Универзитет у Београду
Електротехнички факултет

Захарије Р. Радивојевић

МЕТОДОЛОГИЈА ПРОЈЕКТОВАЊА СИМУЛАТОРА АРХИТЕКТУРЕ И ОРГАНИЗАЦИЈЕ РАЧУНАРА

докторска дисертација

Београд, 2012

UNIVERSITY OF BELGRADE
School of Electrical Engineering

Zaharije R. Radivojević

METHODOLOGY FOR DESIGNING SIMULATORS OF COMPUTER ARCHITECTURE AND ORGANIZATION

Doctoral Dissertation

Belgrade, 2012

Ментор:

др Јован Ђорђевић, редовни професор

Универзитет у Београду, Електротехнички факултет

Чланови комисије:

др Јован Ђорђевић, редовни професор

Универзитет у Београду, Електротехнички факултет

др Зоран Јовановић, редовни професор

Универзитет у Београду, Електротехнички факултет

др Душан Старчевић, редовни професор

Универзитет у Београду, Факултет организационих наука

др Бошко Николић, ванредни професор

Универзитет у Београду, Електротехнички факултет

Датум одбране: 04.06.2012.

Мајци

Наслов докторске дисертације: Методологија пројектовања симулатора архитектуре и организације рачунара

Резиме: У овом раду се разматра методолошки приступ дизајну симулатора из области архитектуре и организације рачунара који треба да омогући развој симулатора дигиталних система произвољног нивоа сложености способних за рад у конкурентном и дистрибуираном окружењу. Да би се омогућио формирање методологије на почетку рада је приказан преглед наставе у области архитектуре и организације рачунара на основним студијама, као и преглед области пројектовања симулатора где је посебан акценат био стављен на области конкурентног и дистрибуираног програмирања које студенти треба да познају као би могли да развију симулаторе који омогућавају рад у таквом окружењу. На основу спроведене евалуације симулатора који се користе у настави из области архитектуре и организације рачунара а који имају расположив изворни код предложено је решење које се заснива на коришћењу слојевите архитектуре код које је сваки слој одговоран за други вид обраде и комуникације. Предложено решење се састоји из коришћења пет слојева: логичког, извршног, презентационог, симулационог, и слоја физике. Детаљи везани за процедуре и објашњења техника које се користе за реализацију ових слојева су приказани у раду. За сваки слој предложеног решења је дат аналитички модел процене времена извршавања симулације у зависности од улазних параметара приликом рада у конкурентном и дистрибуираном окружењу. Централни део рада описује симулатор дискретних догађаја опште намене развијен према описаној методологији као симулатор архитектуре и организације рачунара који је способан за рад у конкурентном и дистрибуираном окружењу. Опис симулатора и његових делова је дат са становишта детаља имплементације где су представљени пакети реализовани на основу предложене методологије, као и са становишта коришћења где су описане карактеристичне ситуације у којима се симулатор може користити. На основу имплементације симулатора и пратећих библиотека развијене су лабораторијске вежбе и пројекти из предмета конкурентно и дистрибуирано програмирање, које су представљене у наставку рада као и евалуација постигнутих резултата у настави. Поред ове евалуације на крају рада је

представљена и евалуација симулатора са становишта експерименталних резултата и са становишта аналитичког модела као би се утврдило у којим случајевима и у ком обиму се могу користити симулатори развијени сходно описаној методологији.

Развијена је методологија која се заснива на коришћењу слојевите архитектуре код које је сваки слој одговоран за различите облике комуникације и обраде. Одлука да се користи слојевита архитектура приликом пројектовања симулатора архитектуре и организације рачунара способних за извршавање у конкурентном и дистрибуираном окружењу омогућила је да се на једноставан начин раздвоје поступци обраде од поступака комуникације, синхронизације и интеракције између слојева без улажења у начине на које се подаци користе; На основу предложене методологије је без потешкоћа развијен SLEEP, симулатор архитектуре и организације рачунара са слојевитом архитектуром који се може користити и као симулатор дискретних догађаја опште намене. Показало се да предложени приступ развоју SLEEP симулатора доводи до стварања могућности за извршавање симулације користећи рад са једном нити, рад са више нити и рад у дистрибуираном окружењу; На основу евалуације резултата добијених током коришћења симулатора као један од закључака се издваја чињеница да се поступак пројектовања симулатора архитектуре и организације може користити у настави из конкурентног и дистрибуираног програмирања као полазна основа за поступак паралелизације апликација са једном током контроле. Поред тога симулатор се може користити и приликом тестирања тако паралелизованих апликација јер на конзистентан начин ствара велико оптерећење које у великој мери покрива понашање програма са већим бројем токова контроле; На основу евалуације резултата добијених на основу аналитичког модела и мерења перформанси може се закључити да се време извршавања симулације у области наставе архитектуре и организације рачунара може значајно смањити у конкурентном и дистрибуираном окружењу у случају ниског степена интеракције са корисником. У случају просечног нивоа интеракције ограничавајући фактор приликом извршавања симулација представља време потребно за приказ резултата симулације које је упоредиво са временом потребним за обраду саме симулације.

