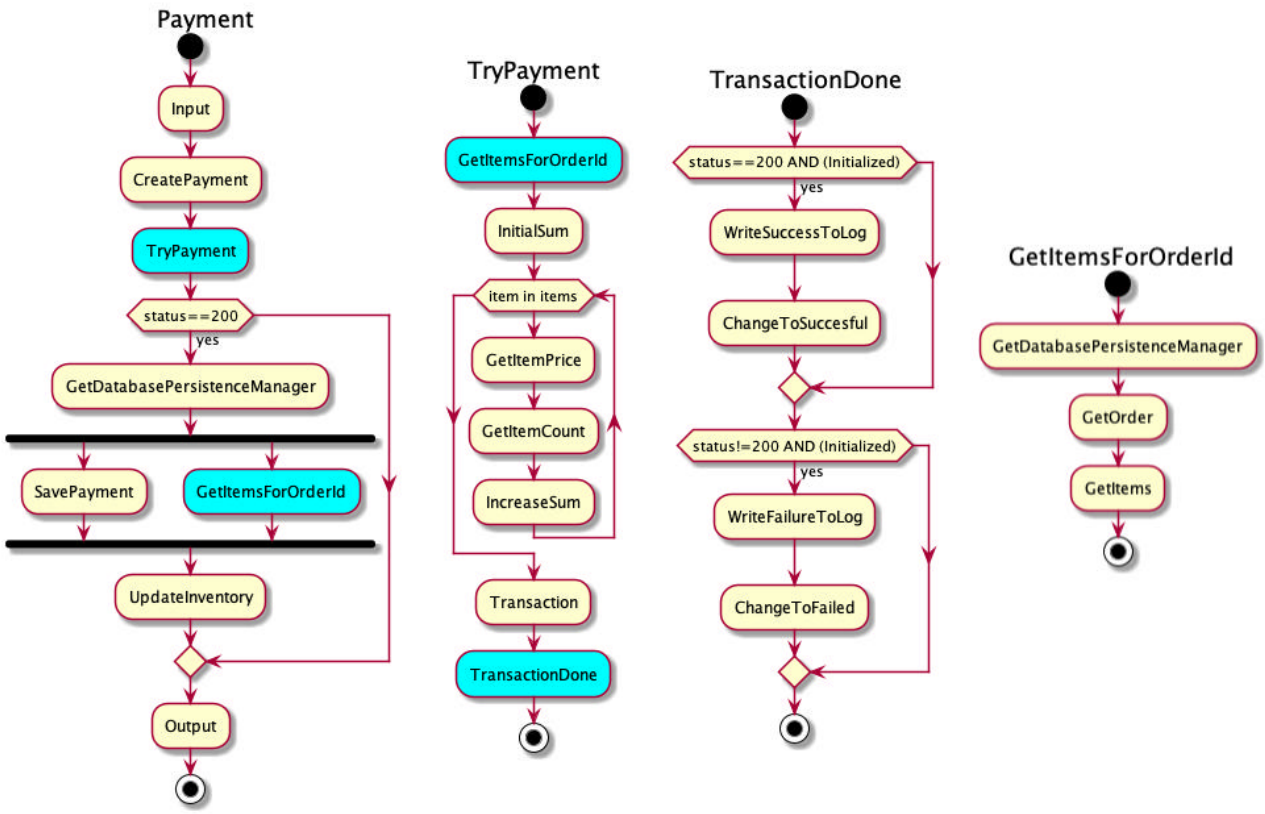
Algorithm Build Activity Diagram - Part 2

```
10: procedure BuildOnFunctionalityEnter(functionality)
11:     if Parallel(functionality) = true then
12:         OpenFork
13:     if Loop(functionality) = true then
14:         OpenLoop(functionality)
15:         AddActivityOrAction(functionality, "loop")
16:         if ExecutorExists(functionality) = true AND SubfunctionalityExists(functionality) = true then
17:             BuildActivityDiagram(functionality)
18:     else if Optional(functionality) = true then
19:         OpenIf(functionality)
20:         AddActivityOrAction(functionality, "optional")
21:         if ExecutorExists(functionality) = true AND SubfunctionalityExists(functionality) = true then
22:             BuildActivityDiagram(functionality)
23:     else if Standard(functionality) = true then
24:         if Reusable(functionality) = true then
25:             AddActivityOrAction(functionality)
26:             if Referenced(functionality) = true AND OneReusableFunctionalityWithSubfunctionalitiesExists(functionality) = true then
27:                 BuildActivityDiagram(functionality)
28:         else
29:             if ExecutorExists(functionality) = false AND SubfunctionalityExists(functionality) = false then
30:                 return
31:             AddActivityOrAction(functionality, "standard")
```

Algorithm Build Activity Diagram - Part 3

```
32: procedure BuildOnFunctionalityExit(functionality)
33:     if Loop(functionality) then
34:         CloseLoop
35:     else if Optional(functionality) then
36:         CloseIf
37:     else if Standard(functionality) then
38:         if ExecutorExists(functionality) = false AND SubfunctionalityExists(functionality) = false then
39:             return
40:     if Parallel(functionality) then
41:         CloseFork
```

Слика 13 Псеудокод алгоритма за генерисање UML дијаграма активности.



Слика 14 Генерисани UML дијаграми активности за пример Процес плаћања у продавници.

GetItemsForOrderId је означена за поновну употребу, па је представљена активношћу и одговарајућим дијаграмом активности. Дакле, Слика 14 приказује дијаграме активности Payment, TryPayment, TransactionDone и GetItemsForOrderId.

UML дијаграми секвенце генеришу се на основу гледишта Функционално, Информације и Имплементације. Псеудокод алгоритма за генерисање дијаграма секвенце за опис архитектуре написан у АФД је дат на Слици 15. На почетку генерисања процесира се функција на највишем нивоу описа архитектуре извршавањем процедуре BuildSequenceDiagram. Животне линије User и I/O се додају на дијаграм и додаје се порука од User ка I/O животној линији, па се свака функција процесира на исти начин. У случају да функција представља методу класе која није редефинисана ни у једној од класа у хијерархији извршава се регуларна процедура додавања елемената на дијаграм, у супротном се извршава специјална процедура.

У оквиру регуларног процесирања функција, уколико се функција извршава у паралели, на дијаграму се отвара фрагмент за паралелно извршавање уколико фрагмент не постоји или паралелни операнд у оквиру фрагмента уколико фрагмент постоји. У случају да је дефинисан извршилац за функцију, функција има под-функцију, или је функција означена за поновну употребу, тада се отвара фрагмент за петљу уколико се функција извршава у петљи, или се отвара опциони фрагмент уколико се функција опционо извршава. Уколико је извршилац дефинисан за функцију на дијаграм се додаје животна линија. Назив додате животне линије је назив класе извршиоца функције. Ако функција није означена за поновну употребу додаје се нова порука од животне линије претходно процесиране функције до животне линије посматране функције. Ако је функција означена за поновну употребу фрагмент референцирања

се додаје на животну линију функције и додаје се нова порука од животне линије претходно процесиране функције до животне линије посматране функције. Порука представља позив методе класе или објекта који је дефинисан извршиоцем и функционалношћу. Уколико функција која је означена за поновну употребу има под-функције, нови дијаграм секвенце се креира за функцију и додавање нових елемената за под-функцију се не обавља на текућем дијаграму. Након што се додавања на дијаграм обаве за посматрану функцију, под-функције се процесирају на исти начин. Када се процесирање под-функција заврши, финална додавања се врше на дијаграму за функцију. Додаје се порука од животне линије функције до животне линије претходно процесиране функције. Порука представља повратни тип методе класе или објекта која је представљена функционалношћу и њеним извршиоцем. У случају да је дефинисан извршилац за функцију, функција има под-функцију, или је функција означена за поновну употребу, тада се затвара фрагмент за петљу уколико се функција извршава у петљи, или се затвара опциони фрагмент уколико се функција опционо извршава. Уколико је функција означена за паралелно извршавање и она је на њеном нивоу у опису архитектуре последња тако означена функција затвара се фрагмент за паралелно извршавање.

Специјална процедура се извршава у случају да функција представља методу класе или објекта коју редефинише нека од класа у хијерархији. Уколико функција представља методу коју класа извршиоца те функције не редефинише, креира се нови дијаграм секвенце. Новокреирани дијаграм секвенце описује позив посебне методе која на основу стварног типа позиваоца позива одређену дефиницију методе у хијерархији. На новом дијаграму додаје се

Algorithm Build Sequence Diagrams - Part 1

```
1:  procedure BuildSequenceDiagrams(topLevelFunctionality)
2:      BuildSequenceDiagram(topLevelFunctionality)
3:      for each functionality in GetPolymorphFunctionalities do
4:          EndBuildPolymorph(functionality)

5:  procedure BuildSequenceDiagram(functionality)
6:      if MethodIsRedifined(functionality) = false then
7:          BuildRegularSequenceDiagram(functionality)
8:      else
9:          BuildStubSequenceDiagram(functionality)
10:         BuildPolymorph(functionality)

11: procedure BuildRegularSequenceDiagram(functionality)
12:     diagram ← BeginDiagram
13:     BuildFunctionality(diagram, functionality, functionality)
14:     EndDiagram(diagram)

15: procedure BuildFunctionality(diagram, functionality,
    currentTopFunctionality)
16:     BuildOnFunctionalityEnter(diagram, functionality,
    currentTopFunctionality)
17:     if Reusable(functionality) = false OR functionality =
    currentTopFunctionality then
18:         for each subFunctionality in SubFunctionalities(functionality)
    do
19:             BuildFunctionality(diagram, subFunctionality,
    currentTopFunctionality)
20:     BuildOnFunctionalityExit(diagram,functionality,
    currentTopFunctionality)
```

Слика 15 (први део) Псеудокод алгоритма за генерисање UML дијаграма секвенце.

Algorithm Build Sequence Diagrams - Part 2

```
21: procedure BuildOnFunctionalityEnter(diagram, functionality, currentTopFunctionality)
22:     if Parallel(functionality) = true then
23:         previous ← PreviousFunctionality(functionality)
24:         if previous ≠ null AND Parallel(previous) = true then
25:             AddParallelOperand(diagram)
26:         else
27:             OpenParallelFragment(diagram)
28:     if (executor ≠ null OR SubfunctionalityExists(functionality) = true OR Reusable(functionality) = true) AND functionality ≠ currentTopFunctionality then
29:         if Loop(functionality) = true then
30:             OpenLoopFragment(diagram)
31:         else if Optional(functionality) = true then
32:             OpenOptFragment(diagram)
33:         BuildMessageOnFunctionalityEnter(diagram,functionality,currentTopFunctionality)

34: procedure BuildOnFunctionalityExit(diagram, functionality, currentTopFunctionality)
35:     BuildMessageOnFunctionalityExit(diagram, functionality, currentTopFunctionality)
36:     if (executor ≠ null OR SubfunctionalityExists(functionality) = true OR Reusable(functionality) = true) AND functionality ≠ currentTopFunctionality then
37:         if CallLoopfunctionality = true OR Optional(functionality) = true then
38:             CloseFragment(diagram)
39:         if Parallel(functionality) = true then
40:             succeeding ← SucceedingFunctionality(functionality)
41:             if succeeding = null OR Parallel(succeeding) = false then
42:                 CloseFragment(diagram)
```

Algorithm Build Sequence Diagrams - Part 3

```
43: procedure BuildStubSequenceDiagram(functionality)
44:     diagram ← BeginDiagram
45:     className ← ConcludeClassName(Executor(functionality))
46:     class ← GetClass(className)
47:     classNameTop ← GetName( GetTopSuperClassWithMethod(functionality))
48:     if className = classNameTop then
49:         return
50:     sendMessage ← CreateSendMessage(functionality)
51:     replyMessage ← CreateReplyMessage(functionality)
52:     lifelineForClass ← GetLifeline(functionality)
53:     lifelineForTopClass ← GetLifelineForTopSuperClassWithMethod(functionality)
54:     AddFoundMessage(diagram, lifelineForClass, sendMessage)
55:     AddMessage(diagram, lifelineForClass, lifelineForTopClass, sendMessage)
56:     AddMessageReply(diagram, lifelineForTopClass, lifelineForClass, replyMessage)
57:     AddLostMessage(diagram, lifelineForClass, replyMessage)
58:     EndDiagram(diagram)
```

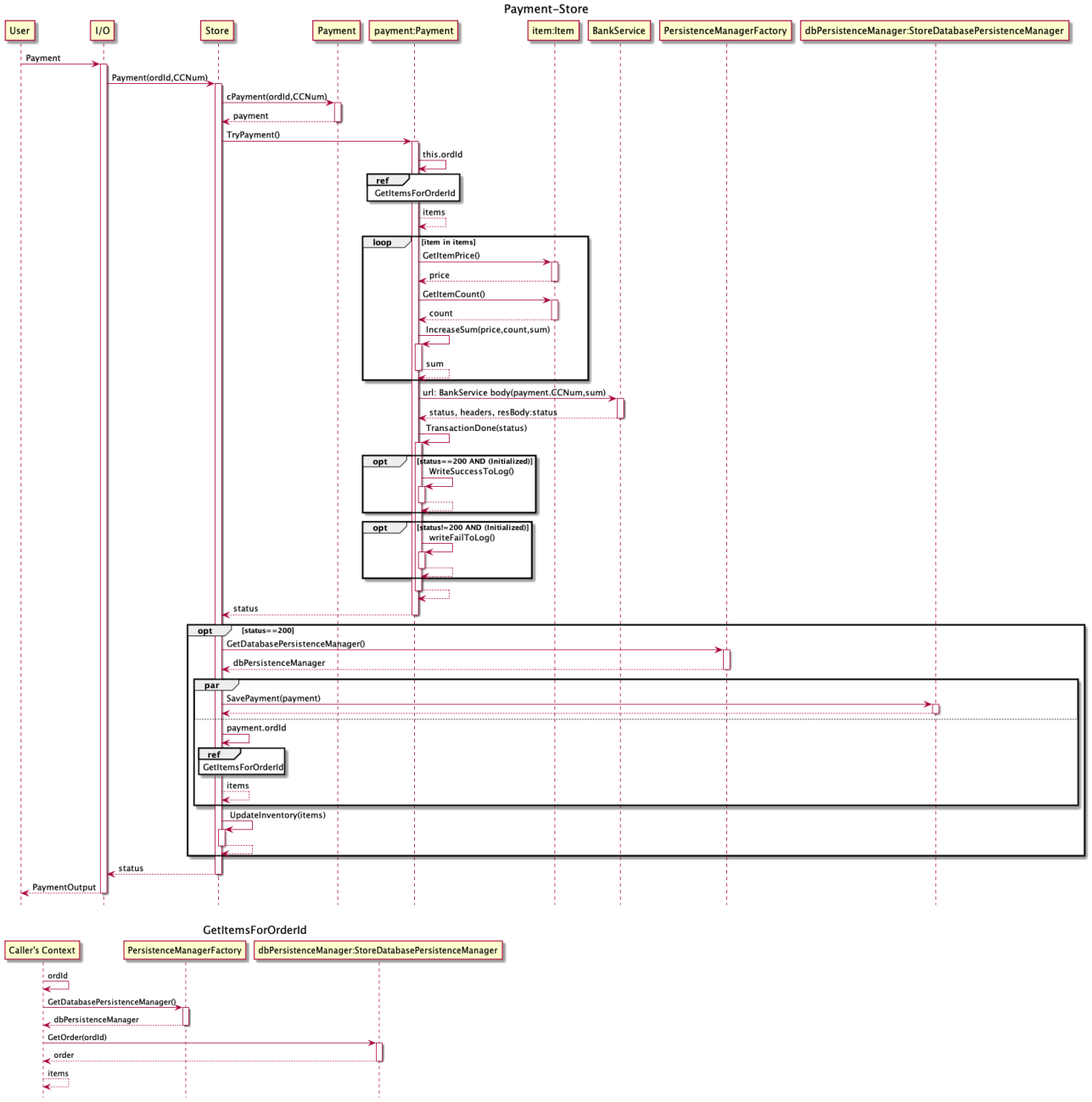
Слика 15 (други део) Псеудокод алгоритма за генерисање UML дијаграма секвенце.

Algorithm Build Sequence Diagrams - Part 4

```
59: procedure BuildPolymorph(functionality)
60:     classes ← GetClassesThatContainAndDoNotRedefineMethod(functionality)
61:     conditionInstanceOf ← ""
62:     for each class in classes do
63:         AddToCondition(conditionInstanceOf,class)
64:     if PolymorphDiagramExists(functionality) = false then
65:         diagram ← BeginDiagram(functionality)
66:         lifelineForTopClass ← GetLifelineForTopSuperClassWithMethod(functionality)
67:         sendMessage ← CreateSendMessage(functionality)
68:         AddFoundMessage(diagram, lifelineForTopClass, sendMessage)
69:         OpenAltFragment(diagram,functionality)
70:         AddAlternative(diagram,functionality,conditionInstanceOf)
71:     else
72:         AddAlternative(diagram,functionality,conditionInstanceOf)
73:     BuildFunctionality(diagram,functionality,functionality)

74: procedure EndBuildPolymorph(functionality)
75:     diagram ← GetDiagram(functionality)
76:     CloseFragment(diagram)
77:     lifelineForTopClass ← GetLifelineForTopSuperClassWithMethod(functionality)
78:     replyMessage ← CreateReplyMessage(functionality)
79:     AddLostMessage(diagram,lifelineForTopClass,replyMessage)
80:     EndDiagram(diagram)
```

Слика 15 (трећи део) Псеудокод алгоритма за генерисање UML дијаграма секвенце.

животна линија за извршиоца функције, као и животна линија класе највише у хијерархији која је редефинисала ту методу. Додаје се пронађена порука ка животној линији класе извршиоца функције. Пронађена порука представља позив методе која је представљена функционалношћу. Порука се додаје од животне линије класе извршиоца функције до животне линије класе највише у хијерархији која је редефинисала ту методу. Порука представља позив редефинисане методе. Додају се одговори на додате поруке, чиме је новокреирани дијаграм употпуњен. Поред новокреираног дијаграма, све методе са истим потписом у хијерархији представљене су једним заједничким дијаграмом који креира процедура BuildPolymorph. Заједнички дијаграм се састоји од оквира алтернативе, где свака алтернатива представља различиту редефиницију методе. Услов за алтернативу је да је објекат инстанца одређене класе у хијерархији. На овај начин заједнички дијаграм представља полиморфно извршавање методе. Прво се проналазе све класе ниже у хијерархији од класе извршиоца функције и заједно представљају услов за алтернативу. Уколико је заједнички дијаграм већ направљен, нови фрагмент алтернативе се додаје у заједнички дијаграм са дефинисаним условом, а функција се процесира регуларном процедуром. У супротном се креира заједнички дијаграм, на дијаграм се додаје линија живота за класу која је редефинисала методу а налази се у навише у хијерархији. Додаје се пронађена метода ка претходно додатој линији живота. Отвара се фрагмент алтернативе, нова алтернатива се додаје на заједнички дијаграм и функција се процесира регуларном процедуром. Када се све функције описа архитектуре процесирају, финалне акције се раде за сваки заједнички дијаграм. Фрагменти алтернативе се затварају, изгубљене поруке се додају од животне линије у заједничком дијаграму и тиме се заједнички дијаграми комплетирају. Након комплетирања заједничких дијаграма порука одговора се



Слика 16 Генерисани UML дијаграми секвенце за пример Процес плаћања у продавници.

додаје од животне линије I/O ка животној линији User. Ово закључује креирање дијаграма секвенце. Остали детаљи алгоритма могу се пронаћи на Слици 15.

UML дијаграми секвенце, генерисани користећи описани алгоритам за опис архитектуре система за плаћање у продавници, су приказани на Слици 16. Први дијаграм је генерисан на основу свих функција за опис архитектуре система за плаћање у продавници, осим за под-функције функције GetItemsForOrderId. Други дијаграм је генерисан на основу функције GetItemsForOrderId која је означена за поновну употребу, као и њених под-функција.

UML дијаграм стања генерише се на основу гледишта Функционално, Извршавање, Имплементација и Стање података. Псеудокод алгоритма за генерисање дијаграма стања за опис архитектуре написан у АФД дат је на Слици 17. UML дијаграм стања се састоји од

елемената стања и транзиција између стања. Генерисање дијаграма стања заснива се на дефинисаним стањима за објекте података и на дефинисаним транзицијама у оквиру машина стања. Уколико је стање дефинисано за објекат података, генерише се дијаграм стања за класу објекта података уколико не постоји. Функција у оквиру описа архитектуре може представљати машину стања уколико задовољава два услова. Први услов је да функција мора да представља методу објекта класе, односно мора имати дефинисаног извршиоца који је објекат података, као и да први податак у оквиру функције мора бити исти улазни објекат података. Други услов је да под-функције посматране функције морају да дефинишу транзиције између стања. Функција представља транзицију између стања уколико је део услова за њено извршавање стање тог

Algorithm Build State Diagrams - Part 1

```
1: procedure BuildStateDiagrams(topLevelFunctionality)
2:     BuildFunctionality(topLevelFunctionality)
3:     for each diagram in GetStateMachineDiagrams do
4:         BuildTransitionsToFinalState(diagram)
```

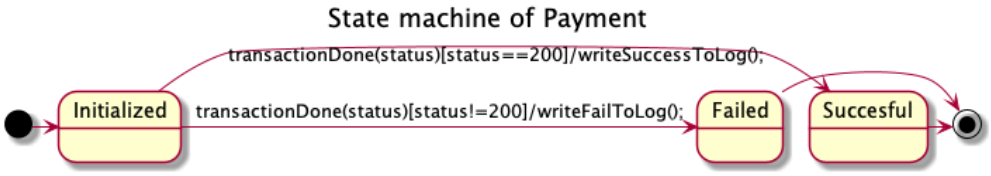
Algorithm Build State Diagrams - Part 2

```
5: procedure BuildFunctionality(functionality)
6:     dataFlowWithState ← GetDataFlowWithState(functionality)
7:     superFunctionality ← SuperFunctionality(functionality)
8:     if HasStateInCondition(functionality) = false AND
       dataFlowWithState ≠ null AND IsOutput(dataFlowWithState) =
       true then
9:         stateMachineDiagram ← GetStateMachineDia-
       gram(dataFlowWithState)
10:        if NOT Initialized(stateMachineDiagram) then
11:            AddInitialAndFinalStates(stateMachineDiagram)
12:        stateName ← GetStateName(dataFlowWithState)
13:        newState ← AddState(stateMachineDiagram, stateName)
14:        initialState ← GetInitialState(stateMachineDiagram)
15:        AddTransition(stateMachineDiagram, initialState, newState)
16:    else if HasStateInCondition(functionality) = true AND Is-
       StateMachineFunctionality(superFunctionality) then
17:        firstDataFlow ← FirstDataFlow(functionality)
18:        stateMachineDiagram ← GetStateMachineDia-
       gram(dataFlowWithState)
19:        if NOT Initialized(stateMachineDiagram) then
20:            AddInitialAndFinalStates(stateMachineDiagram)
21:        stateNameFrom ← GetStateNameFromCondi-
       tion(functionality)
22:        stateNameTo ← GetStateNameFromLastSubfunctional-
       ity(functionality)
23:        event ← CreateTransitionEvent(superFunctionality)
24:        condition ← GetConditionWithoutStates(functionality)
25:        actions ← GetTransitionActions(functionality)
26:        stateFrom ← AddState(stateMachineDiagram,
       stateNameFrom)
27:        stateTo ← AddState(stateMachineDiagram, stateNameTo)
28:        AddTransition(stateMachineDiagram, stateNameFrom,
       stateNameTo, event, condition, actions)
29:    for each subFunctionality in SubFunctionalities(functionality) do
30:        BuildFunctionality(subFunctionality)
```

Слика 17 (први део) Псеудокод алгоритма за генерисање UML дијаграма стања.

```
Algorithm Build State Diagrams - Part 3
31: function GetTransitionActions(functionality)
32:     actions ← ∅
33:     for each subFunctionality in SubFunctionalities(functionality) do
34:         executor ← Executor(subFunctionality)
35:         if executor ≠ null then
36:             if NOT HasStateInDataflows(subFunctionality) then
37:                 action ← CreateAction(subFunctionality)
38:                 AddAction(actions, action)
39:     return actions

40: procedure BuildTransitionsToFinalState(stateMachineDiagram)
41:     finalState ← GetFinalState(stateMachineDiagram)
42:     for each state in GetStatesWithoutOutputTransitions(stateMachineDiagram) do
43:         AddTransition(state, finalState)
```

Слика 17 (други део) Псеудокод алгоритма за генерисање UML дијаграма стања.