Кључне речи: Пројектовање симулатора; Архитектура и организација рачунара; Конкурентно програмирање; Дистрибуирано програмирање; Симулатори дискретних догађаја опште намене; Аналитички модел.

Научна област: Електротехника и рачунарство

Ужа научна област: Рачунарска техника и информатика

УДК број: 621.3:004.2

Title of the dissertation: Methodology for designing simulators of computer architecture and organization

Summary: This paper presents methodological approach to the computer architecture and organization simulator design. The methodological approach should help students to bridge the gap between theory and practice in the domain of computer architecture and organization simulator design, and to design simulators capable to work in a concurrent and distributed environment. In order to achieve this goal at the beginning of the paper a survey of undergraduate level courses was given. The courses include those in the field of computer architecture and organization and in the field of concurrent and distributed programming. One possible solution to the problem is given after an analysis of simulators that have available source code. Proposed solution is based on a multi-layer design where each layer is responsible for different type of processing and communication. The solution includes five different layers: logic, simulation, execution, presentation, and physics. For each proposed layer an analytical model that estimates execution time of simulation in a concurrent and distributed environment is given. The central part of the paper presents a discrete event simulator names SLEEP developed by using the proposed methodology. The design is presented through explanations of simulator structure and simulator usage. Based on the developed simulator and supporting libraries a set of laboratory exercises and project assignment was presented. At the end of the paper evaluation of the proposed analytical model and laboratory excursuses and project assignments was given.
Methodology, for designing simulators of computer architecture and organization, based on the layered architecture where each layer is responsible for different forms of communication and processing was developed. The decision to use a layered architecture for designing simulators of computer architecture and organization that are capable to work in a concurrent and distributed environment enabled strict separation between simulation execution and processes communication, process synchronization and interaction between the layers; Based on the proposed methodology computer architecture and organization simulator named SLEEP wad developed. The simulator adopts layered approach and can be uses as a general purpose discrete event simulator. The proposed approach enables the SLEEP simulator to execute simulations within single-thread, multi-thread and distributed environment; Based on the evaluation

obtained during the SLEEP simulator usage can be concluded that the process of designing simulators for computer architecture and organization can be used for teaching parallelization within a concurrent and distributed programming. Based on the evaluation of results, obtained from the analytical model and from the performance measurement, can be concluded that the execution time of simulations, which are used for teaching computer architecture and organization, can be significantly reduced in a concurrent and distributed environment in cases of low interaction with the user.

Key words: Simulator design; Computer architecture and organization; Concurrent programming; Distributed programming; General purpose discrete event simulator; Analytical model.

Scientific field: Electrical and Computer Engineering

Scientific subfield: Computer Engineering and Information Theory

UDC number: 621.3:004.2

Садржај

1 Увод ...1

2 Дефиниција проблема и предлог решење ...5

2.1 Архитектура и организација рачунара ... 5

2.1.1 Преглед области Архитектура и организација рачунара ... 5

2.1.2 Преглед постојећих симулатора из области Архитектуре и организације рачунара ... 7

2.1.3 Преглед карактеристика симулатора ... 13

2.1.4 Евалуација симулатора са становишта покривања области архитектуре и организације рачунара ... 17

2.1.5 Евалуација симулатора са становишта карактеристика ... 19

2.2 Конкурентно и дистрибуирано програмирање ... 22

2.2.1 Преглед области Конкурентно и дистрибуирано програмирање ... 22

2.2.2 Преглед постојећих решења из области Конкурентно и дистрибуирано програмирање ... 26