Слика 18 Генерисани UML дијаграм стања за пример Процес плаћања у продавници.

податка, које уједно и представља стање из којег се врши транзиција, док њена под-функција уводи ново стање за исти објекат податка, које уједно представља стање у које се врши транзиција. На почетку процеса генерисања, процесира се прво функција на највишем нивоу декомпозиције извршавањем процедуре BuildStateDiagrams. Уколико функција има дефинисано стање за објекат податка, креира се дијаграм стања за класу тог објекта податка и иницијализује додавањем иницијалног и финалног стања, у супротном се дохвата већ креирани дијаграм стања. Дефинисано стање се додаје на дијаграм. Уколико функција није део описа машине стања, додаје се транзиција од иницијалног стања до новододатог стања. Уколико функција представља транзицију стања као део машине стања, дефинисано стање представља стање у које се врши транзиција. Стање из којег се врши транзиција је дефинисано у услову функције која представља машину стања. Функција која представља машину стања такође представља догађај за транзицију. Услов функције која представља машину стања представља услов за транзицију. Под-функције функције која представља транзицију, које имају извршиоца а не дефинишу стање представљају акције које се раде по транзицији. Стање из којег се врши транзиција и стање у који се врши транзиција се додају на дијаграм. Транзиција од претходно прододатог ка другододатом стању, заједно са догађајем, условом, акцијама се додаје на дијаграм. Под-функције обрађене функције се процесирају на исти начин. Након процесирања свих функција, раде се последња додавања на дијаграме. За свако стање на креираним дијаграмима из којих не постоји транзиција, додаје се транзиција на дијаграм у којем се стање налази од тог стања ка финалном стању на истом дијаграму. Овим се завршава креирање дијаграма стања. Остали детаљи алгоритма могу се пронаћи на Слици 17.

UML дијаграм стања, генерисан користећи описани алгоритам за опис архитектуре система за плаћање у продавници, је дат на Слици 18. Слика 18 приказује дијаграм стања

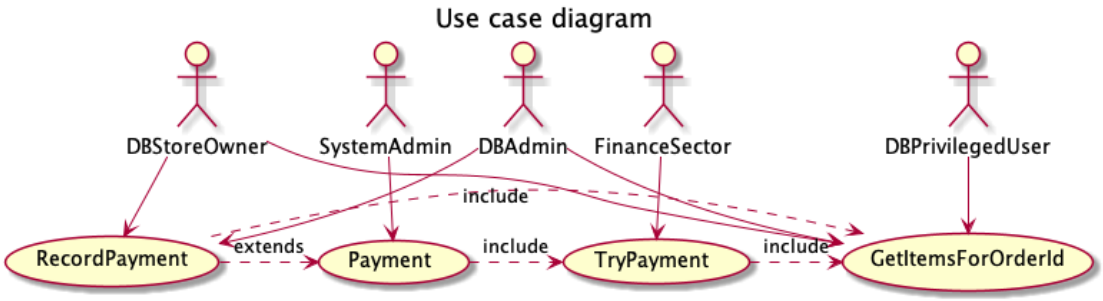
генерисан за објекат класе Payment. На линијама су написане транзиције, догађаји, услови и акције.

UML дијаграм случајева употребе генерише се на основу гледишта Функционално, Извршавање и Контекст. Псеудокод алгоритма за генерисање дијаграма случајева коришћења за опис архитектуре написан у АФД дат је на Слици 19. Случај коришћена у АФД представљен је функцијом над којом су дефинисани актери. На почетку процеса генерисања прво се процесира функција на највишем нивоу декомпозиције извршавањем процедуре BuildUseCaseDiagram. На почетку, актери се закључују за функцију. Актери за функцију су актери дефинисани за ту функцију, а у случају функције која је означена за поновну употребу и актери дефинисани за остале функције са истим називом. Уколико функција уводи новог актера у опис архитектуре, нови актер се додаје на дијаграм, случај коришћена уколико не постоји се додаје на дијаграм са истим називом као назив функције и асоцијација од додатог актера до додатог случаја коришћења се додаје на дијаграм. У случају да је за функцију

Algorithm Build Use case Diagrams

```
1:  procedure BuildUseCaseDiagram(topLevelFunctionality)
2:      AddBegin
3:      BuildFunctionality(topLevelFunctionality)
4:      AddEnd

5:  procedure BuildFunctionality(functionality)
6:      roles ← ConcludeRoles(functionality)
7:      hasNewRoles ← false
8:      for each role in roles do
9:          if RoleExistsInSuperFunctionalities(role) = false then
10:             hasNewRoles ← true
11:             useCaseName ← Name(functionality)
12:             AddAssociation(role, useCaseName)
13:     if hasNewRoles = true then
14:         superFunctionality ← GetSuperFunctionalityWithRoles
15:         if superFunctionality ≠ null then
16:             optionalExist ← OptionalExistsUpTo(functionality,
    superFunctionality)
17:             allAreExclusive ← AllAreExclusiveUpTo(functionality,
    superFunctionality)
18:             if allAreExclusive = true then
19:                 AddGeneralization(functionality, superFunctionality)
20:             else if optionalExist = true then
21:                 if IncludeExists(superFunctionality, functionality) =
    false then
22:                     AddExtends(functionality, superFunctionality)
23:             else
24:                 if ExtendsExists(functionality, superFunctionality) =
    true then
25:                     RemoveExtends(functionality, superFunctionality)
26:                 AddIncludes(superFunctionality, functionality)
27:     for each subFunctionality in SubFunctionalities(functionality) do
28:         BuildFunctionality(subFunctionality)
```

Слика 19 Псеудокод алгоритма за генерисање UML дијаграма случајева употребе.



Слика 20 Генерисани UML дијаграм случајева употребе за пример Процес плаћања у продавници.

дефинисан актер проналази се над-функција више у хијерархији, за коју је дефинисан актер, у односу на посматрану функцију. Уколико су све функције ниже у хијерархији у односу на пронађену над-функцију све до нивоа на којем је дефинисана посматрана функција ексклузивне случај употребе који дефинише над-функција је генерални случај употребе случаја употребе који дефинише посматрана функција. Тада се одговарајући елемент генерализације додаје на дијаграм. Уколико на путањи од посматране функције до пронађене над-функције постоји функција која се условно извршава, случај употребе који дефинише посматрана функција представља проширење случаја употребе који дефинише пронађена над-функција. Тада се одговарајући елемент проширења додаје на дијаграм. У супротном случај употребе који дефинише посматрана функција је увек укључен у случај употребе који дефинише пронађена над-функција. Тада се одговарајући елемент укључивања додаје на дијаграм. У случају да је између два случаја употребе у једном делу описа архитектуре закључен однос проширивања, а у другом однос укључивања, однос укључивања има предност. Под-функције функције се процесирају на исти начин. Овим се завршава креирање дијаграма случајева употребе. Остали детаљи везани за алгоритам могу се пронаћи на Слици 19.

UML дијаграм случајева употребе, генерисан коришћењем описаног алгоритма за опис архитектуре система за плаћање у продавници је дат на Слици 20. На генерисаном дијаграму су генерисана четири случаја употребе, један за сваку функцију која експлицитно има дефинисане актере. Сваки дефинисани актер у опису архитектуре је додат на дијаграм. Актер DBAdmin је дефинисан за функције GetItemsForOrderId и RecordPayment, дакле асоцијација је додата од актера DBAdmin ка случајевима употребе GetItemsForOrderId и RecordPayment. Функција TryPayment се не извршава условно у оквиру функције Payment, односно случај употребе Payment укључује случај употребе TryPayment. Функција RecordPayment се условно извршава унутар функције Payment, односно случај употребе RecordPayment проширује случај употребе Payment.

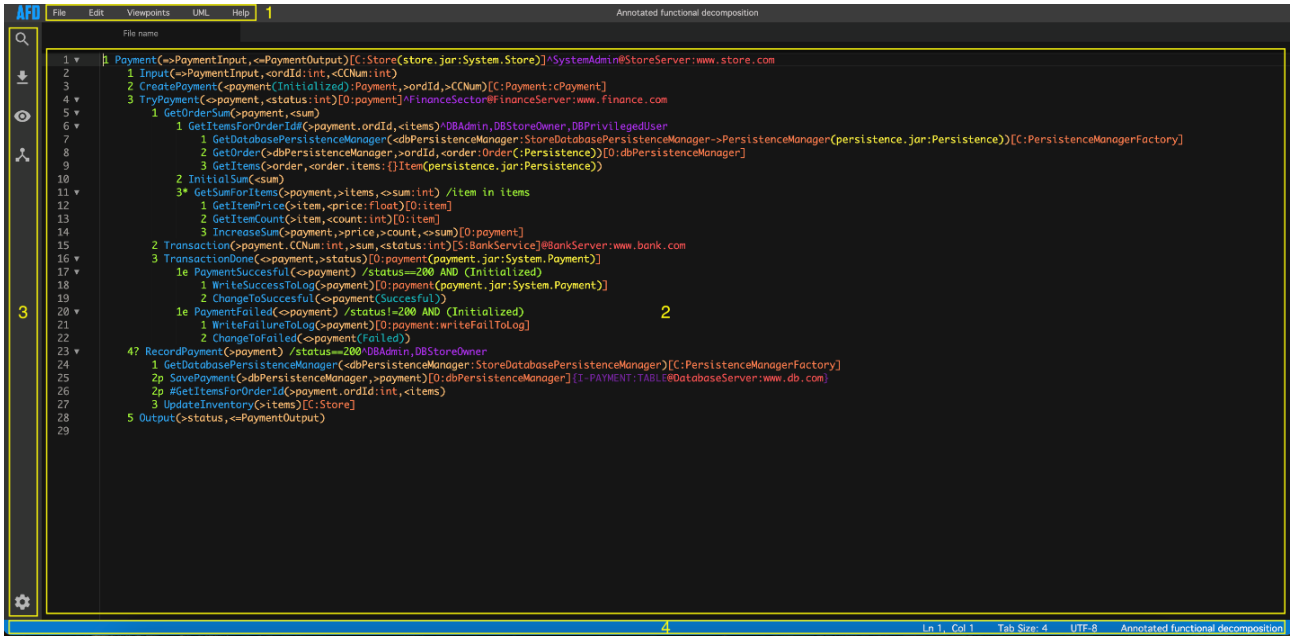
6. АФД алат

АФД алат пружа подршку за практичну употребу АФД. Алат омогућава заинтересованим странама да креирају, мењају, анализирају и управљају описом архитектуре система. Опис архитектуре представљен као интегрални модел у АФД може се поделити у погледе на основу гледишта које је одабрала заинтересована страна. Евентуалне неконзистентности између погледа детектује АФД алат и приказује их у интегралном моделу како би их заинтересована страна лакше разумела и њима руковала. Поред управљања у АФД алату, интегрални модел може бити преведен у одговарајуће UML дијаграме и евентуално искоришћен у другим алатима.

АФД алат је имплементиран као веб апликација и приказан је на Слици 21. Трака менија се налази на врху прозора. Трака менија састоји се од File, Edit, Viewpoints, UML и Help дугмади. Централни део прозора алата састоји се од едитора текста у средини и бројева линија са леве стране. Трака са алаткама се налази са леве стране прозора. Трака са алаткама састоји се од дугмета за претрагу, чување фајла, отварање бочног панела за рад са гледиштима, отварање бочног панела за рад са UML дијаграмима и дугмета за подешавање алата. Статусна трака се налази на дну прозора. Статусна трака приказује број линије и број колоне курсора и кодирање фајла.

Трака менија, која се налази на врху прозора, омогућава заинтересованим странама да отворе опис архитектуре, представљен фајлом са екстензијом afd, у алат едитора. Трака менија пружа скраћенице за претрагу текста у отвореном опису архитектуре, промену теме алата, отварање бочне траке за рад са гледиштима, отварање бочне траке за рад са UML. Трака менија такође пружа везе ка веб сајту алата и ка бочној траци за преглед скраћеница на тастатури.

Централни део прозора алата омогућава заинтересованим странама да креирају, мењају, анализирају и управљају описом архитектуре система. Бројеви линија приказани лево од описа архитектуре требало би да олакшају заинтересованим странама да идентификују линије описа архитектуре и да олакшају управљање описом. Опис архитектуре система је приказан у

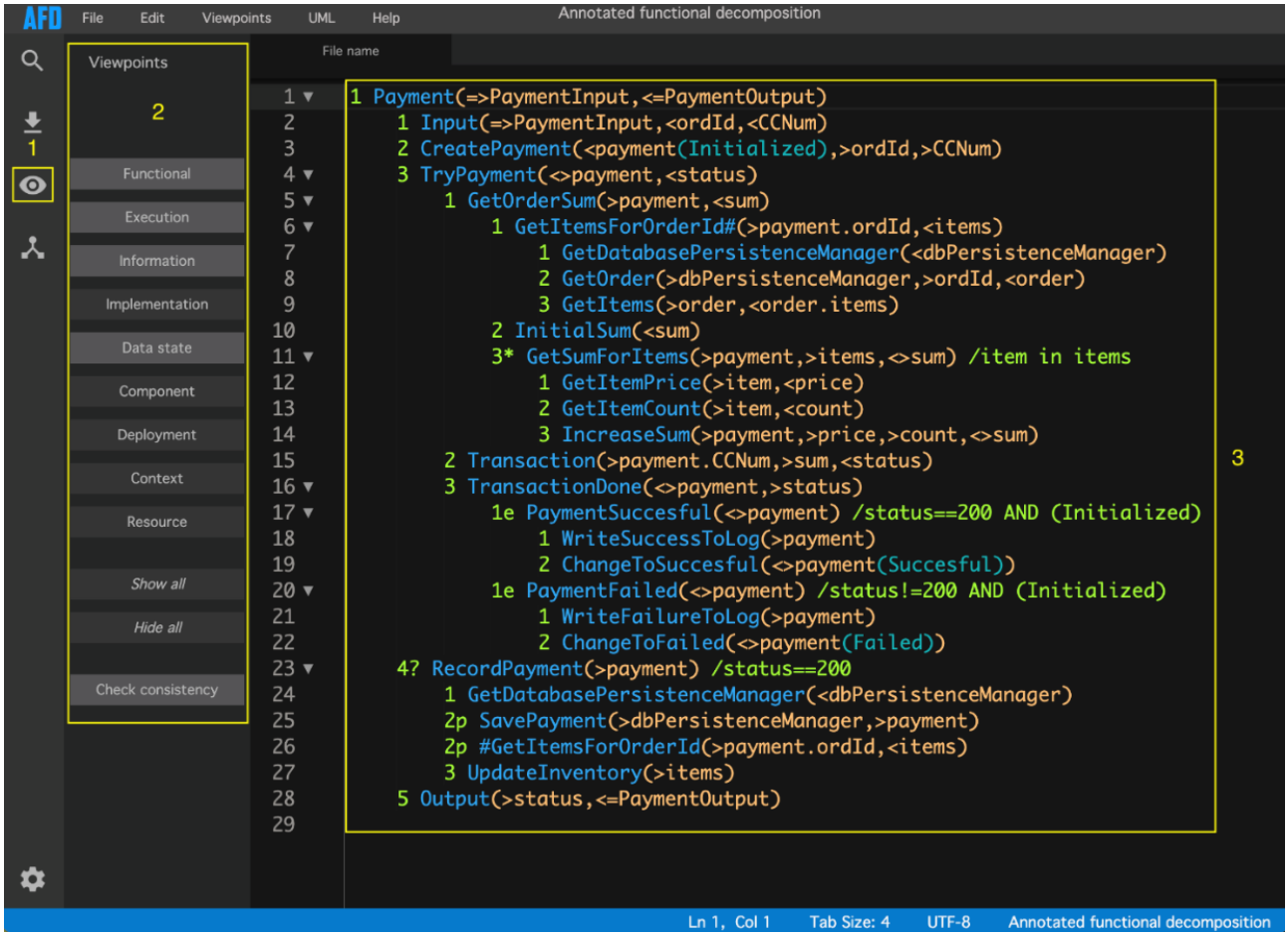


Слика 21 АФД алат. 1 – Трака менија, 2 – Едитор, 3 – Трака са алаткама, 4 – Статусна трака.

текстуалном едитору као текст написан у АФД језику. Слика 21 приказује исти пример процеса плаћања у продавници. Текст је обојен тако да свака боја представља различито гледиште. Погледи описа архитектуре дати на Слици 21 покривају све примере гледишта, дакле све доступне боје су приказане у примеру. Бојом кодирана гледишта треба да помогну заинтересованим странама у управљању описом архитектуре у случају да заинтересоване стране гледају опис архитектуре са више различитих гледишта.

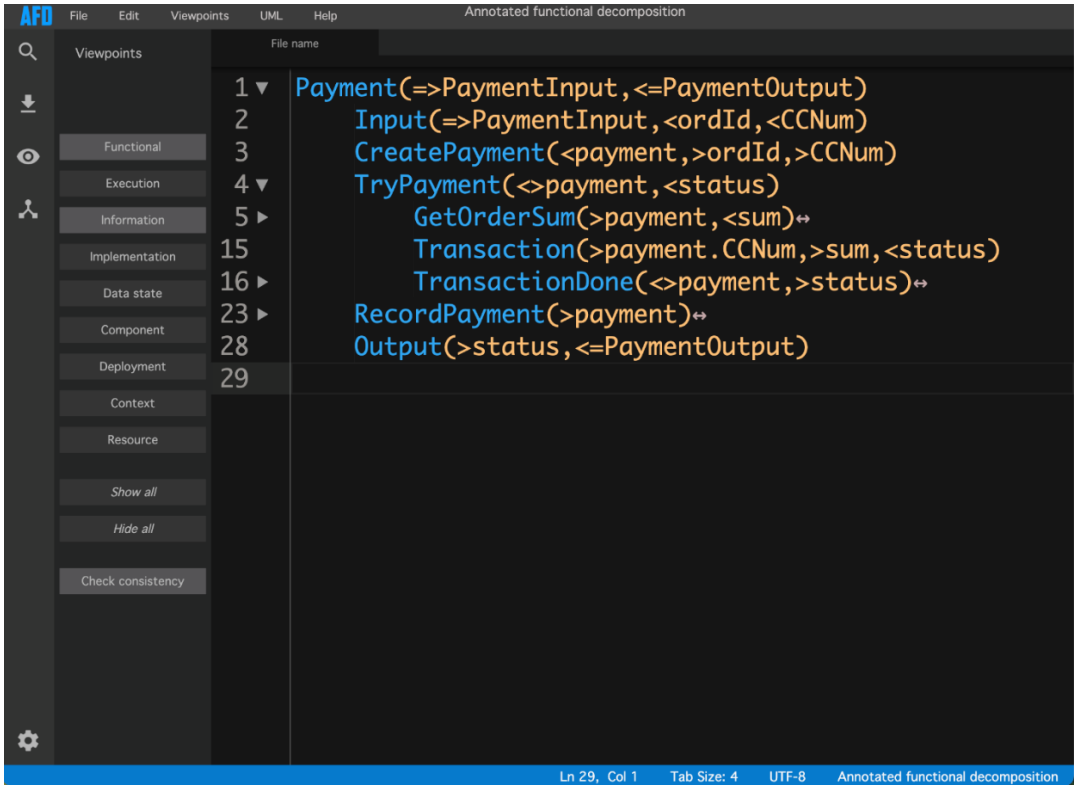
Трака са алаткама, која се налази на левој страни прозора, омогућава заинтересованим странама да претраже текст у опису архитектуре, сачувају опис на којем тренутно раде као фајл са екстензијом afd, да отворе бочну траку за рад са гледиштима, да отворе бочну траку за рад са UML, као и да отворе бочну траку са подешавањима алата. Слика 22 приказује отворену бочну траку за рад са гледиштима. Бочна трака за рад са гледиштима омогућава заинтересованим странама да погледају опис архитектуре система са селектованих гледишта од интереса. На основу селектованих гледишта, едитор на Слици 22 приказује исти пример описа архитектуре са Слике 21, али само са селектованих гледишта, у овом примеру: Функционално, Извршавање, Информације и Стање података.

АФД алат омогућава заинтересоване стране да гледају опис архитектуре са њиховог гледишта. Опис архитектуре се може састојати од низа функција којима заинтересоване стране тренутно не управљају, па се јавља потреба за фокусирањем на одређене функције. АФД алат помаже заинтересованим странама да сакрију одређене функције кликом на стрелице које се

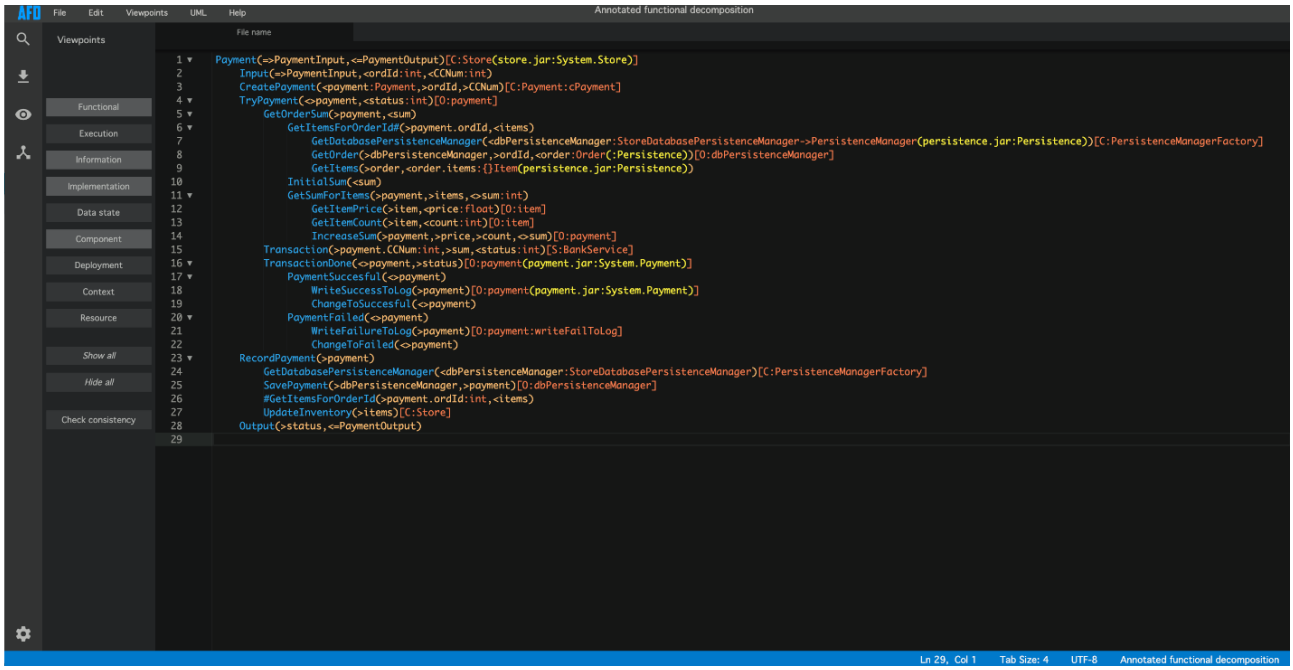


Слика 22 Селекција гледишта. 1 – Дугме за бочни бар са гледиштима, 2 – Бочни бар са гледиштима, 3 – Погледи описа архитектуре са селектованих гледишта.

налазе иза бројева линија. Кликом на стрелицу лево од одређене функције сакрива све његове под-функције. Слика 23 приказује сценарио где заинтересоване стране гледају опис архитектуре са гледишта Функционално и Информације и где су сакрили све функције које су део функција на линијама 5, 16 и 23. Након сакривања одређених гледишта и одређених под-функција, АФД алат омогућава заинтересованим странама да користе функције као што су копирање текста и генерисање UML дијаграма на основу видљивог дела описа архитектуре. АФД алат, омогућавајући заинтересованим странама да сагледају опис архитектуре са



Слика 23 Сакривање одређених функција од заинтересованих страна. Функције на линијама 5, 16 и 23 су пресавијене.

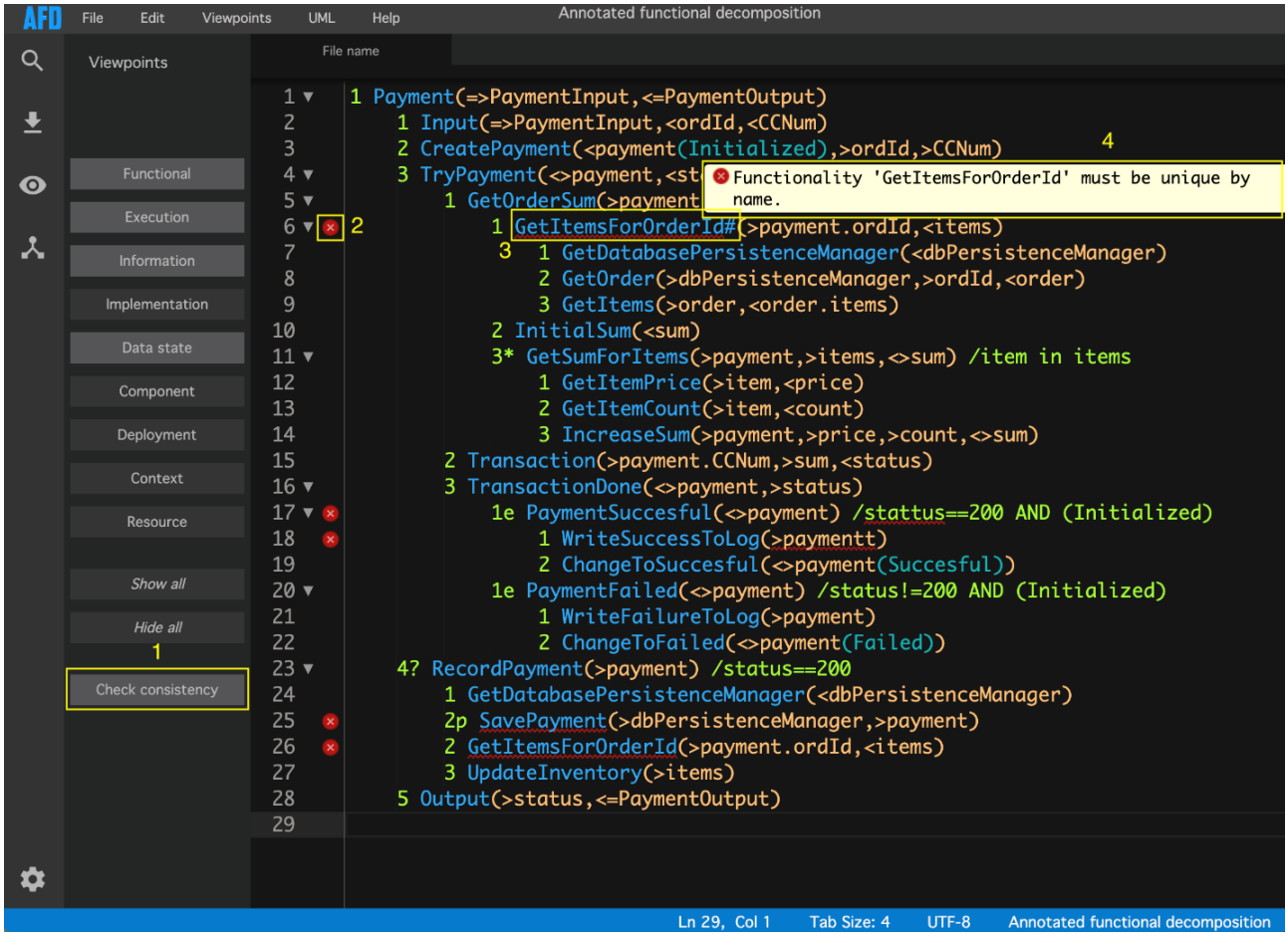


Слика 24 Управљање великим погледом у оквиру АФД алата.

различитих гледишта, као и да се фокусирају на функције од интереса, не оптерећују заинтересоване стране другим информацијама које им нису важне, па би тако АФД алат требало да помогне заинтересованим странама у управљању описом архитектуре.

Велики поглед је у АФД представљен погледом Компоненте. Управљање великим погледом у оквиру АФД алата ради се селекцијом гледишта Компоненте и управљањем његовим погледом као што је приказано на Слици 24. Компоненте су у АФД алату обојене жутом бојом. Гледиште Компоненте анотира гледиште Имплементација, па тако и гледишта Информације и Функционално. Дакле, гледишта Компоненте, Имплементација, Информације и Функционално су селектоване у оквиру бочне траке. Касније, велики поглед на систем се може такође приказати и на UML дијаграму компоненти који може бити генерисан АФД алатом.

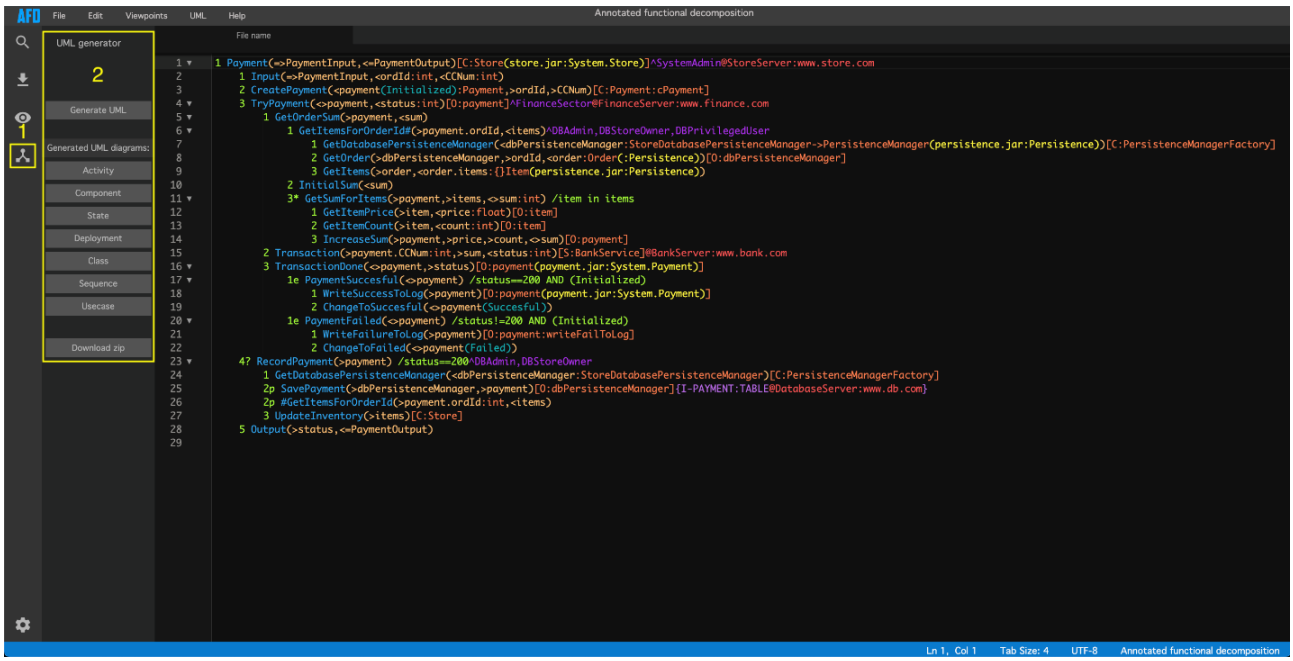
Примери дати на Сликама 21, 22, 23 и 24 су примери описа архитектуре чији су погледи међусобно конзистентни, дакле сва правила конзистентности су задовољена. Међутим, у процесу креирања описа архитектуре система, креирани погледи не морају нужно бити конзистентни, односно одређена правила конзистентности не морају бити задовољена. АФД алат пружа подршку процесу управљања конзистенцијом омогућавајући детекцију неконзистентности и приказивање неконзистентности заинтересованим странама. Провера конзистентности се може укључити или искључити користећи дугме „Check consistency“ на дну бочне траке за гледишта што је приказано на Слици 25.



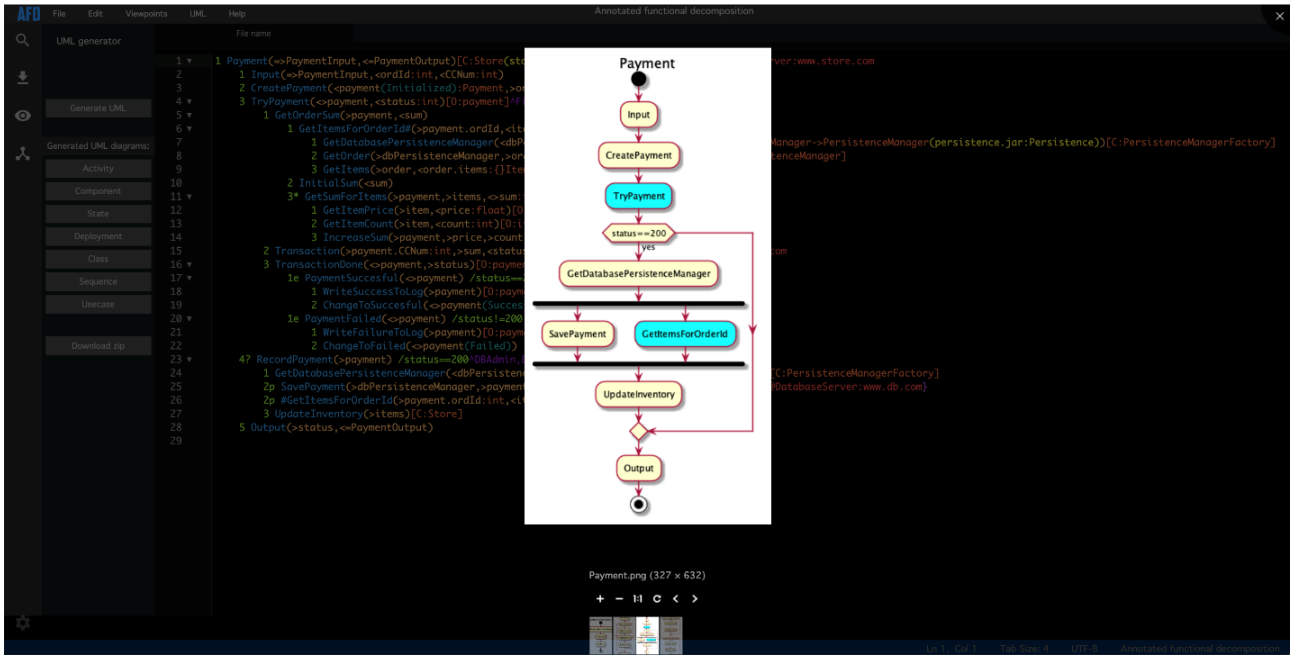
Слика 25 Неконзистентности у АФД. 1 – Дугме за проверу конзистентности, 2 – Незадовољена правила конзистентности груписана за линију, 3 – Означен део описа архитектуре који не задовољава правило конзистентности, 4 – Незадовољено правило конзистентности приказано као грешка заинтересованој страни.

Слика 25 илуструје пример описа архитектуре који се гледа са гледишта Функционално, Извршавања, Информације и Стања података у оквиру којих нека од правила конзистентности нису задовољена. Назив функције, која је дефинисана за поновну употребу, у линији 6 није јединствен из разлога што функција у линији 26 користи исти назив без референцирања функције за поновну употребу. Дакле, правило конзистентности дефинисано за гледиште Функционално наведено као: „Функција која се може поново употребити има знак # после свог назива и мора имати јединствен назив у декомпозицији, у смислу да на другим местима може постојати само функција која је употребљава.“ није задовољено. Позиционирање курсора над функцијом у линији 6 приказује информацију о грешки заинтересованој страни поводом правила конзистентности које није задовољено. У линији 17 први операнд логичког услова као део погледа Извршавање није претходно дефинисан као објекат података у погледу Информације за функцију у истој линији. Дакле, правило конзистентности дефинисано за комбинацију гледишта Функционално, Извршавање и Информације наведено као: „Ако је објекат података део логичког услова функције, тада тај објекат података мора бити излазни објекат података из функције која је исто увучена и која јој претходи, улазни објекат података у првој над-функцији, или итератор у услову за петљу прве над-функције посматране функције уколико она има дефинисан услов за петљу.“ није задовољено. У линији 18 за улазни објекат података није претходно дефинисан изворни објекат података у погледу Информације. Дакле, правило конзистентности дефинисано за комбинацију гледишта Функционално и Информације наведено као: „Уколико је објекат података улазни у функцију он мора бити излазни објекат података из функције која је исто увучена и која јој претходи, улазни објекат података у првој над-функцији, или итератор у услову за петљу прве над-функције посматране функције уколико она има дефинисан услов за петљу.“ није задовољено. У линији 25 функција је означена за извршавање у паралели, али ниједна функција која јој претходи или је следи није означена за извршавање у паралели. С тога, правило конзистентности дефинисано за комбинацију гледишта Функционално и Извршавање наведено као: „Непосредно пре или после функције која је означена за паралелно извршавање мора постојати бар једна функција која је исто увучена и означена за паралелно извршавање и има исти редни број.“ није задовољено. Последње, на линији 26 функција која је означена за секвенцијално извршавање нема додељен коректан редни број у односу на претходну функцију која је исто увучена. Према томе, правило конзистентности дефинисано за комбинацију гледишта Функционално и Извршавање наведено као: „Функција која је означена за секвенцијално извршавање мора имати редни број који је за један већи од редног броја функције која јој претходи и исто је увучена. У случају да функција нема функцију која је исто је увучена и претходи јој, њен редни број мора бити 1.“ није задовољено. На овај начин, АФД алат приказује детектоване неконзистентности, док је руковање неконзистентностима препуштено заинтересованим странама да ураде мануелно. Руковањем неконзистентностима и задовољењем свих правила конзистентности, опис архитектуре система је у конзистентном стању.

Креирани опис архитектуре који је у конзистентном стању, може бити преведен АФД алатом у UML дијаграме. АФД алат пружа превођење у седам UML дијаграма који највише одговарају АФД језику: дијаграм класа, компоненти, распоређивања, активности, секвенце, стања и случајева коришћења. Слика 26, која садржи исти пример процеса плаћања у продавници, приказује отворену бочну траку за рад са UML дијаграмима. Генератор UML дијаграма може бити покренут уколико су сва правила конзистентности задовољена, кликом

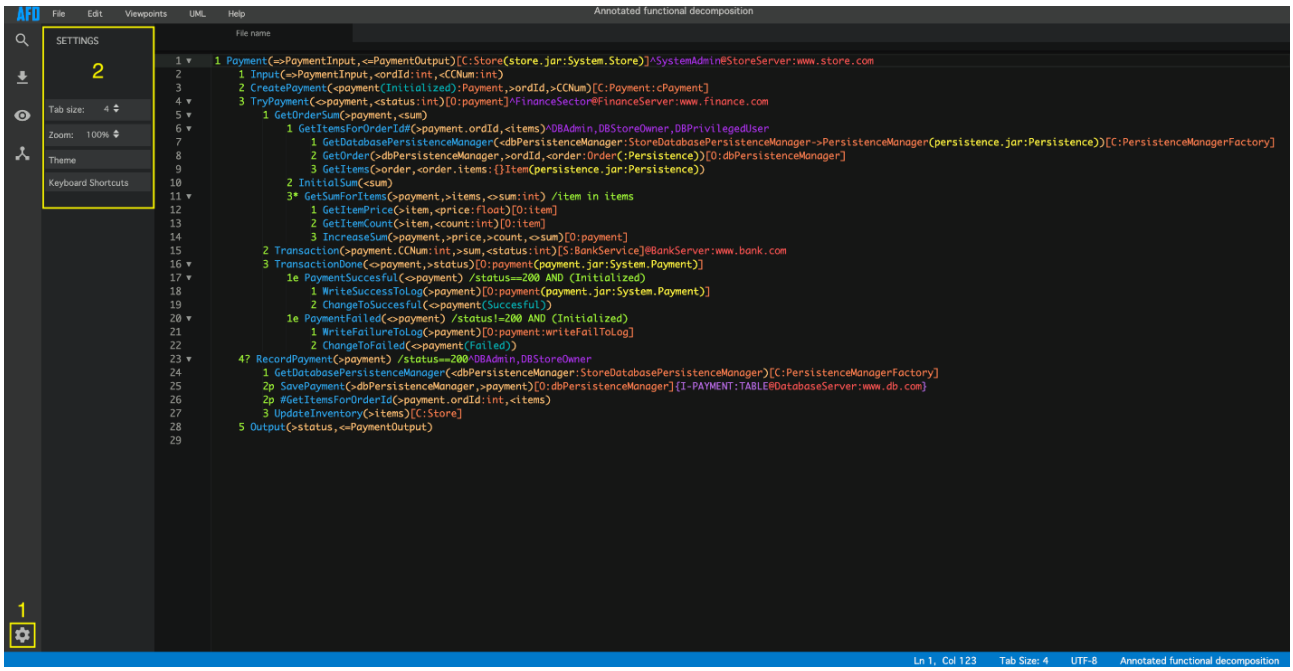


Слика 26 Генерисање UML дијаграма. 1 – Дугме за бочну траку за рад са UML дијаграмима, 2 – Бочна трака за рад са UML дијаграмима.



Слика 27 Генерисани UML дијаграм активности приказан у АФД алату.

на дугме „Generate UML“ на врху бочне траке. Генерисани типови UML дијаграма излистани су испод дугмета за генерисање. У овом примеру генерисани су сви типови UML дијаграма. Селекцијом једног генерисаног типа, сви дијаграми тог типа могу се видети испред едитора као што је приказано на Слици 27. Сви генерисани дијаграми могу бити сачувани на локалној машини кликом на дугме „Download zip“ које се налази испод листе генерисаних типова дијаграма. Креирање UML модела из текста је брже у односу на креирање UML модела у графичким алатима. Могућност АФД алата да генерише UML може бити мотивација за његову употребу у поређењу са текстуалним алатима за моделовање UML као што су Umple и USE алати [69,70,71]. За разлику од других алата за моделовање, АФД алат уводи методолошки приступ у моделовање проблема који остали алати не пружају [10].



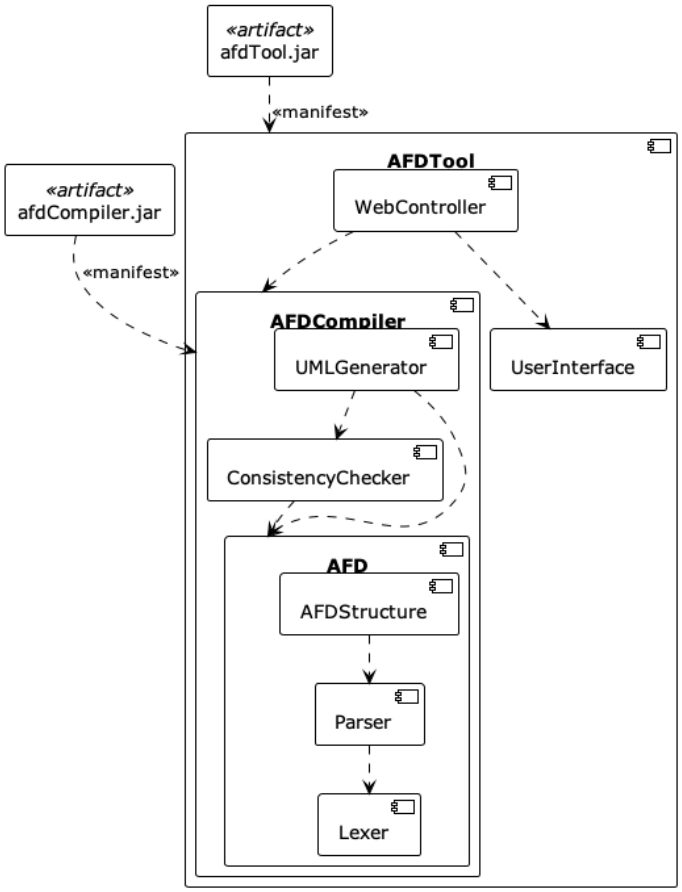
Слика 28 Подешавања у АФД алату. 1 – Дугме за бочни панел са подешавањима, 2 - Бочни панел са подешавањима.