2.3 Предлог решења на глобалном нивоу ... 27

3 Пројектовање симулатора архитектуре и организације рачунара ...31

3.1 Логичка структура симулатора ... 31

3.1.1 Пројектовање логичког дела симулатора ... 32

3.1.2 Пројектовање извршног дела симулатора ... 44

3.1.3 Пројектовање симулационог дела симулатора ... 60

3.1.4 Пројектовање физичког дела симулатора ... 70

3.1.5 Пројектовање презентационог дела симулатора ... 75

3.2 Аналитички модел времена извршавања симулације ... 85

3.2.1 Моделовање времена обраде логичког слоја ... 87

3.2.2 Моделовање времена обраде извршног слоја ... 87

3.2.3 Моделовање времена обраде презентационог слоја ... 105

3.2.4 Моделовање времена обраде симулационог слоја ... 106

3.2.5 Моделовање времена обраде слоја физике ... 106

4 SLEEP симулатор ...107

4.1 Реализација симулатора SLEEP ... 108

4.1.1 Логички слој симулатора ... 111

4.1.2 Извршни слој симулатора ... 133

4.1.3 Симулациони слој симулатора ... 138

4.1.4 Презентациони слој симулатора ... 165

4.1.5 Слој физике компонената ... 167

4.2 Начин коришћења SLEEP симулатора ... 168

4.2.1 Креирање логичких компонената ... 171

4.2.2 Креирање графичког интерфејса компонената ... 179

4.2.3 Задавање параметара компонената ... 183

4.2.4 Шаблонске компоненте 185
4.2.5 Праћење тока симулације 187
4.2.6 Задавање параметара конкурентне и дистрибуиране симулације 192
4.2.7 Задавање параметара физике компонената 193

5 Евалуација резултата 195

5.1 Евалуација резултата симулатора добијених на основу коришћења у настави 195

5.1.1 Организација лабораторијских вежби 197
5.1.2 Поступак тестирања 206
5.1.3 Студентски пројекти 208
5.1.4 Евалуација коришћења симулатора и библиотека 211

5.2 Евалуација резултата добијених на основу експерименталних резултата и аналитичког модела 214

5.2.1 Експериментални резултати 214
5.2.2 Евалуација аналитичког модела на основу експерименталних резултата 226

6 Закључак 239

Литература 243

Прилози 254

Прилог А: Преглед курсева на Електротехничком факултету 254

Прилог Б: Детаљан преглед симулатора из области Архитектуре и организације рачунара 261

Прилог В: Преглед наставе из конкурентног и дистрибуираног програмирања на првих 100 универзитета Шангајске листе 285

Прилог Г: Листа слика: 295

Прилог Д: Листа табела: 301

1 Увод

Рачунари су присутни у свим сферама живота и рада човека, а то присусто се огледа кроз коришћење од најмањих до највећих уређаја са којима се човек сусреће. Човек у неким ситуацијама није свестан присуства рачунара јер су сакривени од корисника и обављају специјализовану обраду. Корисник се најчешће среће са рачунаром кроз интеракцију са одговарајућим корисничким програмима. Да би кориснички програми могли да раде на рачунару поред самог програма потребне су и услуге оперативног система који води рачуна о повезаности програма са хардвером који се налази у позадини који треба да обави извршавање задатог програма. Интеракција коју и кориснички програм и оперативни систем обављају са хардвером се постиже кроз архитектуру рачунара која представља интерфејс према организацији рачунара која је сакривена од корисника. Да би се удовољило човековим потребама за све комплекснијим израчунавањима и обрадама рачунари постају све сложенији да би омогућили конкурентну обраду већег броја захтева. Некада један рачунар, ма како био сложен, није довољан да обави такву обраду и појављује се потреба за повезивањем већег броја рачунара који би у дистрибуираном окружењу обављали обраду. Да би се омогућио развој овако сложених рачунарских система постоји потреба за изучавањем архитектуре и организације рачунара, као и техника конкурентног и дистрибуираног програмирања као би се искористили развијени системи.

Студенти се са концептима рада рачунарских система сусрећу на предавањима и вежбама на табли које треба да им омогуће да разумеју апстрактне појмове о раду рачунарских система. Изучавање рада рачунарских система захтева непрестано прилагођавање и унапређивање знања и вештина које студенти треба да савладају. Један од начина да се ово оствари је унапређивањем наставе и коришћењем одговарајућих алата који треба да помогну студентима да премосте јаз између теоријских основа стечених на предавањима и практичних концепата. Алати треба да омогуће студентима да изучавају понашање постојећих система, али и да им омогуће да самостално креирају и прате рад креираних система. Приликом рада алати треба да омогуће верно праћење рада система и са

довољним нивоом детаља.