Слика 28 приказује отворени бочни панел за подешавања. Бочни панел за подешавања омогућава заинтересованим странама да промене величину табулатора, да повећају или смање величину фонта текста који представља опис архитектуре система, да измене тему едитора, као и да виде листу скраћеница на тастатури. АФД алат тренутно подржава три теме. АФД алат од скраћеница на тастатури пружа подршку за претрагу текста, поништавање писања, понављање писања, копирање видљивог и сакривеног дела текста, копирање само видљивог дела текста, савијање функције која је идентификована курсором, савијање и одвијање свих функција у опису архитектуре, чување описа архитектуре, отварање постојећег описа архитектуре, скраћенице за приказивање и сакривање сваког од гледишта посебно, приказивање и сакривање свих гледишта осим гледишта Функционално које је увек видљиво.

АФД алат је имплементиран као веб алат у Јава програмском језику и доступан је на адреси: https://afd.etf.bg.ac.rs/. Са циљем приказивања начина рада АФД алата у наставку су описани структура и понашање алата као и његова имплементација. У оквиру потпоглавља 6.1. описане су основне компоненте АФД алата и њихови међусобни односи. У потпоглављу 6.2. дат је пример имплементације једног алгоритма за генерисање UML дијаграма. У оквиру потпоглавља 6.3. описана је интеграција АФД алата са алатом PlantUML ради генерисања UML дијаграма. У потпоглављу 6.4. описана је интеграција АФД алата са код-едитор компонентом CodeMirror ради генерисања корисничког интерфејса.

6.1. Компоненте алата

Архитектура АФД алата је описана UML дијаграмом компоненти који је приказан на Слици 29. АФД алат представљен је компонентом AFDTool која се састоји од компонената WebController, AFDCompiler и UserInterface које групишу функционалности алата у три целине. Прва целина врши обрађивање захтева корисника алата, друга целина обавља анализу описа архитектуре, док трећа целина омогућава генерисање корисничког интерфејса алата.



Слика 29 Компоненте АФД алата.

Захтеве корисника који су направљени кроз кориснички интерфејс алата процесира компонента WebController. Први пример захтева који може бити обрађиван је провера конзистенције на основу добијеног описа архитектуре уз детекцију евентуалних кршења правила конзистентности и њихових позиција у опису. Други пример је захтев за генерисањем UML дијаграма на основу описа архитектуре. Остали примери су захтеви за дохватањем типова генерисаних UML дијаграма, дохватањем генерисаних UML дијаграма и остали. WebController процесирање захтева који стижу од корисника делегира компоненти AFDCompiler, од које након добијања одговора исти делегира компоненти UserInterface. Компонента UserInterface одговор форматира и представља кориснику на одговарајући начин у оквиру интерфејса алата.

Добијене описе архитектура процесира компонента AFDCompiler. У оквиру ње компонента AFD ради прву фазу процесирања. Описи се разлажу на токене које идентификује компонента Lexer упаривањем текста у регуларне изразе. Опис архитектуре даље парсира компонента Parser на основу добијених токена и синтаксних правила језика. При парсирању се врши провера синтаксних правила и формирају терминални и нетерминални симболи. Симболе даље процесира компонента AFDStructure и креира структуру описа архитектуре, засновану на објектима и њиховим међусобним односима, која је погодна ради касније анализе.

Креирану структуру компонента AFD даље пружа остатку компоненте AFDCompiler. Структура се анализира ради провере правила конзистентности над описом у оквиру компоненте ConsistencyChecker и ова анализа представља другу фазу процесирања описа. Свако правило посебно се проверава и као резултат се детектују делови описа који крше

посматрано правило и њихове позиције у текстуалном опису. Сваком кршењу правила придружује се и одговарајуће текстуално објашњење.

Креирана структура података на основу описа архитектуре и резултат провере конзистенције су потребни ради генерисања UML дијаграма које представља трећу фазу процесирања. Генерисање UML дијаграма обавља компонента UMLGenerator у случају да не постоје прекршена правила конзистентности. На почетку се за сваки тип UML дијаграма проверава да ли у опису архитектуре постоје информације које се на дијаграмима тог типа могу представити и формирају се типови UML дијаграма који ће бити генерисани. Пример провере за генерисање дијаграма класа је да у опису архитектуре у оквиру погледа Имплементација треба да постоји наведен бар један назив класе. За сваки тип UML дијаграма процесира се добијена структура описа посебним алгоритмом и као резултат генерисања добијају се UML дијаграми одговарајућег типа.

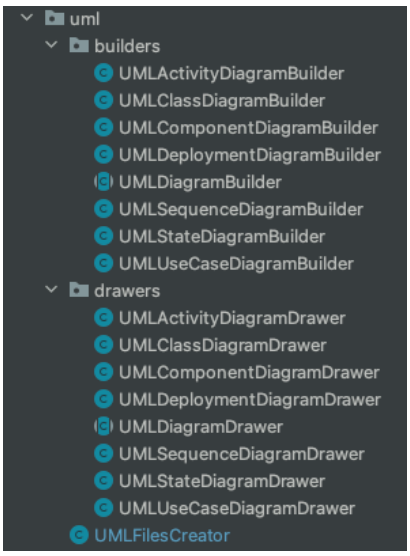
Од три фазе процесирања прва фаза, парсирање и креирање структуре на основу описа архитектуре, се увек ради приликом процесирања описа. Друга и трећа фаза су опционе и оне се раде уколико WebController то затражи у зависности од захтева корисника. На пример уколико корисник алата жели проверу конзистенције док креира опис, контролер ће захтевати обављање прве и друге фазе процесирања. У случају да корисник затражи генерисање UML дијаграма контролер ће захтевати обављање све три фазе процесирања. Након што компонента AFDCompiler обави процесирање наставља се са креирањем корисничког интерфејса.

Компонента WebController резултат процесирања предаје компоненти UserInterface која је одговорна за креирање корисничког интерфејса. Подаци добијени процесирањем се интерпретирају и на одговарајући начин представљају у оквиру интерфејса. Први пример је допуњавање постојећег описа архитектуре информацијама везаним за прекршена синтаксна правила или правила конзистентности и на основу тако допуњеног описа креира графички интерфејс. Други пример је креирање интерфејса на основу генерисаних UML дијаграма. Интерфејс креиран од стране компоненте UserInterface се даље приказује кориснику кроз веб претраживач. Оваква архитектура подсећа на образац дизајна Model–View–Controller, где контролер представља компонента WebController која обрађује захтеве корисника, утиче на генерисање структуре података од стране компоненте AFDCompiler, а такву структуру односно модел представља компонента UserInterface у оквиру интерфејса који генерише.

Читав алат, односно компонента AFDTool манифестује се фајлом afdTool.jar која је постављена на сервер машину. Компонента AFDCompiler манифестује се фајлом afdCompiler.jar који је део фајла afdTool.jar. Извршавањем фајла afdTool.jar на сервер машини постиже се покретање веб сервера који опслужује захтеве корисника алата.

6.2. Демонстрација имплементације алгоритма за генерисање UML дијаграма

Компонента UMLGenerator врши генерисање UML дијаграма коришћењем класа које су груписане у оквиру пакета uml и приказане су на Слици 30. Пакет uml састоји се од пакета builders који садржи класе које спроводе алгоритме за генерисање дијаграма и пакета drawers који садржи класе за исцртавање појединих елемената дијаграма. Додатно, у оквиру пакета uml налази се класа UMLFilesCreator која започиње процес генерисања дијаграма.

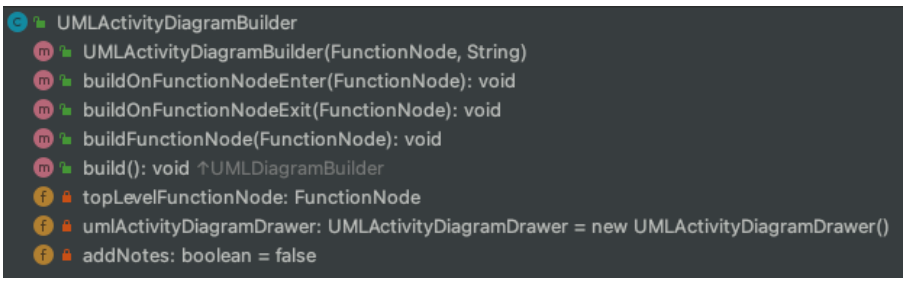


Слика 30 Класе коришћене за генерисање UML дијаграма.

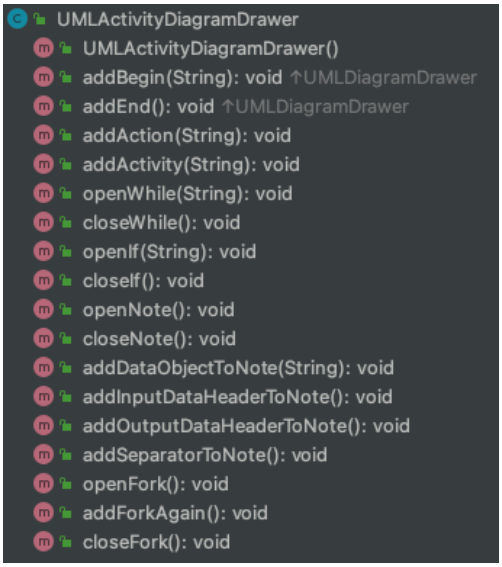
```
List<UMLDiagramBuilder> umlDiagramBuilders = new ArrayList<>();

if(canCreateUMLSequenceDiagrams())
    umlDiagramBuilders.add(new UMLSequenceDiagramBuilder(structures.getAFDStructure(),
structures.getClassStructure(), structures.getAFDStructure().getTopLevelFunctionNode(),
pathToUMLFolderPngAndTxt + "/sequence"));

if(canCreateUMLActivityDiagrams())
    umlDiagramBuilders.add(new
UMLActivityDiagramBuilder(structures.getAFDStructure().getTopLevelFunctionNode(),
pathToUMLFolderPngAndTxt + "/activity"));

if(canCreateUMLClassDiagrams())
    umlDiagramBuilders.add(new UMLClassDiagramBuilder(structures.getClassStructure(),
pathToUMLFolderPngAndTxt + "/class"));

if(canCreateUMLUsecaseDiagrams())
    umlDiagramBuilders.add(new UMLUseCaseDiagramBuilder(structures.getAFDStructure(),
pathToUMLFolderPngAndTxt + "/usecase"));

if(canCreateUMLComponentDiagrams())
    umlDiagramBuilders.add(new UMLComponentDiagramBuilder(structures.getComponentStructure(),
pathToUMLFolderPngAndTxt + "/component"));

if(canCreateUMLDeploymentDiagrams())
    umlDiagramBuilders.add(new UMLDeploymentDiagramBuilder(structures.getDeploymentStructure(),
pathToUMLFolderPngAndTxt + "/deployment"));

if(canCreateUMLStateDiagrams())
    umlDiagramBuilders.add(new UMLStateDiagramBuilder(structures.getStateStructure(),
pathToUMLFolderPngAndTxt + "/state"));

for (UMLDiagramBuilder umlDiagramBuilder : umlDiagramBuilders) {
    umlDiagramBuilder.startBuild();
}
for (UMLDiagramBuilder umlDiagramBuilder : umlDiagramBuilders) {
    umlDiagramBuilder.join();
}
```

Слика 31 Започињање генерисања UML дијаграма.

Део имплементације методе у оквиру класе UMLFilesCreator која покреће генерисање UML дијаграма приказана је на Слици 31. У првом делу имплементације за сваки тип UML дијаграма проверава се да ли у оквиру описа архитектуре постоје информације које се могу приказати. Уколико је могуће креирати дијаграм одређеног типа креира се његов градитељ представљен објектом одговарајуће класе из пакета builders. Градитељи раде као засебне нити спроводећи алгоритме генерисања.



Слика 32 Структура класе UMLActivityDiagramBuilder.



Слика 33 Структура класе UMLActivityDiagramDrawer.

Градитељ је представљен апстрактном класом UMLDiagramBuilder, а градитељи конкретних дијаграма су остале класе у пакету builders које проширују апстрактну класу UMLDiagramBuilder. Конкретно, у наставку ће бити приказана имплементација алгоритма за генерисање дијаграма активности у оквиру класе UMLActivityDiagramBuilder чији је псеудокод претходно дат.

Структура класе UMLActivityDiagramBuilder приказана је на Слици 32. Конструктор класе као први аргумент прима објекат класе FunctionNode који представља функцију у оквиру описа архитектуре за коју се дијаграм прави, а у случају првог дијаграма вредност аргумента је прва функција у опису, односно она која је компонована од свих функција система. Вредност свог првог аргумента конструктор уписује у поље topLevelFunctionNode. Други аргумент конструктора је путања до фолдера у који треба генерисати дијаграме активности. Поље umlActivityDiagramDrawer се користи током генерисања ради уноса појединих елемената на дијаграм активности. Методе које у префиксу имају текст build служе за генерисање дијаграма. Ове методе заснивају се на класама за исцртавање у оквиру пакета drawers па ће оне прво бити описане.

У оквиру пакета drawers цртач је представљен апстрактном класом UMLDiagramDrawer, а цртачи конкретних дијаграма су остале класе у пакету drawers које проширују апстрактну класу UMLDiagramDrawer. Конкретно, овде је приказан део дефиниције класе UMLActivityDiagramDrawer за исцртавање дијаграма активности, а њена структура дата је на Слици 33. Класа се састоји од многобројних метода за уношење одређених елемената на

```
public void addAction(String actionName) {
    umlDiagramStringBuilder.append(":" + actionName + ";");
    umlDiagramStringBuilder.append(System.lineSeparator());
}
```

Слика 34 Дефиниција методе addAction.

```
public void build() {
    umlActivityDiagramDrawer.addBegin(topLevelFunctionNode.getName());
    buildFunctionNode(topLevelFunctionNode);
    umlActivityDiagramDrawer.addEnd();

    umlActivityDiagramDrawer.saveToFile(umlFolderPathString, topLevelFunctionNode.getName());
}
```

Слика 35 Дефиниција методе build.

```
public void buildFunctionNode(FunctionNode functionNode) {
    log.info("Building function node at line " + functionNode.getLine());

    buildOnFunctionNodeEnter(functionNode);
    if(!functionNode.isReferenced() || functionNode == topLevelFunctionNode)
        if(!functionNode.implementationFlowExists() || functionNode == topLevelFunctionNode)
            for (FunctionNode subFunctionNode : functionNode.getSubFunctionsList()) {
                buildFunctionNode(subFunctionNode);
            }
    buildOnFunctionNodeExit(functionNode);
}
```

Слика 36 Дефиниција методе buildFunctionNode.

дијаграм активности као што су методе addAction која додаје акцију, addActivity која додаје активност, openFork која додаје чвор за рачвање на дијаграм активности и остале. Додавање елемената врши се уписом у текст који се касније додаје у текстуални фајл. Текстуални фајл се затим процесира и из њега генерише слика која представља дијаграм. Дефиниција методе addAction која врши додавање текста везано за елемент акције дата је на Слици 34. Текст у себи садржи назив акције и додаје се на досада формирани текст који ће на крају бити уписан у текстуални фајл. Текст има посебан формат како би се касније могло извршити његово парсирање и генерисање слике.

У оквиру класе UMLActivityDiagramBuilder метода build приказана на Слици 35 започиње генерисање UML дијаграма тако што користећи објекат umlActivityDiagramDrawer на почетку дода наслов дијаграма у текстуални опис, процесира функцију система методом buildFunctionNode, заврши формирање текста и текст упише у фајл.

Метода buildFunctionNode чија је дефиниција приказана на Слици 36 за сваку функцију описа позива методу buildOnFunctionNodeEnter на почетку процесирања и buildOnFunctionNodeExit на крају процесирања, док се између та два позива процесирају под-функције рекурзивним позивом методе buildFunctionNode. Под-функције се процесирају под условом да функција која се обрађује није означена за поновну употребу или немају дефинисаног извршиоца, јер се у супротном за њих касније креира генератор посебног дијаграма активности. Додатно, ови услови не важе за прву функцију описа чије се под-функције увек процесирају.

Поље topLevelFunctionNode класе UMLActivityDiagramBuilder представља функцију описа за коју се креира дијаграм активности. Метода buildOnFunctionNodeEnter представља први део процесирања функције и њен први део дефиниције је дат на Слици 37. Може се генерисати више дијаграма активности, а за сваки постоји посебан генератор у оквиру којег ће

```
public void buildOnFunctionNodeEnter(FunctionNode functionNode){
   if(functionNode.containsUserDataFlowInput() && functionNode == topLevelFunctionNode)
      return;

   if(functionNode.getName().startsWith("Output") && functionNode.isOptional()) {
      umlActivityDiagramDrawer.openIf(functionNode.getCondition());
      return;
   }

   if(functionNode.isParallel()) {
      if(functionNode.getPreviousFunctionNodeOnSameLevel() != null &&
functionNode.getPreviousFunctionNodeOnSameLevel().isParallel())
         umlActivityDiagramDrawer.addForkAgain();
      else
         umlActivityDiagramDrawer.openFork();
   }

   if(functionNode.getName().equals("Input")){
      umlActivityDiagramDrawer.addAction("Input");
   }
   else if(functionNode.isLoop()) {
      if(functionNode != topLevelFunctionNode) {
         umlActivityDiagramDrawer.openWhile(functionNode.getCondition());
         if(functionNode.implementationFlowExists()) {
            if(functionNode.subfunctionExists()) {
               umlActivityDiagramDrawer.addActivity(functionNode.getName());
               UMLActivityDiagramBuilder umlADBuilder = new
UMLActivityDiagramBuilder(functionNode, umlFolderPathString);
               umlADBuilder.build();
            }
            else
               umlActivityDiagramDrawer.addAction(functionNode.getName());
         }
         else
            return;
      }
      else
         return;
   }
```

Слика 37 Дефиниција методе buildOnFunctionNodeEnter - први део.

```
else if(functionNode.isOptional()) {
   if(functionNode != topLevelFunctionNode) {
      umlActivityDiagramDrawer.openIf(functionNode.getCondition());
      if(functionNode.implementationFlowExists()) {
         if(functionNode.subfunctionExists()) {
            umlActivityDiagramDrawer.addActivity(functionNode.getName());
            UMLActivityDiagramBuilder umlADBuilder = new UMLActivityDiagramBuilder(functionNode,
umlFolderPathString);
            umlADBuilder.build();
         }
         else
            umlActivityDiagramDrawer.addAction(functionNode.getName());
      }
      else
         return;
   }
   else
      return;
}
```

Слика 38 Дефиниција методе buildOnFunctionNodeEnter - други део.

ово поље имати различиту вредност. Посебан дијаграм активности се не прави за функцију која декомпонује улазни ток података, што представља прву проверу у оквиру методе buildOnFunctionNodeEnter. Након ове иницијалне провере и додавања елемента за одлуку са одговарајућим условом на дијаграм у случају функције која компонује излазни ток података врши се провера да ли постоји ознака за паралелно извршавање. Уколико постоји ознака за

```
         else if(functionNode.isStandard()){
   if(functionNode.isReusable()) {
      if(functionNode != topLevelFunctionNode ) {
         if(functionNode.getNumberOfReusableFunctionNodesWithSubfunctions() == 1)
            umlActivityDiagramDrawer.addActivity(functionNode.getName());
         else
            umlActivityDiagramDrawer.addAction(functionNode.getName());
      }

      if(functionNode.isReferenced() &&
functionNode.getNumberOfReusableFunctionNodesWithSubfunctions() == 1) {
         if(functionNode != topLevelFunctionNode) {
            UMLActivityDiagramBuilder umlADBuilder = new UMLActivityDiagramBuilder(functionNode,
umlFolderPathString);
            umlADBuilder.build();
         }
      }
   }
   else if((!functionNode.isReusable() && functionNode != topLevelFunctionNode) ||
(functionNode.isReferenced() && functionNode == topLevelFunctionNode)){
      if(!functionNode.implementationFlowExists() && functionNode.subfunctionExists() &&
!functionNode.containsUserDataFlowOutput())
         return;

      if(functionNode.implementationFlowExists() && functionNode.subfunctionExists()) {
         umlActivityDiagramDrawer.addActivity(functionNode.getName());
         UMLActivityDiagramBuilder umlADBuilder = new UMLActivityDiagramBuilder(functionNode,
umlFolderPathString);
         umlADBuilder.build();
      }
      else
         umlActivityDiagramDrawer.addAction(functionNode.getName());
   }
   else return;
}
```

Слика 39 Дефиниција методе buildOnFunctionNodeEnter - трећи део.

паралелно извршавање додаје се одговарајући елемент за рачвање на дијаграм. Додаје се акција на дијаграм у случају функције која декомпонује улазни ток података. У случају функције која има ознаку за извршавање у петљи, ако се за њу не креира тренутни дијаграм активности додаје се елемент за одлуку са уловом који дефинише извршавање у петљи. Функција се даље процесира уколико има извршиоца тако да ако функција има под-функције за њу се покреће генерисање засебног дијаграма активности на исти начин, а у супротном се додаје акција са њеним називом на тренутни дијаграм активности. Аналогно се поступа у случају да функција има ознаку за условно извршавање када се додаје елемент за одлуку са условом за извршавање, што је приказано на Слици 38.

Последњи део методе buildOnFunctionNodeEnter који се извршава за функције које немају посебну ознаку у извршавању приказан је на Слици 39. У случају функције која је означена за поновну употребу или да употребљава другу функцију може се додати активност или акција. Уколико постоји више таквих функција са истим називом и функција која се употребљава има под-функције, додаје се активност јер ће за такву функцију бити креиран посебан дијаграм активности. У супротном се додаје акција. Додатно, за функцију која је означена за поновну употребу креира се посебан дијаграм активности. На крају у случају да функција није означена за поновну употребу или је означена да употребљава другу функцију додаје се акција или активност на већ описан начин у зависности од тога да ли имају под-функције и извршиоца.

Метода buildOnFunctionNodeExit завршава процесирање функција и она се позива након методе buildOnFunctionNodeEnter и након процесирања свих под-функција, а њена

```
public void buildOnFunctionNodeExit(FunctionNode functionNode){
   if(functionNode.getName().equals("Input") || functionNode.containsUserDataFlowInput())
      return;

   if(functionNode.getName().startsWith("Output")){

      if(functionNode.isOptional()) {
         umlActivityDiagramDrawer.closeIf();
      }
   }
   else if(functionNode.isLoop()) {
      if(functionNode == topLevelFunctionNode)
         return;

      umlActivityDiagramDrawer.closeWhile();
   }
   else if(functionNode.isOptional()) {
      if(functionNode == topLevelFunctionNode)
         return;

      umlActivityDiagramDrawer.closeIf();
   }
   else if(functionNode.isStandard()){
      if((!functionNode.isReusable() || functionNode == topLevelFunctionNode) &&
!functionNode.implementationFlowExists() && functionNode.subfunctionExists())
         return;

   }

   if(functionNode.isParallel()) {
      if(functionNode.getSucceedingFunctionNodeOnSameLevel() == null ||
!functionNode.getSucceedingFunctionNodeOnSameLevel().isParallel())
         umlActivityDiagramDrawer.closeFork();
   }
}
```

Слика 40 Дефиниција методе buildOnFunctionNodeExit.

дефиниција дата је на Слици 40. Ова метода обавља сличне почетне провере као метода buildOnFunctionNodeEnter. Даље, у случају функције која је означена за условно извршавање на дијаграм активности додаје елемент за спајање одвојених токова контроле. За функцију која је означена за извршавање у петљи додаје ток контроле ка одговарајућем претходно додатом чвору за одлуку. За функцију која има ознаку за паралелно извршавање и последња је у низу функција означених за паралелно извршавање додаје се симбол за спајање паралелних токова контроле. Додатно, прескачу се функције које су се прескакале и у методи buildOnFunctionNodeEnter јер се не представљају на дијаграму активности. Овим се завршава процесирање функције и њених под-функција ради генерисања дијаграма активности.

6.3. Интеграција АФД алата са PlantUML

Инстанце класа у оквиру пакета drawers врше генерисање текстуалних фајлова који описују UML дијаграме. Парсирање текстуалних фајлова АФД алат делегира алату PlantUML [72] ради генерисања слика UML дијаграма. PlantUML омогућава генерисање великог броја UML дијаграма, са тим што неколико стандардних UML 2 елемената приказује на мало другачији начин како би повећао своју употребљивост. Пример текстуалног фајла генерисаног од стране АФД алата за генерисање UML дијаграма активности Payment за пример процеса плаћања у продавници дат је на Слици 41.

PlantUML парсира текстуални фајл и за одређене делове текста генерише одговарајуће елементе на слици која представља UML дијаграм. Парсирање почиње од линије у којој пише @startuml, након чега је наведен назив дијаграма активности Payment. Текст start пресликаће

```
@startuml
title Payment
start
:Input;
:CreatePayment;
#Cyan:TryPayment;
if(status==200) then (yes)
:GetDatabasePersistenceManager;
fork
:SavePayment;
fork again
#Cyan:GetItemsForOrderId;
end fork
:UpdateInventory;
endif
:Output;
stop
@enduml
```

Слика 41 Генерисани текстуални опис UML дијаграма активности Payment за пример Процес плаћања у продавници.

се у стартни симбол на дијаграму. Текст :Input; и текст :CreatePayment; пресликаће се на одговарајуће акције Input и CreatePayment на дијаграму активности које се повезују током контроле од стартног симбола. Активности се на дијаграму представљају другом бојом у односу на акције, па се текст #Cyan:TryPayment; пресликава на активност на дијаграму. Чвор за одлуку добија се на основу текста if(status==200) then (yes) у оквиру којег је дефинисан услов за одлуку status==200 који ће бити уписан у оквиру чвора као и назив излазног тока контроле yes за случај испуњеног услова. Наредни текст до линије endif биће пресликан на елементе у оквиру тока контроле yes. У оквиру тока контроле yes биће додата акција GetDatabasePersistenceManager као и чвор за паралелно рачвање на основу текста fork. Текст fork again раздваја паралелне токове контроле који се спајају у чвору који се додаје на основу текста end fork. На основу тога у први паралелни ток додаје се акција SavePayment, а у други паралелни ток активност GetItemsForOrderId. Након спајања паралелних токова, у оквиру тока контроле yes даље се додаје акција UpdateInventory. Након спајања два почетна тока контроле који одговарају успешно и неуспешно испуњеном услову у чвору за спајање који се додаје на основу текста endif, додаје се последња акција Output. На крају се додаје симбол краја на основу текста stop и парсирање фајла се завршава линијом у којој пише @enduml.

Генерисана слика UML дијаграма активности на основу текстуалног фајла дата је као први дијаграм на Слици 14. Текстуални описи других UML дијаграма које генерише АФД алат имају сличну структуру са тим да се користе други текстуални описи за елементе дијаграма. Генерисане слике UML дијаграма представљају резултат рада компоненте UMLGenerator.

6.4. Интеграција АФД алата са CodeMirror

Графички кориснички интерфејс је реализован имплементацијом фајлова формата html, css, js у оквиру компоненте UserInterface. За потребе пружања низа функционалности у оквиру едитора текста који се налази у централном делу АФД алата проширена је компонента CodeMirror [73]. CodeMirror је едитор кода који је направљен за коришћење на вебу и може се користити на веб сајтовима како би се имплементирала поља за унос текста са подршком за одређене функције. CodeMirror има велики програмски интерфејс који је проширив како би се омогућиле специфичне функције.

Мењање карактеристика карактера у оквиру текста као што су њихова боја и видљивост омогућене су у оквиру CodeMirror маркерима. Маркери представљају придруживања

кориснички дефинисаних css класа одређеним деловима текста како би он имао жељене карактеристике. Пружене су функције за маркирање одређеног дела текста, дохватање маркера, уклањање маркера и остале.

Како би се омогућило бојење текста у АФД алату направљен је засебан парсер текста који се у склопу корисничког интерфејса извршава у оквиру веб претраживача. Парсер представља проширење CodeMirror и дефинисан је као машина стања која на улазу добија карактере из текста, а чије је свако стање мапирано на једно од гледишта на опис архитектуре. Сваком гледишту одговара једна css класа која дефинише његову боју у текстуалном опису. На овај начин, у зависности од стања у којем се машина налази карактери добијају одређену боју. Парсирање текста ради се приликом сваке измене текста.

Направљено је проширење којим едитор кориснику пружа могућност сакривања свих под-функција одређене функције. За селектовану функцију одређује се текст који представља одговарајуће под-функције и ради његово маркирање css класом која врши сакривање текста. Аналогно је имплементирано приказивање свих под-функција одређене функције.

Сакривање погледа је проширење којим алат кориснику пружа могућност да сакрије поглед на основу изабраног гледишта. Сваком карактеру текста приликом парсирања придружује се тренутно стање машине стања, самим тим и гледиште којем припада. Маркирањем сваког карактера css класом одређеног гледишта даје му припадност одређеном погледу. Приликом сакривања погледа дохватају се сви карактери који њему припадају и додатно се маркирају ради сакривања. Аналогно је имплементирано приказивање одређених погледа.

Провера конзистенције уколико је затражена од стране корисника обавља се слањем текста из едитора веб серверу који обавља провере. Веб сервер враћа листу грешака у тексту, које могу бити синтаксне или семантичке као што су кршења правила конзистентности. Грешке су представљене листом чији елементи садрже информацију о деловима текста који се односе на грешку као и текстуалне описе. Делове текста алат маркира посебном css класом за грешке која омогућава да се за њих појаве поруке у одговарајућим деловима едитора.

Постоје и друга проширења компоненте CodeMirror у оквиру алата као што су скраћенице на тастатури. Скраћенице се могу користити за копирање видљивог текста из едитора, чување текста у оквиру фајла, отварање постојећег фајла и остале. Копирање видљивог дела текста заснива се на формирању видљивог текста анализом целокупног текста и маркера над њим.

АФД алат, иако је креиран као једноставан инструмент за коришћење АФД језика и не представља довољно зрело развојно окружење, демонстрира како АФД и интегрално моделовање могу да олакшају управљање описима архитектура. Креирање интегралног модела који може бити подељен у погледе њиховим бојењем или сакривањем уколико је потребно, може помоћи у разумевању комплексности које постоје иза моделованог система. У исто време имајући интегрални модел који је расположив све време, олакшава одређене активности управљања конзистенцијом. Штавише, могућност превођења интегралног модела у одговарајуће UML дијаграме чини усвајање АФД и његову могућу интеграцију са другим методологијама изводљивијом.

7. Евалуација

АФД је евалуиран у контексту других архитектуралних језика, на курсу информационих система и у контексту управљања конзистенцијом. У потпоглављу 7.1. су приказани резултати евалуације АФД у контексту других архитектуралнх језика и приказано је које захтеве за архитектуралним језицима пружа АФД и којим техникама. У потпоглављу 7.2. су приказане методе коришћене за спровођење квантитативних и квалитативних анализа везаних за практично искуство са коришћењем АФД на курсу информационих система, као и њихови резултати. У потпоглављу 7.3. приказане су методе коришћене за спровођење квантитативних анализа у вези практичног искуства са коришћењем АФД за потребе управљања конзистенцијом, као и њихови резултати. У потпоглављу 7.4. поновљени су одговори на полазне хипотезе који су дати у потпоглављима 7.1., 7.2. и 7.3. У потпоглављу 7.5. су дате претње валидности за спроведене експерименте.

7.1. Евалуација АФД у контексту других архитектуралних језика

За тему архитектуралних језика показано је велико интересовање од стране заједнице софтверске архитектуре и број архитектуралних језика се брзо повећавао [2]. 2000их година Медвидовић и остали дали су оквир за класификацију и поређење архитектуралних језика. Оквир дефинише карактеристике компоненти, конектора, архитектуралне конфигурације и подршке алата од којих су неки захтевани, а неки могу опционо бити обезбеђени од стране архитектуралних језика [74]. 2015. године Lago и остали предложили су оквир језичких захтева [37] на основу анкете Malavolta и осталих која је спроведена на 48 практиканата из 40 ИТ компанија усмерена ка разумевању потреба практиканата за архитектуралне језике [36]. 2018. године Ozkaya је користио захтеве дефинисане од стране Lago и осталих, декомпоновао их на под-захтеве према Lago и осталима и према осталим темељним публикацијама из софтверске архитектуре. Ozkaya је анализирао постојеће архитектуралне језике који су утврђени од стране Malavolta и осталих на основу скупа захтева који су веома важни практикантима са циљем да се помогне инжењерима нових архитектуралних језика у поређењу постојећих језика и утврђивању њихових предности и мана у погледу подршке за бројне захтеве [2]. Захтеви које је дефинисао Ozkaya подељени су у три групе: дефиниција језика, карактеристике језика и подршка алата које су приказане у Табели 7. У остатку овог потпоглавља АФД ће бити критички евалуиран поређењем са 124 остала архитектурална језика, групу по групу, у складу са сваким дефинисаним захтевом.

Дефиниција језика састоји се од захтева за нотацијом, нефункционалних захтева и захтева за формалну семантику. Нотација једног архитектуралног језика може бити текстуална или визуелна, од којих практиканти преферирају било коју у зависности од њихових искустава и потреба. АФД је архитектурални језик који користи текстуалну нотацију ради креирања описа архитектуре система. Текстуална нотација у АФД је дефинисана његовим синтаксним правилима. Текстуалну нотацију користи 40% архитектуралних језика.

Нефункционални захтеви описују квалитативне захтеве за софтверски систем, као што су перформансе и сигурносни захтеви. АФД пружа спецификацију нефункционалних захтева неформалном нотацијом која је покривена гледиштем Ресурси. Нефункционални захтеви су подржани од стране 21% архитектуралних језика, док је спецификација нефункционалних захтева неформалном нотацијом подржана од стране 13% архитектуралних језика.

Табела 7 Захтеви за архитектуралне језике које је дефинисао Ozkaya [2] и њихова подршка у АФД. 124 архитектурална језика реферисана у табели утврдили су Malavolta и остали.

Група захтева	Захтев	Под-захтеви	Пружено од стране АФД	Техника коришћена у АФД	Проценат и број архитектуралних језика који пружају захтев	Број архитектуралних језика који пружају под-захтев или захтев користећи исту технику као АФД[1][2]
Дефиниција језика	Нотација	Текстуална, Визуелна	Текстуална	Нема специфичне технике	100% (124)	Текстуална 40% (49)
	Нефункционални захтеви	-	Нефункционални захтеви	Неформална нотација	21% (26)	Нефункционални захтеви 13% (16)
	Формална семантика	-	Не	Нема специфичне технике	48% (60)	/
Карактеристике језика	Више гледишта	Логичко, Информације, Физичко, Распоређивање, Понашање, Конкурентност, Развој, Операције	Функционално, Извршавање, Информације, Имплементација, Стање података, Компоненте, Распоређивање, Контекст, Ресурси	Нема специфичне технике	91% (113)	Функционално +Компоненте +Контекст 91% (113) +Информације +Стање података 77% (96) +Извршавање 47% (58) Распоређивање 15% (19)
	Проширивост и подешавање	Синтаксно проширивање, Семантичко проширивање	Синтаксно проширивање, Семантичко проширивање	Наслеђивање језика	16% (20)	Синтаксно проширивање 0.01% (1) Семантичко проширивање 0% (0)
	Оквир за програмирање	Едитор за моделовање, Генератор софтверског кода	Едитор за моделовање	Нема специфичне технике	56% (70)	Едитор за моделовање 56% (70)
Подршка алата	Аутоматизована анализа	Конзистенција, Комплетност, Коректност, Компатибилност	Конзистенција	Кориснички дефинисано својство: Булова логика	47% (58)	Конзистенција (све технике) 19.3% (24)
	Верзионисање	-	Не	Нема специфичне технике	15% (18)	/
	Колаборација	Синхрона, Асинхрона	Не	Нема специфичне технике	8% (10)	/
	Управљање знањем	-	Управљање знањем	Нема специфичне технике	23% (29)	Управљање знањем 23% (29)
	Дизајн оријентисан на архитектуру софтвера	-	Дизајн оријентисан на архитектуру софтвера	Генерисање UML дијаграма	60% (75)	Нема података
	Управљање великим погледом	-	Управљање великим погледом	Композитне компоненте	56% (70)	Управљање великим погледом 35% (43)

[1] Име захтева/под-захтева написано је пре процента и броја архитектуралних језика.
[2] У случају да одређена техника није коришћена у АФД, приказани проценат и број архитектуралних језика не обухватају ту технику.

Семантика архитектуралних језика може бити дефинисана формално или неформално. Формална семантика архитектуралних језика дефинише се математички заснованим формалним методама, док се неформална семантика архитектуралних језика може дефинисати говорним језиком. Семантика АФД дефинисана је неформално семантичким правилима. Неформална семантика подржана је од стране 52% архитектуралних језика.

Карактеристике језика се састоје од захтева за више гледишта, проширивост и подешавање као и захтева за оквир за програмирање. Више гледишта разматраних од стране Ozkaya су Логичко, Информације, Физичко, Распоређивање, Понашање, Конкурентност, Развој и Операције [2]. Ова гледишта су мапирана на примере гледишта дефинисана у АФД у Табели 3. Примери гледишта које АФД пружа су Функционално, Извршавање, Информације, Имплементација, Стање података, Компоненте, Распоређивање, Контекст и Ресурси. Примери гледишта Функционално, Компоненте, Контекст, Информације, Стање података и Извршавање пружени су од стране 47% архитектуралних језика, док гледиште Распоређивање пружа само 15% архитектуралних језика.

Проширивање и подешавање односе се на могућност проширења језика захтевима од интереса. Проширивање језика може бити синтаксно и семантичко. Синтаксно проширивање укључује могућност за додавање нових, мењање и брисање постојећих елемената архитектуре без мењања семантике језика. Семантичко проширивање укључује могућност додавања нових гледишта, нових нефункционалних својстава, протокола интеракције и конектора. Семантичка екстензија може бити урађена увођењем нових синтаксних и семантичких правила у језик. АФД синтакса може бити проширена проширивањем постојећих и додавањем нових правила у језик. Проширивање АФД синтаксе промовише додавање нових елемената архитектуре, мењање постојећих или уклањање нежељених елемената архитектуре без мењања АФД семантике. АФД семантика може бити проширена наслеђивањем постојећих и додавањем нових правила у језик, писањем нових семантичких правила и коришћењем гледишта Ресурс. Проширивање АФД семантике промовише додавање нових гледишта, креирање елемената архитектуре унутар нових гледишта, и нефункционалних елемената унутар гледишта Ресурс. Проширивање и подешавање пружено је од стране 16% архитектуралних језика, где скоро ниједан архитектурални језик није користио наслеђивање језика као технику синтаксног и семантичког проширивања и подешавања.

Оквир за програмирање даје подршку архитектуралним језицима у њиховој примени у процесу развоја софтвера. Оквир за програмирање састоји се од едитора за моделовање који омогућава креирање погледа које архитектурални језик подржава, а опционо може генерисати код који имплементира софтвер. Спецификација, мењање и одржавање описа архитектуре у АФД омогућено је АФД алатом, који служи као едитор за моделовање. Оквир за програмирање пружа 56% архитектуралних језика и сви они пружају едитор за моделовање ради креирања описа архитектуре.

Подршка алата састоји се од захтева за аутоматизованом анализом, верзионисање, колаборацију, управљање знањем, дизајн оријентисан на архитектуру софтвера, управљање великим погледом. Аутоматизована анализа бави се аутоматизованим проверама ради циљева анализе: конзистенција, комплетност, коректност и компатибилност. Конзистенција се бави елементима архитектуре који се преклапају у различитим гледиштима а при том немају задовољавајући заједнички опис, дакле бави се било каквим контрадикцијама између

елемената у различитим погледима. Комплетност говори да ли опис архитектуре задовољава све дефинисане захтеве за системом. Коректност говори да ли опис архитектуре задовољава жељене особине система које су дефинисале заинтересоване стране као што су правила добре формираности или услови за застој система (енгл. deadlock). Компатибилност говори да ли се опис архитектуре слаже са архитектуралним стилом или смерницама за дизајн. АФД алат пружа аутоматизовану анализу са циљем конзистенције тако што АФД алат аутоматизовано детектује неконзистентности кроз семантичка правила која се ослањају на Булову логику, а делегира разрешавање неконзистентности заинтересованим странама. Аутоматизовану анализу пружа 47% архитектуралних језика, али ниједан од архитектуралних језика који подржава аутоматизовану анализу не узима у обзир сва 4 циља анализе. Од свих архитектуралних језика, 19.3% пружа аутоматизовану анализу са циљем конзистенције.

Верзионисање се бави чувањем и приступањем елементима описа архитектуре. Различите верзије елемената архитектуре могу бити сачуване у репозиторијуму и касније им се може приступити и поново се могу искористити као део описа архитектуре. Верзионисање није подржано од стране АФД алата. Верзионисање пружа 15% архитектуралних језика.

Колаборација заинтересованих страна смањује време потребно за креирање описа архитектуре и побољшава квалитет креираног описа архитектуре. Колаборација може бити синхрона или асинхрона. Синхрона колаборација захтева да заинтересоване стране раде на опису архитектуре у исто време, док асинхрона колаборација допушта заинтересованим странама да раде на опису архитектуре у различито време које највише одговара њиховом распореду. Колаборација није пружена од стране АФД алата. Колаборацију пружа 8% свих архитектуралних језика, и сви они пружају асинхрону колаборацију.

Управљање знањем бави се пружањем и дељењем знања о архитектуралним језицима практикантима. Знање о архитектуралним језицима се обично дели на веб сајту који пружа материјале који могу да користе практиканти као што су приручници, упутства за употребу, публикације, алати и остало и усмерава их на неку платформу за дискусију као што су форуми и корисничке групе. Знањем о АФД и АФД алату управља се веб сајтом који пружа увод у АФД и његов алат, публикације и контакт информације. Управљање знањем пружа 23% архитектуралних језика.

Дизајн оријентисан на архитектуру софтвера бави се интеграцијом процеса архитектуре софтвера у остале фазе процеса развоја софтвера као што су спецификација захтева, софтверски дизајн ниског нивоа, имплементација софтвера. Дизајн оријентисан на архитектуру софтвера у одређеним архитектуралним језицима пружа се њиховим алатом који омогућава функционалности као што су аутоматизована анализа, аутоматско генерисање софтверског дизајна ниског нивоа и софтверског кода, поновна употреба архитектуре софтвера преко репозиторијума, и интеграција спецификације архитектуре софтвера у спецификацију захтева. Дизајн оријентисан на архитектуру софтвера АФД пружа интеграцијом АФД алата као дела процеса за развој софтвера. Након што се дефинишу захтеви за системом што је резултат прве фазе процеса развоја софтвера, АФД алат може бити искоришћен ради креирања описа архитектуре система. Након што се креира опис архитектуре, АФД може да генерише UML дијаграме као визуелну репрезентацију архитектуре софтвера. UML се користи да пружи визуелни опис архитектуре из разлога што је доста популаран међу практикантима за моделовање архитектуре софтвера са различитих гледишта [1]. Према томе, полазна хипотеза:

„Опис креиран предложеним архитектуралним језиком је могуће аутоматски превести у одговарајући опис за архитектурални језик који се тренутно користи у пракси.“ је тачна. Након што се креира опис архитектуре и опционо генеришу UML дијаграми, треба урадити имплементацију софтвера као следећу фазу развоја софтвера.