На курсевима архитектуре и организације рачунара као један од основних алата користе се софтверска окружења за симулацију рада реалних система која су развијена посебно за ову намену. Ова софтверска окружења се по могућностима које пружају кориснику и сложености своје имплементације доста разликују у зависности да ли студенте треба да упознају са конкретним имплементацијама појединачних концепата из архитектуре и организације рачунара или да им омогуће да самостално пројектују дигиталне системе. У зависности од године на којој се користе симулатори се разликују и по нивоу детаља које студенти могу да прате током симулације. Што се тиче алата и окружења који се користе на курсевима који се баве конкурентним и дистрибуираним програмирањем скуп лабораторијских вежби је развијен тако да се студенти на дискретан и ограничен начин сусрећу са практичним проблемима. Приступи лабораторијским вежбама на курсевима везаним за конкурентно и дистрибуирано програмирање се заснивају на коришћењу малих, независних задатака, где сваки задатак треба да демонстрира посебан концепт. Овако развијени мали задаци су често вештачки, имају за циљ да јасно истакну концепт, али се ретко могу применити у већој мери приликом развоја практичних и сложених решења. Детаљном анализом расположивих софтверских окружења уочено је да не постоји систем који на одговарајући начин може у потпуности да подржи наставу на курсевима архитектуре и организације рачунара где студенти треба да се баве пројектоване симулатора, као ни наставу из конкурентног и дистрибуираног програмирања где студенти треба да се обуче да изврше паралелизацију секвенцијалног кода који се извршава у једном току контроле.

Докторска дисертација има за циљ да развије методолошки приступ пројектовању симулатора из области наставе архитектуре и организације рачунара који треба да омогући развој симулатора дигиталних система произвољног нивоа сложености способних за рад у конкурентном и дистрибуираном окружењу. Ради модуларности и проширивости, као један могући приступ решењу предлаже се методологија која треба да се заснива на коришћењу слојевите архитектуре код које је сваки слој одговоран за различите облике комуникације и обраде. Предложено решење треба да се састоји из

коришћења пет слојева: логичког, извршног, презентационог, симулационог, и слоја физике. Логички слој симулатора треба да се односи на опис логичког понашања компоненти које се симулирају и њихове међусобне везе на нивоу понашање. Извршење слој симулатора треба да омогући коришћење различитих алгоритме извођења симулације као би добили што прецизнији резултати симулације. Презентациони слој симулатора треба да представља везу онога шта се симулира са корисником система. Симулациони слој симулатора треба да обезбеди слој апстракције између инстанци извршног слоја симулатора и презентационог слоја симулатора како би се омогућило извршавање у дистрибуираном окружењу. Слој физике симулираних компоненти треба да обезбеди могућност провере и израчунавања потребних физичких карактеристика које логичка симулација може да произведе. Методологија треба да предложи и поступке комуникације, синхронизације и интеракције између слојева предложене архитектуре како би се омогућило пројектованом симулатору архитектуре и организације рачунара извршавање у конкурентном и дистрибуираном окружењу. Овако развијени симулатор би требало да помогне студентима да кроз интердисциплинарни приступ схвате како функционишу сложени дигитални системи, како се описани дигитални хардверски системи могу симулирати у софтверу, како се софтвер може генерализовати на друге системе, и како се ти системи могу симулирати у конкурентном и дистрибуираном систему. За представљен методологију потребно је дати аналитички модел времена потребног за обављање симулације како би се могло одредити који слојеви су критични ресурси и када има потребе за радом у конкурентном и дистрибуираном окружењу.

Докторска дисертација садржи шест поглавља, скуп неопходних прилога и преглед коришћене литературе. Друго поглавље садржи описе симулатора који се користе у области архитектура и организација рачунара, као и преглед алгоритама симулације. У оквиру ове главе се разматра област наставе архитектуре и организације рачунара, као и конкурентног и дистрибуираног програмирања, са становишта IEEE/ACM препорука, као и евалуација симулатора са становишта тих препорука, али и реализованих алгоритама симулације. На крају ове главе је дат оквирни предлог решења проблема пројектовања симулатора архитектуре и