Управљање великим погледом бави се техникама за приказивање великих и комплексних система на разумљив начин у оквиру погледа који може лако да се разуме и анализира. Док је подршка за више гледишта круцијална за представљање различитих аспеката софтверског система, архитектурални језик требало би такође да подржи спецификацију великог погледа. Велики поглед може бити урађен архитектуралним језиком користећи различите технике: композитне компоненте, композитне структуре конектора, наслеђивање, композитна понашања и аспектно-оријентисана спецификација. Композитне компоненте рукују скалабилношћу система кроз спецификацију композитних структура у погледу осталих под-компоненти и њихових интеракција. Композитне компоненте су најпожељнија техника за управљање великим погледом коју користе архитектурални језици. Композитне структуре конектора омогућавају спецификацију комплексних протокола интеракције на основу једноставних механизама интеракције. Наслеђивање као принцип дефинисан у парадигми објектно-оријентисаног софтверског инжењеринга, као техника у управљању великим погледом омогућава проширивање спецификација компоненти, њихове структуре и понашања, другом спецификацијом компоненте. Композитна понашања специфицирају понашања компоненти на основу понашања постојећих компоненти. Аспектно-оријентисана спецификација решава сложене преклапајуће проблеме, као што су сигурност, контрола приступа, нефункционална својства, и специфицира их модуларно као аспекте. Управљање великим погледом пружено је АФД алатом у оквиру погледа Компоненте, спецификацијом компоненти и њихове структурне композиције, користећи технику композитних компоненти. Управљање великим погледом пружа 56% архитектуралних језика, при чему композитне компоненте као технику управљања великим погледом обезбеђује 35% архитектуралних језика.

Из претходне анализе може се сумирати да АФД и његов алат пружају 9 од 12 захтева за архитектуралне језике осим формалне семантике, верзионисања и колаборације, и пружају 13 од 20 под-захтева за архитектуралне језике. Међутим, поред подршке за више погледа, која има циљ да повећа управљање описа архитектуре, АФД пружа интегрално моделовање. Према томе, полазна хипотеза која гласи: „Могуће је користити интегрално моделирање за креирање описа архитектуре софтверског система кроз више погледа.“ је тачна. Интегрално моделовање, иако није приказано као један од захтева за архитектуралне језике у Табели 7, олакшава управљање неконзистентностима без обзира на њихов извор што повећава управљивост описом архитектуре. Додатно, АФД имплементира све принципе рачунарског размишљања као што је већ описано, па је полазна хипотеза која гласи: „Могуће је предложити нов архитектурални језик који би подржао све принципе рачунарског размишљања.“ тачна.

7.2. АФД на курсу информационих система

Овде ће бити евалуирано практично искуство са коришћењем АФД током курса Информациони системи 1 (ИС1) на Електротехничком факултету, Универзитета у Београду [10]. ИС1 је обавезан курс за студенте Софтверског инжењерства треће године основних академских студија и изборни курс за студенте Рачунарске технике и информатике четврте

Табела 8 Преглед тема обрађених током курса Информациони системи 1.

Недеља	Предавања		Аудиторне вежбе		Лабораторијске вежбе	
	Тема	Трајање	Тема	Трајање	Тема	Трајање
H1	Увод у ИС	90 мин	Моделовање података – модел ЕО	90 мин		
H2	АФД	90 мин	АФД	90 мин		
H3	АФД	90 мин	АФД	90 мин		
H4	Приступи дизајнирању и анализи ИС	90 мин	JEEE – приступи развоју ИС	90 мин	Лаб 1 – АФД	180 мин
H5	Први колоквијум					
H6	UML – дијаграми класа	90 мин	Java Message Service - JMS	90 мин		
H7	Сценарио случаја употребе	90 мин	Java Message Service - JMS	90 мин		
H8	UML – дијаграми активности и стања	90 мин	Java Persistence API - JPA	90 мин	Лаб 2 – JMS	180 мин
H9	UML – дијаграми секвенце	90 мин	Java Persistence API - JPA	90 мин		
H10	Други колоквијум					
H11	Приступи имплементацији ИС	90 мин	Java Persistence API - JPA	90 мин		
H12	Архитектура ИС - SOA	90 мин	RESTful сервиси – JAX-RS	90 мин		
H13	Управљање подацима	90 мин	RESTful сервиси – JAX-RS	90 мин		
H14	Таксономија и класификација ИС	90 мин	Дискусија о пројектном задатку	90 мин	Лаб 3 – REST+JPA	180 мин
H15	Евалуација и оцењивање пројектног задатка					
	Завршни испит					

године основних академских студија. Недељни распоред часова обухвата два часа за предавања, два часа за аудиторне вежбе и решавање задатака и један час за лабораторијске вежбе. Курс носи 6 ЕСПБ (европски систем преноса и акумулације бодова). Предавања се фокусирају на теоретске концепте и лекције са мањим примерима. Аудиторне вежбе се састоје од решавања проблема из тема које се уче током лекција и интерактивних демонстрација користећи развојно окружење за Java EE, док лабораторијске вежбе представљају практичне задатке. Табела 8 даје преглед тема обрађених током курса. Коначна оцена добијена на курсу зависи од практичних и писмених задатака. Практични аспект укључује домаћи задатак (20%), и обавезне лабораторијске вежбе (20%). Остатак оцене (60%) се добија кроз два колоквијума (10% сваки) и завршног испита (40%). Намена евалуације била је да се процени да ли коришћење АФД олакшава креирање описа архитектуре у поређењу са архитектуралним језиком који се тренутно користи у пракси односно олакшава решавање проблема и омогућава бољи фокус на потребне логичке провере и ограничења за дизајнирање информационих система. Остатак описује методе коришћене за евалуацију и добијене резултате, док су претње валидност описане касније.

7.2.1. Експеримент 1 – Методе

Методе коришћене да процене да ли архитектурални језик АФД студентима може олакшати креирање описа архитектуре у поређењу са архитектуралним језиком који се тренутно користи у пракси односно олакшава решавање проблема у домену логичког дизајна информационих система, састојале су се од квантитативних евалуација [10]. Квантитативна евалуација посматрала је четири генерације студената са одсека за Софтверско инжењерство и Рачунарску технику и информатику. Све генерације студената училе су исте теме, укључујући АФД и UML, које су покривене истим материјалима. Током колоквијума, студенти су решавали проблеме креирања делова описа архитектура софтверских система односно логичког дизајна

информационих система примењујући и АФД и UML. Проблеми који су студенти решавали нису им били познати пре колоквијума. Проблеми су били сличног нивоа сложености, од којих је сваки садржао сценарио случаја употребе унутар домена проблема (на пример грађевинска компанија, фармацеутска компанија и приватна медицинска пракса). Свака генерација је решавала један проблем користећи АФД на првом колоквијуму и други проблем користећи UML на другом колоквијуму. Свака генерација је имала различит први и други проблем, ипак, неки проблеми су поновљени између генерација. Када је проблем поновљен између две генерације, од једне се тражило да користе АФД, док се од друге тражило да користе UML. Решавање проблема коришћењем АФД рађено је са гледишта Функционално, Извршавање, Информације и Ресурси, док је решавање проблема коришћењем UML рађено UML дијаграмима класа, секвенци и активности. Студентска решења су оцењивана само на основу тога колико су логичких провера и ограничења имплицираних проблемом успешно идентификовали и адресирали, док су синтаксни пропусти занемарени. Оцене које је једна генерација постигла на проблему користећи АФД и оцене које је нека друга генерација постигла на истом проблему користећи UML су искоришћене за квантитативну анализу. Међутим проблеме које је решавало мање од 40 студената користећи АФД или UML су искључени из анализе. Двосмерна анализа варијансе (ANOVA) и Kruskal–Wallis тестови су коришћени у сврхе анализе. Током ових анализа, просечне оцене посматраних генерација су такође узете у обзир да би се проценио њихов утицај на резултате евалуације.

7.2.2. Експеримент 1 – Резултати

Квантитативна евалуација је посматрала студенте који су решавали три различита проблема која се односе на логички дизајн информационих система. Студенти који су решавали проблеме користећи АФД су сматрани током евалуације експерименталним групама, а они који су решавали проблеме користећи UML су сматрани контролним групама. Преглед резултата студената из сваке групе и проблема [10] је дат у Табели 9.

Експерименталне групе постигле су више просечне оцене у односу на контролне групе за сваки проблем током периода посматрања, а за детаљнију анализу спроведен је двосмерни ANOVA тест. Први фактор у анализи био је посматрани проблем са три нивоа (проблем 1, проблем 2 и проблем 3). Други фактор је био посматрана група са два нивоа (експериментална и контролна). Зависна променљива била је резултат студената (резултујућа оцена на проблему)

Табела 9 Преглед резултата студената применом АФД и UML на различите проблеме.

Бр. проблема	Група	Генерација	Бр. студената	Просечна оцена на проблему (0-10)	Просечна оцена током студија (6-10)
1	АФД	2017/2018 СИ	113	8,022	8,345
	UML	2019/2020 СИ	144	7,608	8,334
2	АФД	2016/2017 СИ	91	7,956	8,457
	UML	2018/2019 СИ	103	7,757	8,304
3	АФД	2017/2018 РТИ	49	8,490	7,928
	UML	2016/2017 РТИ	85	7,124	8,547

Табела 10 Резултати двосмерног ANOVA теста за експеримент 1.

Фактори	Степени слободе	Средњи квадрат	F	Значајност
група	1	57,067	16,007	0,000
проблем	2	0,129	0,036	0,964
група * проблем	2	13,916	3,903	0,021
Грешка	579	3,565		

Табела 11 Резултати непараметарских независних узорака тестова Kruskal–Wallis за експеримент 1.

Бр. проблема	Степени слободе	Бр. студената	Значајност
1	1	257	0,110
2	1	194	0,007
3	1	134	0,000

за посматрану групу и посматрани проблем. Резултати двосмерног ANOVA теста [10] су дати у Табели 10.

Главни ефекат фактора посматране групе је статистички значајан, $F(1, 579) = 16,007$, $p = 0,000$, што сугерише да постоји разлика између експерименталних и контролних група, и као што је претходно наведено, експерименталне групе су постигле више просечне оцене. Главни ефекат фактора посматраног проблема није статистички значајан, $F(2, 579) = 0,036$, $p = 0,964$, што говори да не постоји општа разлика у постигнућима између типова проблема. Међутим, интеракција између два фактора је статистички значајна, $F(2, 579) = 3,903$, $p = 0,021$, што значи да је у одређеним групама, посматрани проблем могао утицати на резултате студената. Поред ове анализе, просечне оцене студената током студија су уведене као трећи фактор и показано је да имају значајан главни ефекат, тако да су више успешни студенти постигли у просеку боље резултате у односу на мање успешне студенте. ANOVA претпоставке нису у потпуности испуњене, па су такође спроведени непараметарски Kruskal–Wallis тестови који су дали исте резултате у погледу статистичке значајности [10] као што је приказано у Табели 11. На основу резултата овог експеримента утврђено је да је полазна хипотеза: „Коришћење новог архитектуралног језика може студентима олакшати креирање описа архитектуре у поређењу са архитектуралним језиком који се тренутно користи у пракси.“ тачна.

7.2.3. Експеримент 2 – Методе

Методе коришћене да процене да ли коришћење архитектуралног језика АФД може студентима бити лак за разумевање у поређењу са архитектуралним језиком који се тренутно користи у пракси и олакшава решавање проблема у домену логичког дизајна информационих система састојале су се од квалитативне евалуације. Квалитативна евалуација перципирања АФД од стране студената је извршена кроз добровољну, анонимну анкету студената из последње од четири посматране генерације. Ова анкета се састојала од више питања затвореног типа, од којих су се нека односила на АФД.

7.2.4. Експеримент 2 – Резултати

Анонимна анкета која је коришћена за квалитативну евалуацију садржала је четири питања која се односе на АФД, и била је попуњена од стране 54 студента која су похађала курс током школске 2019/2020 године. Резултати анкете [10] су дати у Табели 12. Студенти су одговарали на питања користећи петостепену Likert скалу. Више од две трећине испитаника одговорило је да је АФД лак за разумевање (74%) и коришћење за решавање проблема (67%). Сваки четврти студент је изјавио да АФД олакшава разумевање UML секвенцијалних дијаграма. Више од 39% студената је одговорило да је АФД примењив у пракси, док сличан број (32%) није био сигуран у вези са његовом примењивошћу. Овај последњи одговор је вероватно последица тога да АФД није тренутно широко препозната техника, као што је UML или друге технике, које се предају током курса. Након што су одговорили на питања анкете, студенти су могли да поделе коментаре у вези са начином на који се АФД предавао током курса.

Табела 12 Одговори на анкету изражени у процентима (%).

	Веома лако	Лако	Нисам сигуран	Тешко	Веома тешко
Разумевање АФД је	18,52	55,56	20,37	5,56	0
Коришћење АФД у решавању проблема је	12,96	53,7	16,67	16,67	0
	Дефинитивно не	Вероватно не	Могуће	Вероватно да	Дефинитивно да
АФД олакшава разумевање UML дијаграма секвенце	14,81	31,48	29,63	22,22	1,85
АФД је примењив у пракси	13,21	15,09	32,08	26,42	13,21

Коментари су сугерисали да би требало мање времена посветити АФД јер га није тешко разумети и користити, а да би било корисно увођење комплекснијих проблема уместо већег броја мање комплексних проблема. На основу резултата овог експеримента утврђено је да је полазна хипотеза: „Коришћење новог архитектуралног језика може студентима бити лак за разумевање у поређењу са архитектуралним језиком који се тренутно користи у пракси.“ тачна.

7.3. АФД у контексту управљања конзистенцијом

Овде ће бити евалуирано практично искуство са коришћењем АФД за потребе управљања конзистенцијом у описима архитектура софтверских система, од стране две групе испитаника. Првој групи припадају особе које су пратиле курс ИС1 на Електротехничком факултету, Универзитета у Београду, дипломирале су и запослене су у струци. Другој групи припадају студенти основних студија који су пратили курс ИС1 на Електротехничком факултету, Универзитета у Београду. Намена евалуације била је да се процени да ли коришћење АФД олакшава проналажење неконзистентности у описима архитектура у поређењу са другим архитектуралним језиком који се користи у пракси. У наставку су описане методе коришћене за евалуацију и добијени резултати, док су претње валидност описане касније.

7.3.1. Експеримент 3 – Методе

Методе коришћене да процене да ли архитектурални језик АФД може олакшати уочавање неконзистентности у опису архитектуре у поређењу са архитектуралним језиком који се тренутно користи у пракси састојале су се од квантитативних евалуација. Ради спровођења овог експеримента био је изазов пронаћи испитанике који би издвојили неколико сати ради спровођења експеримента с обзиром на њихове обавезе на послу или факултету. Квантитативна евалуација посматрала је 10 учесника експеримента који су завршили одсек Софтверско инжењерство или Рачунарска техника и информатика на Електротехничком факултету, Универзитета у Београду, дипломирале су. запослени су у струци и имали су сличне просечне оцене током студија. Додатно, квантитативна евалуација посматрала је 14 учесника експеримента који су студенти основних студија одсека Софтверско инжењерство или Рачунарска техника и информатика на Електротехничком факултету, Универзитета у Београду.

Сви учесници учили су исте теме, укључујући АФД и UML, које су покривене истим материјалима. Током тестирања учесници су тражили неконзистентности у описима архитектура креираним у АФД и у UML. Проблеми које су испитаници решавали нису им били познати пре тестирања. Проблеми су били сличног нивоа сложености, од којих је сваки садржао сценарио случаја употребе унутар домена проблема (на пример плаћање у продавници, куповина карата у биоскопу, креирање нове партије на шаховском турниру,

формирање рачуна у online биоскопу). Постојало је четири проблема. У сваком проблему постојала су два описа архитектуре где је први био у дефинисан у АФД, а други дефинисан у UML. Током тестирања сваки учесник је уочавао неконзистентности у сва четири проблема. Сваки непарни учесник је у првом проблему уочавао неконзистентности у АФД опису, у другом у UML опису, у трећем у АФД опису, а у четвртом проблему у UML опису. Сваки парни учесник је у првом проблему уочавао неконзистентности у UML опису, у другом у АФД опису, у трећем у UML опису, а у четвртом проблему у АФД опису. Сваки проблем имао је шест неконзистентности које је требало пронаћи. На овај начин је постигнута униформна расподела проблема, неконзистентности, као и архитектуралних језика за опис архитектура (АФД и UML) на учеснике експеримента. Учесници експеримента нису имали помоћ алата јер је дугме „Check consistency" за аутоматску детекцију неконзистентности било онемогућено. Приликом решавања проблема евидентирано је време почетка и време завршетка решавања проблема као и времена проналаска сваке неконзистентности. Решења учесника су оцењивана само на основу тога колико су неконзистентности у описима архитектура успешно пронашли. Оцене које су учесници постигли на проблему посматрањем АФД описа и оцене који су учесници постигли на истом проблему посматрањем UML описа су искоришћене за квантитативну анализу. Двосмерна анализа варијансе (ANOVA) и Kruskal–Wallis тестови су коришћени у сврхе анализе. Додатно у случају студената основних студија, посматрано је да ли постоје корелације између просечних оцена , оцена добијених на предмету ИС1, поена остварених на првом колоквијуму на предмету ИС1, поена остварених на другом колоквијуму на предмету ИС1 са оценама које су студенти постигли током експеримента.

7.3.2. Експеримент 3 – Резултати

Квантитативна евалуација је посматрала учеснике који су тражили неконзистентности у описима архитектура четири различита проблема. Решења за описе који су дефинисани у АФД су сматрана током евалуације експерименталним групама, а решења за описе који су дефинисани у UML су сматрани контролним групама. Преглед резултата учесника, који су запослени у струци, из сваке групе и проблема је дат у Табели 13.

Експерименталне групе постигле су више просечне оцене у односу на контролне групе за сваки осим другог проблема током периода посматрања, а за детаљнију анализу спроведен

Табела 13 Преглед резултата учесника запослених у струци при тражењу неконзистентности у АФД и UML описима различитих проблема.

Бр. проблема	Група	Бр. учесника	Просечна оцена на проблему (0-6)
1	АФД	5	5,6
	UML	5	3,6
2	АФД	5	4,0
	UML	5	4,4
3	АФД	5	5,4
	UML	5	4
4	АФД	5	4,8
	UML	5	4,2

Табела 14 Резултати двосмерног ANOVA теста за експеримент 3.

Фактори	Степени слободе	Средњи квадрат	F	Значајност
група	1	8,100	5,586	0,024
проблем	3	0,467	0,322	0,810
група * проблем	3	2,700	1,862	0,156
Грешка	32	1,450		

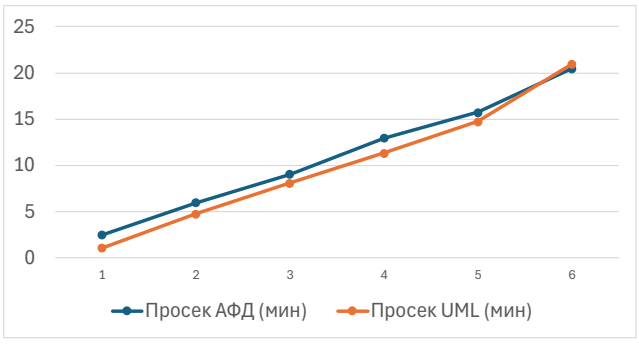
Табела 15 Резултати непараметарских независних узорака тестова Kruskal–Wallis за експеримент 3.

Бр. проблема	Степени слободе	Бр. учесника	Значајност
1	1	10	0,056
2	1	10	0,841
3	1	10	0,151
4	1	10	0.690

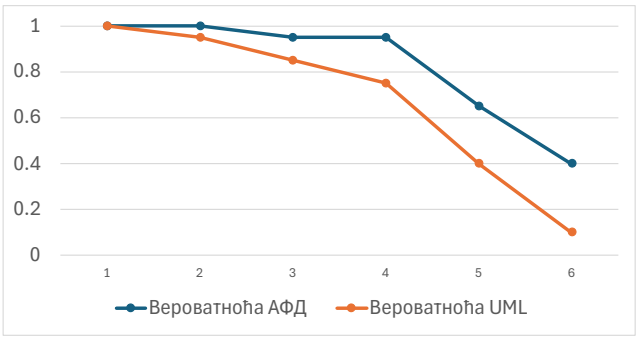
је двосмерни ANOVA тест. Први фактор у анализи био је посматрани проблем са четири нивоа (проблем 1, проблем 2, проблем 3 и проблем 4). Други фактор је био посматрана група са два нивоа (експериментална и контролна). Зависна променљива била је резултат учесника (резултујућа оцена на проблему) за посматрану групу и посматрани проблем. Резултати двосмерног ANOVA теста су дати у Табели 14.

Главни ефекат фактора посматране групе је статистички значајан, $F(1, 32) = 5,586$, $p = 0,024$, што сугерише да постоји разлика између експерименталних и контролних група, и као што је претходно наведено, експерименталне групе су постигле више просечне оцене. Главни ефекат фактора посматраног проблема није статистички значајан, $F(3, 32) = 0,322$, $p = 0,810$, што говори да не постоји општа разлика у постигнућима између типова проблема. Интеракција између два фактора није статистички значајна, $F(3, 32) = 1,862$, $p = 0,156$, што значи да у одређеним групама, посматрани проблем није могао утицати на резултате учесника. С обзиром да су просечне оцене учесника биле сличне оне нису уведене као трећи фактор. ANOVA претпоставке нису у потпуности испуњене, па су такође спроведени непараметарски Kruskal–Wallis тестови који су дали сличне резултате у погледу статистичке значајности као што је приказано у Табели 15. На основу резултата овог експеримента у случају испитаника запослених у струци утврђено је да је полазна хипотеза: „Коришћење новог архитектуралног језика може олакшати уочавање неконзистентности у опису архитектуре у поређењу са архитектуралним језиком који се тренутно користи у пракси“ тачна.

На основу евидентираних времена проналазака неконзистентности израчуната су просечна времена у проналажењу одређеног броја неконзистентности у примерима што је приказано на Слици 42. На Слици 42 се види да су времена проналажења неконзистентности слична у АФД и UML примерима. На основу евидентираних проналазака конзистентности израчунате су вероватноће проналажења одређеног броја неконзистентности у примерима што је приказано на Слици 43. На Слици 43 се види да је вероватноћа проналаска неконзистентности у АФД примерима већа у односу на вероватноћу проналаска у UML примерима, посебно у случају већег броја неконзистентности.



Слика 42 Време проналаска неконзистентности међу запосленима.

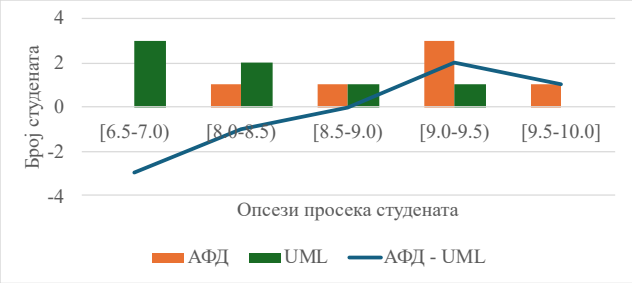


Слика 43 Вероватноћа проналаска неконзистентности међу запосленима.

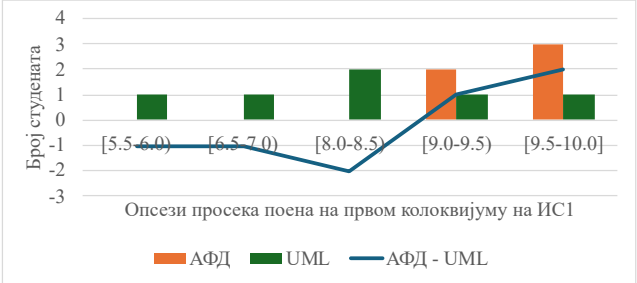
За потребе квантитативне евалуације резултата испитивања друге групе испитаника, односно студената основних студија такође су спроведени ANOVA и Kruskal–Wallis тестови. Ови тестови нису показали статистички значајну разлику. Даље је извршена анализа резултата испитаника на основу њихових просечних оцена (Слика 44), оцена добијених на предмету ИС1 (Слика 45), поена остварених на првом колоквијуму на предмету ИС1 (Слика 46), као и поена остварених на другом колоквијуму на предмету ИС1 (Слика 47). Оцене на предметима које су студенти могли да добију током студија су 5, 6, 7, 8, 9 и 10, док су поени на првом и другом колоквијуму на предмету ИС1 у опсегу од 0 до 10. На првом колоквијуму оцењивано је знање језика АФД, док је на другом колоквијуму оцењивано знање језика UML приликом решавања проблема.

За сваког студента је одређено да ли је бољи резултат постигао на проблемима посматрањем АФД или UML описа, сумирањем оцена на АФД примерима, сумирањем оцена на UML примерима и њиховим упоређивањем. Уколико је сума оцена на АФД примерима била већа од суме оцена на UML примерима студент је боље урадио АФД, уколико су суме биле једнаке студент је подједнако урадио АФД и UML, а уколико је сума оцена на АФД примерима била мања од суме оцена на UML примерима студент је боље урадио UML. На Сликама 44-47 вертикална оса представља број студената, наранџастом бојом приказано је колико је студената боље урадило АФД, зеленом бојом колико је студената боље урадило UML, док студенти који су подједнако урадили АФД и UML нису посебно приказивани. Плава линија представља разлику броја студената који су боље урадили АФД и броја студената који су боље урадили UML.

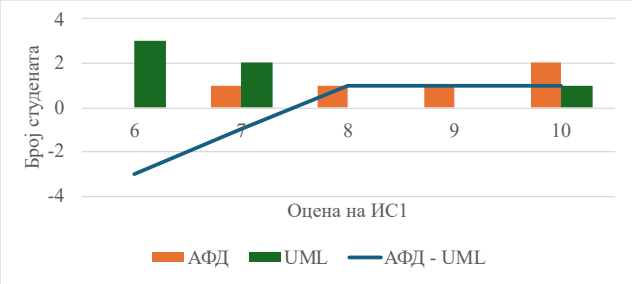
На Сликама 44, 45 и 46 уочено је да је међу студентима који су имали више просечне оцене, који су добили вишу оцену на ИС1, или су освојили више поена на првом колоквијуму из ИС1, било више оних који су боље урадили АФД од оних који су боље урадили UML. Међу студентима који су имали ниже просечне оцене, који су добили нижу оцену на ИС1, или су



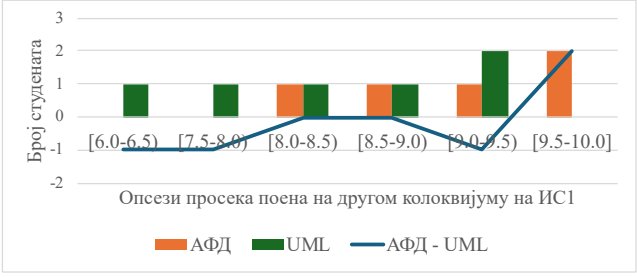
Слика 44 Однос просека студената и резултата.



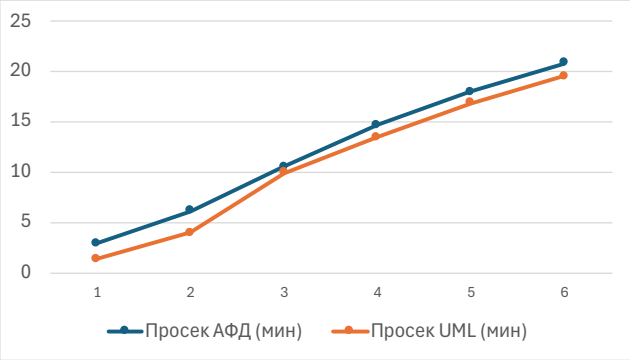
Слика 46 Однос поена студената на првом колоквијуму на ИС1 и резултата.



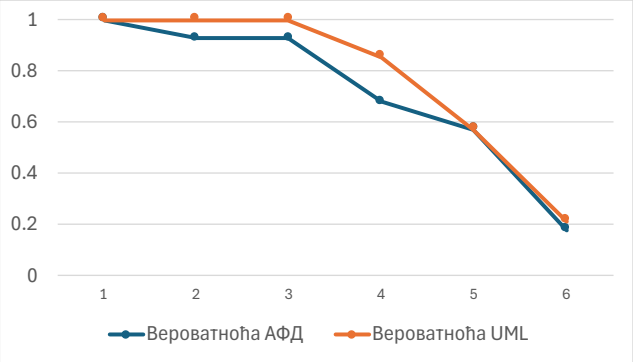
Слика 45 Однос оцене студената на ИС1 и резултата.



Слика 47 Однос поена на другом колоквијуму на ИС1 и резултата.



Слика 48 Време проналаска неконзистентности међу студентима.



Слика 49 Вероватноћа проналаска неконзистентности међу студентима.

освојили мање поена на првом колоквијуму из ИС1, било је више оних који су боље урадили UML од оних који су боље урадили АФД. На тај начин уочена је корелација између просечне оцене студената, оцене на предмету ИС1, поена на првом колоквијуму са резултатима студената. На Слици 47 није уочена јасна корелација између поена остварених на другом колоквијуму и резултата студената. На основу резултата овог експеримента у случају испитаника који су студенти основних студија полазна хипотеза: „Коришћење новог архитектуралног језика може олакшати уочавање неконзистентности у опису архитектуре у поређењу са архитектуралним језиком који се тренутно користи у пракси“ није потврђена, али је примећено да су АФД боље урадили више успешни студенти, док су UML боље урадили мање успешни студенти. Овакав резултат може бити из разлога што је за АФД била доступна мања количина материјала у односу на UML, што је више успешним студентима било довољно за припрему, док осталима није.

На исти начин као у случају учесника који су запослени, за студенте основних студија израчуната су просечна времена у проналажењу одређеног броја неконзистентности што је приказано на Слици 48, као и вероватноће проналажења одређеног броја неконзистентности у примерима што је приказано на Слици 49. На Слици 48 се види да су времена проналажења неконзистентности слична у АФД и UML примерима. На Слици 49 се види да је вероватноћа проналаска једне, две, три или четири неконзистентности у UML примерима већа у односу на вероватноћу проналаска у АФД примерима, док су вероватноће сличне за проналажење пет и шест неконзистентности.

7.4. Одговори на полазне хипотезе

Полазне хипотезе које су уведене у четвртом поглављу приликом дефинисања проблема су тестиране спроведеним евалуацијама. Одговори на полазне хипотезе претходно су дати у оквиру описа евалуација. У наставку ће бити поново набројане све полазне хипотезе и дати одговори на њих.

Полазна хипотеза: „Могуће је предложити нов архитектурални језик који би подржао све принципе рачунарског размишљања.“ је тачна јер архитектурални језик АФД имплементира све принципе рачунарског размишљања што је показано у потпоглављу 5.1. и додатно је споменуто у евалуацији у контексту других архитектуралних језика. Полазна хипотеза: „Могуће је користити интегрално моделирање за креирање описа архитектуре

софтверског система кроз више погледа.“ је тачна јер архитектурални језик АФД користи интегрално моделовање за креирање описа архитектуре софтверског система кроз више погледа што је показано у потпоглављу 5.2 као и у евалуацији у контексту других архитектуралних језика. Полазна хипотеза: „Опис креиран предложеним архитектуралним језиком је могуће аутоматски превести у одговарајући опис за архитектурални језик који се тренутно користи у пракси.“ је тачна јер архитектурални језик АФД пружа могућност превођења својих описа у UML описе архитектура што је показано у потпоглављима 5.3, 6.2 и 6.3 и додатно је споменуто у евалуацији у контексту других архитектуралних језика. Полазна хипотеза: „Коришћење новог архитектуралног језика може студентима олакшати креирање описа архитектуре у поређењу са архитектуралним језиком који се тренутно користи у пракси.“ је тачна на основу резултата првог експеримента. Полазна хипотеза: „Коришћење новог архитектуралног језика може студентима бити лак за разумевање у поређењу са архитектуралним језиком који се тренутно користи у пракси.“ је тачна на основу резултата другог експеримента. Полазна хипотеза: „Коришћење новог архитектуралног језика може олакшати уочавање неконзистентности у опису архитектуре у поређењу са архитектуралним језиком који се тренутно користи у пракси“ је на основу резултата трећег експеримента тачна у случају испитаника запослених у струци, док у осталим случајевима није потврђена.

7.5. Претње валидности за сва три експеримента

Две могуће претње валидности могу се идентификовати у описаном методу квантитативне евалуације првог експеримента, једна екстерна и једна интерна. Екстерна претња, наиме интерференција вишеструког третмана, је последица чињенице да студенти прво уче АФД и користе га на првом колоквијуму, а затим уче UML и користе га на другом колоквијуму. То значи да у тренутку полагања другог колоквијума, студенти већ поседују одређена предзнања о логичком дизајну информационих система. Слично томе, унутрашња претња, наиме сазревање, огледа се у чињеници да полагањем првог колоквијума, студенти стичу одређено искуство везано за начин на који су проблеми наведени и оцењени и тако знају на шта да обрате пажњу на другом колоквијуму [10].

Једна унутрашња претња валидности може бити идентификована у описаној методи квалитативне анализе другог експеримента. Та претња, наиме експериментална смртност, постоји због чињенице да је учешће у анкети било на добровољној бази. Дакле, могло би се претпоставити да су само студенти који су били задовољни курсом и постигнутим оценама одлучили да учествују у анкети [10].

Једна екстерна претња валидности може бити идентификована у описаној методи квантитативне анализе трећег експеримента. Екстерна претња, наиме мали узорак, постоји због чињенице да је тестирање обавило 10 учесника запослених у струци и 14 учесника студената. То значи да резултат не може у потпуности да се генерализује.

8. Смернице за даља истраживања

На основу евалуације спроведене у контексту других архитектуралних језика АФД пружа 9 од 12 захтева и 13 од 20 под-захтева за архитектуралне језике. Захтеви које АФД не пружа су формална семантика, верзионисање и колаборација, а под-захтеви визуелна нотација, генератор софтверског кода, комплетност, коректност, компатибилност, синхрона колаборација и асинхрона колаборација.

Семантика АФД језика може бити формално допуњена како би се математички описала. За ове потребе може се урадити интеграција постојећих формалних метода или формалних језика у АФД. Тако би било омогућено креирање детаљних и прецизних описа архитектура које се даље могу исцрпно анализирати користећи математичке доказе. Други начин који би могао да пружи овакве анализе је проширивање АФД одређеним скупом анотација како би се омогућило мапирање и превођење у моделе формалних језика. Анализе формалних описа могу се омогућити проширивањем АФД алата новим функционалностима и алгоритмима, или користећи већ постојеће алате развијене специфично за те потребе. Даље истраживање би могло обухватити процену погодности интегралног модела и функционалне декомпозиције за креирање формалних описа.

Различите верзије функција дефинисаних у АФД могу се чувати на репозиторијуму тако да се могу одатле касније преузети како би се поново користиле. Верзионисање функција могло би се обавити посебно верзионисањем елемената других погледа који анотирају функције. На овај начин би се у одређеном опису могле користити различите верзије имплементације за исту функцију. Верзионисање би се постигло проширивањем језика новим анотацијама и имплементацијом посебног или коришћењем постојећих репозиторијума у оквиру система за верзионисање кода.

Системи за верзионисање кода могу се искористити као средство за асинхрону колаборацију којом би различити практиканти могли читаве описе архитектура да постављају на репозиторијуме у различито време. Синхрона колаборација би захтевала додатно проширивање АФД алата како би се обезбедило да се колаборација обавља синхронизовано и у исто време или интеграција АФД алата са постојећим решењима за синхрону колаборацију. Такође, АФД алат би требало да омогући практикантима креирање налога као и могућност мапирања делова описа на налоге. Даље истраживање би се могло бавити утицајем верзионисања и колаборације, као и интегралног модела на ефикасност и квалитет описивања архитектура [2].

Визуелна нотација тренутно није у фокусу с обзиром да се интегрални модел заснива на текстуалном опису. Међутим, пружање визуелне нотације могло би омогућити да се интегрални модел приближи широм аудиторијуму, посебно практиканата са не толико искуства [2]. Даље истраживање би могло испитати да ли интегрални модел може бити представљен визуелно, као и потенцијалне ефекте које би визуелна нотација постигла.

Софтверски код може се генерисати из UML дијаграма, добијених превођењем интегралног модела, коришћењем постојећих алата који раде са UML. Са друге стране, софтверски код би се могао генерисати директно из интегралног модела проширењем алата одређеним алгоритмима или другим методама. Генерисање кода директно из интегралног

модела у виду програмског скелета који би се даље проширивао могло би да допринесе даљој интеграцији АФД у процес развоја софтвера, посебно интеграцију са фазом имплементације.

Функционална декомпозиција над којом се АФД заснива иако је потенцијално погодна за адресирање захтева за системом њеним функцијама, не дефинише прецизно када треба стати са декомпозицијом, односно да ли је ниво детаља довољан да комплетно опише систем [2]. Над интегралним моделом се не испитује да ли одређене одлуке у опису задовољавају жељене особине система, као што је постојање застоја, односно АФД не пружа анализу коректности описа. Додатно, не постоји могућност провере да ли су одлике које се доносе у креирању описа компатибилне са одређеним архитектуралним стиловима или смерницама за дизајн. Даље истраживање може ићи у смеру проналажења метода и техника за омогућавање преостала три циља анализе: комплетности, коректности и компатибилности.

Поред основних циљева за анализом, може се радити на пружању могућности за симулацију реалног софтверског система како би се понашање и интеракције у оквиру система могле анализирати и пре имплементације. Поред симулација које би пружиле анализу појединих случајева употребе, могле би се омогућити исцрпне провере свих стања у којима би се систем могао наћи на основу претходно формално описане архитектуре. Могла би се спровести додатна анализа дефинисаних стања ради идентификовања стања у које систем на основну свог понашања не може ући, па тако открити грешке још у почетној фази развоја. Слично, АФД се може даље унапредити претрагом функција ради идентификовања оних које се никада неће користити. Употребљивост АФД могла би се повећати интеграцијом са постојећим језицима или креирати посебне под-језике ради дефинисања захтева практиканата за системом којим би могле да се опишу додатне захтеване особине чија би се испуњеност касније могла проверити. Ради провере наведених претпоставки потребно је наставити развој АФД и спровести одговарајуће експерименте.

Додатна истраживања се могу спровести и над захтевима које АФД већ пружа. Могућа је имплементација других техника и метода за проширивање језика као на пример коришћење посебних под-језика за проширење скупа гледишта. Проширење скупа гледишта на овај начин могло би повећати употребу АФД од стране већег броја практиканата. Употребљивост у фази имплементације потенцијално би се повећала креирањем посебних додатака у оквиру постојећих интегрисаних развојних окружења који би омогућили генерисање кода за нове пројекте. Са друге стране реверзни поступак, односно генерисање АФД описа на основу постојеће имплементације допринело би сагледавању имплементираног система са више гледишта и спровођење свих анализа над генерисаним описом. Додатно, пружање оба поступка, односно генерисање кода из описа и генерисање описа из кода омогућило би спрегнути развој описа и имплементације који би могао имати позитивне ефекте на читав процес развоја.

Заснованост АФД на функционалној декомпозицији и имплементација рачунарског размишљања допринеле су у решавању логичких проблема и олакшаном разумевању АФД. Истраживање у овом домену може довести до проналаска нових методологија које би биле имплементирале у АФД. Њихова имплементација може довести до даљих унапређења при решавању проблема као и управљању описима. Додатно, могу се пронаћи још неки случајеви употребе АФД, као што је употреба за постепено прикупљање захтева корисника током креирања описа [75].

Спроведен експеримент ради испитивања да ли АФД може олакшати проналажење неконзистентности имао је мали узорак. Над АФД се може спровести исти експеримент ради проналажења неконзистентности на већем броју учесника. На овај начин би се повећао узорак и резултати би се могли генерализовати.

9. Закључак

Многи студенти током њихових студија инжењерства уче да примењују различите техничке алате и методе. Међутим, остаје општи недостатак способности да се схвате сви аспекти и логика комплексних проблема, као што су они који су укључени у дизајнирање и имплементацију информационих система. Једно могуће решење може бити учење студената да примене резоновање засновано на рачунарском размишљању које укључује мисаоне процесе за формулацију проблема и за ефикасно изражавање решења. Ова дисертација показала је нови приступ заснован на текстуалном језику АФД који дефинише проширив скуп анотација за имплементацију декомпозиције, препознавања образаца, апстракције, и алгоритама, што су четири стуба рачунарског размишљања. АФД се користи помоћу АФД алата који је имплементиран као веб апликација која обавља синтаксне и семантичке провере, као и генерисање UML дијаграма.

Поред коришћења за потребе оспособљавања за схватањем логике комплексних система АФД је архитектурални језик за потребе описа архитектура комплексних софтверских система. Током година различити архитектурални језици су креирани да опишу комплексне софтверске системе. Велики број доступних језика описују архитектуре кроз више гледишта на тај начин служећи потребама заинтересованих страна. Иако коришћење гледишта помаже у суочавању са комплексним системима, јавља се проблем конзистенције између погледа креираних у складу са гледиштима. Архитектурални језик АФД описан у овој дисертацији пружа интегрално моделовање које олакшава решавање овог проблема. Интегрално моделовање подржава гледишта и омогућава креирање погледа једне уз друге и тако олакшава управљање конзистенцијом. Штавише, интегрални модели креирани у АФД могу бити преведени у одговарајуће UML дијаграме ради даљег дизајна и анализе.

АФД обезбеђује сличне конструкте који су доступни у другим језицима заснованим на функционалној декомпозицији, као што је FFBD и његовим проширењима, док такође укључује имплементационе аспекте проблема погодне за генерисање UML дијаграма из текстуалног описа система. Међутим, за разлику од других текстуалних алата за UML моделовање, АФД уводи методолошки приступ за решавање проблема. Како би се проценило да ли АФД олакшава креирање описа архитектуре у поређењу са архитектуралним језиком који се тренутно користи у пракси односно олакшава решавање проблема у домену логичког дизајна информационих система и како студенти перципирају процес, спроведене су квантитативне и квалитативне евалуације употребе АФД током курса информационих система. Квантитативна евалуација је открила да су студенти који су користили АФД постигли више просечне оцене од оних који су користили UML за решавање истих проблема. Иако је статистичка анализа показала да је разлика између просечних оцена статистички значајна, овај ефекат може бити резултат других скривених фактора. Квалитативна евалуација је сугерисала да две трећине студената сматра да је АФД лак за разумевање за употребу током решавања проблема и у поређењу са UML, тако да АФД може студентима бити лак за разумевање у поређењу са архитектуралним језиком који се тренутно користи у пракси. Поред тога, више од једне трећине студената изразило је оптимизам у погледу примењивости АФД у пракси, иако то није широко препозната техника као што су други мислили.

Додатна евалуација спроведена у овој дисертацији према захтевима коришћеним у литератури за евалуацију 124 архитектурална језика, показала је да АФД пружа 9 од 12 захтева. Како АФД имплементира рачунарско размишљање и уводи интегрално моделовање за креирање више погледа, показано је да је могуће предложити нов архитектурални језик који би подржао све принципе рачунарског размишљања и који користити интегрално моделирање за креирање описа архитектуре софтверског система кроз више погледа. АФД додатно омогућава и аутоматско генерисање UML дијаграма, чиме је показано да је могуће опис креиран предложеним архитектуралним језиком аутоматски превести у одговарајући опис за архитектурални језик који се тренутно користи у пракси. Последње, спроведена је квантитативна анализа у контексту примене АФД у управљању конзистенцијом. Квантитативна анализа је показала да АФД олакшава уочавање неконзистентности у поређењу са UML у случају особа које су запослене, односно имају више искуства, чиме је показано да коришћење новог архитектуралног језика може олакшати уочавање неконзистентности у опису архитектуре у поређењу са архитектуралним језиком који се тренутно користи у пракси.

Литература

[1] M. Ozkaya and F. Erata, "A Survey on the Practical Use of UML for Different Software Architecture Viewpoints," *Inf. Softw. Technol.*, vol. 121, no. C, pp. 106275, May 2020. DOI: 10.1016/j.infsof.2020.106275.

[2] M. Ozkaya, "The analysis of architectural languages for the needs of practitioners," *Softw. Pract. Exp.*, vol. 48, no. 5, pp. 985-1018, Jan. 2018. DOI: 10.1002/spe.2561.

[3] P. Chao, "Exploring students' computational practice, design and performance of problem-solving through a visual programming environment," *Comput. Educ.*, vol. 95, pp. 202-215, 2016.

[4] A. Pears et al., "A survey of literature on the teaching of introductory programming," *ACM SIGCSE Bull.*, vol. 39, no. 4, pp. 204-223, Dec. 2007.

[5] S. Xinogalos, "An evaluation of knowledge transfer from microworld programming to conventional programming," *J. Educ. Comput. Res.*, vol. 47, no. 3, pp. 251-277, 2012.

[6] A. Labusch et al., eds., "Chapter 5: Computational thinking processes and their congruence with problem-solving and information processing," in *Computational Thinking Education*, Singapore: Springer, 2019.

[7] J. M. Wing, "Computational thinking," *Commun. ACM*, vol. 49, no. 3, pp. 33-35, Mar. 2006. DOI: 10.1145/1118178.1118215.

[8] C. Brackmann et al., "Computational thinking: Panorama of the Americas," in *Int. Symp. Comput. Educ. (SIIE)*, 2016, pp. 1-6.

[9] Association for Computing Machinery (ACM) Joint Task Force on Computing Curricula and IEEE Computer Society, "Computer science curricula 2013: Curriculum guidelines for undergraduate degree programs in computer science," November 2020. [Online]. Available: https://www.acm.org/binaries/content/assets/education/cs2013_web_final.pdf

[10] S. Tubić, M. Cvetanović, Z. Radivojević, and S. Stojanović, "Annotated functional decomposition," *Comput. Appl. Eng. Educ.*, vol. *29*, no. *5*, pp. *1390–1402*, Mar, 2021, DOI: 10.1002/cae.22394.

[11] J. Kwon and J. Kim, "A study on the design and effect of computational thinking and software education," *KSII Trans. Internet Inf. Syst.*, vol. 12, no. 8, pp. 4057-4071, 2018.

[12] J. M. Wing, "Computational thinking's influence on research and education for all," *Ital. J. Educ. Technol.*, vol. 25, no. 2, pp. 7-14, 2017.

[13] *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*, N. Rozanski, and E. Woods, 1st ed. Upper Saddle River, New Jersey, United States: Addison-Wesley Professional, 2005.

[14] ISO/IEC/IEEE, "Systems and software engineering - architecture description," ISO/IEC/IEEE 42010:2011(E), 2011.

[15] IEEE, "IEEE Recommended Practice for Architectural Description for Software-Intensive Systems," IEEE Std 1471-2000, 2000.

[16] A. Cicchetti, F. Ciccozzi, and A. Pierantonio, "Multi-view approaches for software and system modelling: a systematic literature review," *Softw. Syst. Model.*, vol. 18, no. 6, pp. 3207-3233, Dec. 2019. DOI: 10.1007/s10270-018-00713-w.

[17] D. Soni, R. L. Nord, and C. Hofmeister, "Software architecture in industrial applications," in *Proc. 17th Int. Conf. on Software Engineering*, Seattle, WA, USA, 1995, pp. 196-207.

[18] P. Kruchten, "The 4+1 View Model of Architecture," *IEEE Softw.*, vol. 12, no. 6, pp. 45-50, Nov. 1995. DOI: 10.1109/52.469759.

[19] *Documenting Software Architectures: Views and Beyond*, P. Clements, D. Garlan, L. Bass, J. Stafford, R. Nord, J. Ivers, R. Little, 1st ed. Boston, Massachusetts, United States: Addison-Wesley Professional, 2002.

[20] *Large-Scale Software Architecture: A Practical Guide using UML*, J. Garland, and R. Anthony, 1st ed. Hoboken, New Jersey, United States: Wiley Publishing, 2002.

[21] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*, 1st ed. Hoboken, NJ, USA: Wiley, 2009.
[22] J. R. Anderson, *How Can the Human Mind Occur in the Physical Universe?*, New York, NY, USA: Oxford Univ. Press, 2010.
[23] W. Hussy, "Denken und Problemlösen" [Thinking and problem-solving], 2nd ed. Stuttgart, Germany: Kohlhammer, 1998.
[24] E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, 1976. Available: http://www.worldcat.org/oclc/01958445
[25] J. Jiang, H. Zhu, Q. Li, Y. Zhao, S. Zhang, P. Gong, and Z. Hong, "Event-based functional decomposition," *Inf. Comput.*, vol. 271, pp. 1-22, 2020.
[26] E. Brimhall et al., "A systematic process for functional decomposition in the absence of formal requirements," in *INCOSE Int. Symp.*, vol. 26, 2016, pp. 1204-1218.
[27] V. Chourey and M. Sharma, "Functional flow diagram (FFD): Semantics for evolving software," in *Proc. Int. Conf. Adv. Comput. Commun. Informatics (ICACCI)*, 2016, pp. 2193-2199.
[28] G. Estrin et al., "SARA (System ARchitects Apprentice): modeling, analysis, and simulation support for design of concurrent systems," *IEEE Trans. Softw. Eng.*, vol. SE-12, no. 2, pp. 293-311, Feb. 1986. Available: http://dl.acm.org/citation.cfm?id=9794.9805; https://doi.org/10.1109/TSE.1986.6312945
[29] D. Harel et al., "STATEMATE: A working environment for the development of complex reactive systems," presented at *Proc. 10th Int. Conf. Software Eng.*, April 11-15, 1988, Singapore, Singapore. Available: http://dl.acm.org/citation.cfm?id=55861
[30] R. Allen and D. Garlan, "A formal basis for architectural connection," *ACM Trans. Softw. Eng. Methodol.*, vol. 6, no. 3, pp. 213-249, Jul. 1997.
[31] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer, "Specifying distributed software architectures," in *European Software Engineering Conference*, W. Schäfer and P. Botella, Eds. New York, NY: Springer, 1995, pp. 137-153. Lecture Notes in Computer Science, vol. 989.
[32] D. C. Luckham et al., "Specification and analysis of system architecture using Rapide," *IEEE Trans. Software Eng.*, vol. 21, no. 4, pp. 336-355, Apr. 1995.
[33] P. H. Feiler, B. A. Lewis, and S. Vestal, "The SAE Architecture Analysis & Design Language (AADL) a standard for engineering performance critical systems," presented at *2006 IEEE Int. Symp. Intell. Control*, Munich, Germany, 2006. Available: https://doi.org/10.1109/CACSD-CCA-ISIC.2006.4776814
[34] Z. Yu, Y. Cai, R. Wang, and J. Han, "π-net ADL: an architecture description language for multi-agent systems," in *International Conference on Intelligent Computing*, D. Huang, X. Zhang, and G. Huang, Eds. New York, NY: Springer, 2005, pp. 218–227. Available: https://doi.org/10.1007/11538356_23
[35] C. Mascolo, "MobiS: a specification language for mobile systems," in *Coordination Languages and Models, Third Int. Conf. COORDINATION'99*, P. Ciancarini and A. L. Wolf, Eds. New York, NY: Springer, 1999, pp. 37–52. Available: https://doi.org/10.1007/3-540-48919-3_5
[36] I. Malavolta et al., "What Industry Needs from Architectural Languages: A Survey," *IEEE Trans. Softw. Eng.*, vol. 39, no. 6, pp. 869-891, Jun. 2013. DOI: 10.1109/TSE.2012.74.
[37] P. Lago, I. Malavolta, and H. Muccini, "The Road Ahead for Architectural Languages," *IEEE Softw.*, vol. 32, no. 1, pp. 98-105, Jan. 2014. DOI: 10.1109/MS.2014.28.
[38] R. Milner, J. Parrow, and D. Walker, "Calculus of mobile processes, I," *Inf. Comput.*, vol. 100, no. 1, pp. 1-40, 1992.

[39] B. Meyer, "Applying ‘design by contract’," *IEEE Comput.*, vol. 25, no. 10, pp. 40-51, Oct. 1992.

[40] M. Richters and M. Gogolla, "OCL: Syntax, semantics, and tools," in *Object Modeling with the OCL, The Rationale behind the Object Constraint Language*, T. Clark and J. Warmer, Eds. New York, NY: Springer, 2002, pp. 42-68. Lecture Notes in Computer Science, vol. 2263. Available: https://doi.org/10.1007/3-540-45669-4_4

[41] K. Dunsire, T. O'Neill, M. Denford, and J. Leaney, "The ABACUS architectural approach to computer-based system and enterprise evolution," presented at *12th IEEE Int. Conf. Workshops Eng. Comput.-Based Syst. (ECBS'05)*, Greenbelt, MD, 2005. Available: https://doi.org/10.1109/ECBS.2005.66

[42] B. Magableh and S. Barrett, "Primitive component architecture description language," presented at *7th Int. Conf. Informatics Syst. (INFOS)*, Cairo, Egypt, 2010.

[43] V. Gruhn and C. Schäfer, "Architecture description for mobile distributed systems using typed π-calculus," *Electr. Notes Theor. Comput. Sci.*, vol. 150, no. 1, pp. 51-60, 2006. Available: https://doi.org/10.1016/j.entcs.2005.12.023

[44] D. Balasubramanian et al., "DREMS ML: a wide spectrum architecture design language for distributed computing platforms," *Sci. Comput. Program.*, vol. 106, pp. 3-29, 2015. Available: https://doi.org/10.1016/j.scico.2015.04.002; http://www.sciencedirect.com/science/article/pii/S0167642315000684.

[45] L. Zheng, Z. Wu, C. Zhang, and F. Yang, "Developing an architecture description language for data processing product line," presented at *2010 7th Int. Conf. Inf. Technol.: New Generations (ITNG)*, Las Vegas, NV, 2010. Available: https://doi.org/10.1109/ITNG.2010.25

[46] A. Amirat and M. C. Oussalah, "First-class connectors to support systematic construction of hierarchical software architecture," *J. Object Technol.*, vol. 8, no. 7, pp. 107-130, Nov. 2009. Available: https://doi.org/10.5381/jot.2009.8.7.a3; http://www.jot.fm/contents/issue_2009_11/article3.html

[47] T. Murata, "Petri nets: properties, analysis and applications," *Proc. IEEE*, vol. 77, no. 4, pp. 541-580, Apr. 1989. Available: https://doi.org/10.1109/5.24143

[48] G. Su, M. Ying, and C. Zhang, "An ADL-approach to specifying and analyzing centralized-mode architectural connection," *Software Architecture*, New York, NY: Springer, 2010, pp. 8-23. Available: https://doi.org/10.1007/978-3-642-15114-9_4

[49] Q. Wu and Y. Li, "ScudADL: an architecture description language for adaptive middleware in ubiquitous computing environments," presented at *2009 ISECS Int. Colloq. Comput. Commun. Control Manage.*, Sanya, China, 2009. Available: https://doi.org/10.1109/CCCM.2009.5267500

[50] S. Becker, H. Koziolek, and R. Reussner, "The Palladio component model for model-driven performance prediction," *J. Syst. Softw.*, vol. 82, no. 1, pp. 3-22, 2009. Available: https://doi.org/10.1016/j.jss.2008.03.066; http://www.sciencedirect.com/science/article/pii/S0164121208001015.

[51] N. Ali, I. Ramos, and C. Solís, "Ambient-PRISMA: ambients in mobile aspect-oriented software architecture," J. Syst. Softw., vol. 83, no. 6, pp. 937-958, 2010. Available: https://doi.org/10.1016/j.jss.2009.12.009; http://www.sciencedirect.com/science/article/pii/S0164121209003161.

[52] B. Schmerl, "xAcme: CMU Acme extensions to xArch," Mar. 23, 2001. [Online]. Available: http://www.cs.cmu.edu/~acme/pub/xAcme/. Accessed: May. 17, 2024.

[53] A. Haber, J. O. Ringert, and B. Rumpe, "MontiArc-Architectural Modeling of Interactive Distributed and Cyber-Physical Systems," Technical Report, RWTH Aachen Univ., Aachen, Germany, Feb. 2012. Available: http://aib.informatik.rwth-aachen.de/2012/2012-03.pdf

[54] M. Pinto, L. Fuentes, and J. M. Troya, "Specifying aspect-oriented architectures in AO-ADL," *Inf. Softw. Technol.*, vol. 53, no. 11, pp. 1165-1182, 2011. Available: https://doi.org/10.1016/j.infsof.2011.04.003; http://www.sciencedirect.com/science/article/pii/S0950584911001005.

[55] M. Ozkaya and C. Kloukinas, "Design-by-contract for reusable components and realizable architectures," in *Proc. 17th Int. ACM SIGSOFT Symp. Component-Based Softw. Eng. (CBSE'14), part of CompArch 2014*, L. Seinturier, E. S. de Almeida, and J. Carlson, Eds., Marcq-en-Baroeul, Lille, France, June 30-July 4, 2014, New York, NY, 2014. Available: http://doi.acm.org/10.1145/2602458.2602463

[56] M. Hamdaqa and L. Tahvildari, "Stratus ML: A layered cloud modeling framework," presented at *2015 IEEE Int. Conf. Cloud Eng. (IC2E)*, Tempe, AZ, 2015. Available: https://doi.org/10.1109/IC2E.2015.42

[57] V. Debruyne, F. Simonot-Lion, and Y. Trinquet, "EAST-ADL – an architecture description language," in *Architecture Description Languages*, New York, NY: Springer, 2005, pp. 181-195. Available: https://doi.org/10.1007/0-387-24590-1_12

[58] R. Weinreich and G. Buchgeher, "Paving the road for formally defined architecture description in software development," presented at *Proc. 2010 ACM Symp. Appl. Comput. (SAC'10)*, New York, NY, 2010. Available: https://doi.org/10.1145/1774088.1774571

[59] J. Rumbaugh, I. Jacobson, and G. Booch, *Unified Modeling Language Reference Manual*, 2nd ed. London, UK: Pearson Higher Educ., 2004.

[60] K. Lau and C. M. Tran, "X-MAN: an MDE tool for component-based system development," in *Proc. 38th Euromicro Conf. Softw. Eng. Adv. Appl. (SEAA)*, V. Cortellessa, H. Muccini, and O. Demirörs, Eds., Izmir, Turkey, Sep. 5-8, 2012. Available: https://doi.org/10.1109/SEAA.2012.32

[61] R. France and B. Rumpe, "Model-based development," *Softw. Syst. Model.*, vol. 7, no. 1, pp. 1–2, 2008.

[62] G. Spanoudakis and A. Zisman, "Inconsistency Management in Software Engineering: Survey and Open Research Issues," in *Handbook of Software Engineering and Knowledge Engineering*, 1st ed. Singapore: World Scientific Publishing Co. Pte. Ltd., 2001, ch. 15, pp. 329-380.

[63] S. I. Weiss, *Product and Systems Development*, Hoboken, NJ: John Wiley & Sons, Inc., 2013, pp. 41-46.

[64] A. Battigaglia, E. Brimhall, and T. Ogle, "Conceptual data modeling for the functional decomposition of mission capabilities," in *INCOSE Int. Symp.*, vol. 27, 2017, pp. 935-949.

[65] S. Lizy-Destrez et al., "xFFBD: Towards a formal yet functional modeling language for system designers," in *Proc. INCOSE Int. Symp.*, 2012, pp. 170-183.

[66] Q. Li, "Computational thinking and teacher education: An expert interview study," *Hum. Behav. Emerg. Technol.*, vol. early access, pp. 1-15, 2020.

[67] L. G. Williams and C. U. Smith, "Performance evaluation of software architectures," in *Proc. 1st Int. Workshop on Software and Performance*, Santa Fe, NM, USA, 1998, pp. 164-177.

[68] E. Jagroep et al., "Extending software architecture views with an energy consumption perspective," *Computing*, vol. 99, no. 6, pp. 553-573, Jun. 2017. DOI: 10.1007/s00607-016-0502-0.

[69] M. Gogolla, F. Buttner, and M. Richters, "USE: A UML-based specification environment for validating UML and OCL," *Sci. Comput. Program.*, vol. 69, no. 1, pp. 27-34, 2007.
[70] M. H. Orabi, A. H. Orabi, and T. C. Lethbridge, "A textual notation for modeling and generating code for composite structure," in *Proc. 6th Int. Conf. MODELSWARD*, 2019, pp. 355-379.
[71] D. Spinellis, "On the declarative specification of models," *IEEE Softw.*, vol. 20, no. 2, pp. 94-96, Mar./Apr. 2003.
[72] PlantUML, "PlantUML Language Reference Guide." [Online]. Available: https://plantuml.com/. Accessed: May. 10, 2024.
[73] CodeMirror, "CodeMirror: In-browser code editor." [Online]. Available: https://codemirror.net/. Accessed: May. 12, 2024.
[74] N. Medvidovic, "A classification and comparison framework for software architecture description languages," *IEEE Trans. Softw. Eng.*, vol. 26, no. 1, pp. 70-93, Jan. 2000. DOI: 10.1109/32.825767.
[75] M. Cvetanović, Z. Radivojević, and S. Tubić, “An approach for software design and development,” in *Proc. SINTEZA 2022*, Belgrade, Serbia, Apr. 2022, pp. 154-162, DOI:10.15308/Sinteza-2022-154-162.

Биографија аутора

Стефан Тубић рођен је 25.08.1991. године у Крагујевцу. Завршио је основну школу Мирко Јовановић 2006. године у Крагујевцу са просечном оценом 5.00, као ђак генерације и носилац дипломе Вук Караџић и добијао награде на такмичењима из математике и физике. Затим 2010. године завршава Прву крагујевачку гимназију, унутар одељења обдарених за математику, са просечном оценом 5.00 где је учествовао на такмичењима из информатике, математике и физике.

Основне академске студије на Електротехничком факултету у Београду уписао је 2010. године и изабрао Одсек за рачунарску технику и информатику. Током основних студија радио је као демонстратор на више предмета при Катедри за рачунарску технику и информатику. Дипломирао је на студијском програму Електротехника и рачунарство 2014. године, након четири године студија, са просечном оценом 9.00 и оценом 10 на дипломском испиту. Тема дипломског рада је „Примена алгоритама у компјутерским играма“, а ментор рада је проф. др Зоран Јовановић. Стефан је био на пракси у фирми Intellex d.o.o. где је радио на развоју iOS апликација.

Мастер академске студије на Електротехничком факултету у Београду уписао је 2014. године на Одсеку за рачунарску технику и информатику. Мастер студије је завршио 2016. године са просечном оценом 9.83 и оценом 10 на мастер раду. Тема мастер рада је „Имплементација софтвера за филтрирање нежељених порука употребом класификационих алгоритама“, а ментор рада је проф. др Милош Цветановић.

Докторске академске студије на Електротехничком факултету у Београду уписао је 2016. године на модулу за Рачунарску технику и информатику. До сада је остварио 120 ЕСПБ, односно положио све испите предвиђене студијским програмом са оценом 10. У току докторских студија објавио је један рад у међународном часопису са импакт фактором, један рад на међународној конференцији и два рада на домаћим конференцијама. Од 2019. до 2023. Стефан Тубић је ангажован на међународном пројекту Уједињених нација програма за развој под називом: „Пројекат за преквалификације у IT сектору”. Од 2016. до 2022. је ангажован на пројекту Министарства науке, технолошког развоја и иновација бр. TR32047 под називом: „Развој хардверске, софтверске и телекомуникационе инфраструктуре е-система за контролу промета и пореза”. Код Министарства науке, технолошког развоја и иновација био је ангажован на још 4 пројекта. Био је ангажован и на једном комерцијалном пројекту.

Од марта 2015. године запослен је као сарадник у настави на Катедри за рачунарску технику и информатику Електротехничког факултета у Београду. Од априла 2017. године запослен је као асистент на Катедри за рачунарску технику и информатику Електротехничког факултета у Београду. Био је ангажован у настави на 7 различитих предмета, који се изводе на оба студијска програма, Електротехника и рачунарство и Софтверско инжењерство и изводи вежбе на табли и лабораторијске вежбе за изузетно велики број студената, на све четири године основних студија, као и на мастер студијама. Као асистент на Електротехничком факултету био је ангажован на следећим предметима: Базе података 1, Информациони системи 1, Информациони системи 2, Софтверски алати база података, Софтверско инжењерство великих база података, Рачунарске мреже 1, Програмирање 1. Говори течно енглески језик.

Изјава о ауторству

Име и презиме аутора: Стефан Тубић

Број индекса: 2016/5008

Изјављујем

да је докторска дисертација под насловом

Језик за опис архитектуре софтверских система

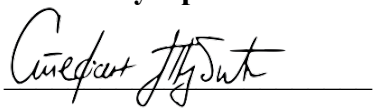
заснован на функционалној декомпозицији

- резултат сопственог истраживачког рада;
- да дисертација ни у целини ни у деловима није била предложена за стицање друге дипломе према студијским програмима других високошколских установа;
- да су резултати коректно наведени и
- да нисам кршио ауторска права и користио интелектуалну својину других лица.

У Београду,

19.06.2024.

Потпис аутора



Изјава о истоветности штампане и електронске верзије докторског рада

Име и презиме аутора: Стефан Тубић

Број индекса: 2016/5008

Студијски програм: Рачунарска техника и информатика

Наслов рада: Језик за опис архитектуре софтверских система заснован на функционалној декомпозицији

Ментор: проф. др Милош Цветановић

Изјављујем да је штампана верзија мог докторског рада истоветна електронској верзији коју сам предао ради похрањивања у **Дигиталном репозиторијуму Универзитета у Београду**.

Дозвољавам да се објаве моји лични подаци везани за добијање академског назива доктора наука, као што су име и презиме, година и место рођења и датум одбране рада.

Ови лични подаци могу се објавити на мрежним станицама дигиталне библиотеке, у електронском каталогу и у публикацијама Универзитета у Београду.

У Београду,

19.06.2024.

Потпис аутора

Изјава о коришћењу

Овлашћујем Универзитетску библиотеку „Светозар Марковић“ да у Дигитални репозиторијум Универзитета у Београду унесе моју докторску дисертацију под насловом:

Језик за опис архитектуре софтверских система

заснован на функционалној декомпозицији

која је моје ауторско дело.

Дисертацију са свим прилозима предао сам у електронском формату погодном за трајно архивирање.

Моју докторску дисертацију похрањену у Дигиталном репозиторијуму Универзитета у Београду и доступну у отвореном приступу могу да користе сви који поштују одредбе садржане у одабраном типу лиценце Креативне заједнице (*Creative Commons*) за коју сам се одлучио.

1. Ауторство (CC BY)
2. Ауторство – некомерцијално (CC BY-NC)
3. Ауторство – некомерцијално – без прерада (CC BY-NC-ND)
4. Ауторство – некомерцијално – делити под истим условима (CC BY-NC-SA)
5. Ауторство – без прерада (CC BY-ND)
6. Ауторство – делити под истим условима (CC BY-SA)

(Молимо да заокружите само једну од шест понуђених лиценци.

Кратак опис лиценци је саставни део ове изјаве.)

У Београду,

19.06.2024.

Потпис аутора

Стефан Ђуић

1. **Ауторство**. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, и прераде, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце, чак и у комерцијалне сврхе. Ово је најслободнија од свих лиценци.

2. **Ауторство – некомерцијално**. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, и прераде, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце. Ова лиценца не дозвољава комерцијалну употребу дела.

3. **Ауторство – некомерцијално – без прерада**. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, без промена, преобликовања или употребе дела у свом делу, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце. Ова лиценца не дозвољава комерцијалну употребу дела. У односу на све остале лиценце, овом лиценцом се ограничава највећи обим права коришћења дела.

4. **Ауторство – некомерцијално – делити под истим условима**. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, и прераде, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце и ако се прерада дистрибуира под истом или сличном лиценцом. Ова лиценца не дозвољава комерцијалну употребу дела и прерада.

5. **Ауторство – без прерада**. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, без промена, преобликовања или употребе дела у свом делу, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце. Ова лиценца дозвољава комерцијалну употребу дела.

6. **Ауторство – делити под истим условима**. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, и прераде, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце и ако се прерада дистрибуира под истом или сличном лиценцом. Ова лиценца дозвољава комерцијалну употребу дела и прерада. Слична је софтверским лиценцама, односно лиценцама отвореног кода.