организације рачунара погодног за рад у конкурентном и дистрибуираном окружењу. У глави три је дата детаљна дефиниција карактеристика софтверског система симулатора дискретних догађаја који се може користити за креирање симулатора дигиталних компоненти у архитектури и организацији рачунара погодног за рад у конкурентном и дистрибуираном окружењу. У наставку овог поглавља је представљен аналитички модел који описује рад предложеног решења као и очекиваних перформанси приликом рада у конкурентном и дистрибуираном окружењу. У четвртом поглављу је приказано реализовано софтверско окружење, SLEEP симулатор, које треба да омогући развој симулатора дискретних догађаја који могу да описују рад дигиталних система који се срећу у архитектури и организацији рачунара. Ово поглавље обухвата детаљан приказ реализације софтверског система са становишта имплементације, али и са становишта коришћења, као и преглед најзначајнијих проблема и начина на који су ти проблеми решени. Пето поглавље даје приказ искустава и резултата у примени реализованог софтверског окружења. Резултати омогућавају процену предложеног софтверског решења, али и аналитичког модела који описује рад у конкурентном и дистрибуираном окружењу. Евалуација се обавља са становишта карактеристичног оптерећења које генеришу симулатори архитектуре и организације рачунара приликом рада у конкурентном и дистрибуираном окружењу, као и са становишта постигнућа у настави након увођења система. У шестом поглављу је изложен закључак. На крају, се дају неопходни прилози и преглед коришћене литературе.

2 ДЕФИНИЦИЈА ПРОБЛЕМА И ПРЕДЛОГ РЕШЕЊЕ

У овог глави се даје преглед области архитектура и организација рачунара као и симулатора који се користе у овој области. Да би се омогућило да се проблем методологије развоја симулатора, архитектуре и организације рачунара, који се могу прилагодити за рад у паралелном окружењу даје се преглед области конкурентно и дистрибуирано програмирање. На крају ове главе је представљен на глобалном нивоу предлог решења методологије пројектовања симулатора архитектуре и организације рачунара погодних за рад у конкурентном и дистрибуираном окружењу.

2.1 Архитектура и организација рачунара

У овој секцији се разматра преглед области архитектура и организација рачунара како би се дефинисао домен проблема који описују симулатори који се користи у овој области. Након прегледа области даје се преглед постојећих симулатора из области архитектуре и организације рачунара како би се уочиле функционалне карактеристике симулатора. На основу размотрених параметара на крају ове секције је представљена евалуација симулатора са становишта покривања области архитектуре и организације рачунара, као и евалуација симулатора са становишта њихових карактеристика.

2.1.1 Преглед области Архитектура и организација рачунара

Пре него што се спроведе генерализација приступу области пројектовања симулатора архитектуре и организације рачунара потребно је утврдити опсег ове области и утврдити који све постојећи симулатори припадају овој области [1]. У случају области Архитектуре и организације рачунара списак тема које покрива ова област може се успоставити коришћењем Смерница за развој програма основних студија из рачунарске технике коју је представило IEEE/ACM удружење [2] и [3]. Ове смернице представљају основ за развој предмета који се могу наћи у свим програмима рачунарске технике и информатике а који покривају област архитектуре и организације рачунара. Архитектура рачунара и организација је

издвојена као једна од најбитнијих области знања које је потребно изучавати и чији скуп тема обухвата 10 јединица. У оквиру сваке јединице знања наведене су теме које је потребно изучавати као и очекивано време које је потребно посветити датој теми.

Табела 1: Листа тема унутар појединих образовних јединица области архитектура и организација рачунара

Образовна јединица	Теме
Основни принципи архитектуре рачунара	Организација фон Нојманове машине Типови података, типови инструкција, начини адресирања, формати инструкција Дохватање инструкција из меморије, фазе декодовања и извршавања инструкције Регистара и регистарски фајлови Типови инструкција и начини адресирања Позиви и повратак из потпрограма Програмирање користећи асемблерски језик Технике улаза/излаза и механизам прекида Друга питања дизајн
Архитектура и организација меморијских система	Хијерархијски меморијски систем Организација, карактеристике и перформансе меморије Кашњење, време циклуса, пропусни опсег Преклапање приступа меморијским модулима Кеш меморија Виртуелна меморија Поузданост меморије система; откривање и исправљање грешака Технологије израде меморија (SRAM, DRAM, EPROM и Flash) Електронске, магнетне и оптичке технологије
Интерфејси и комуникација	Основе улаза/излаза: протоколи на магистрали Технике: програмирани излаз/излаз, DMA Прекиди: векторисан, приоритет, опслуживање, обрада и повратак Повезивање са меморијским системом Магистрала: протоколи и арбитрација
Улазно/излазни уређаји	Спољни системи за складиштење података; организација и структура дискова и оптичких меморија Основни улазно/излазни уређаји, као што су тастатура и миш RAID архитектуре Видео контролери Перформансе улазно излазних уређаја SMART технологије и откривање грешака Интерфејс процесора према мрежним уређајима
Дизајн система процесора	Такт, контролне, адресне и магистрале података Декодовање адреса и меморијски интерфејс Основи паралелног и серијског интерфејса Тајмери System firmware
Организација процесора	Имплементација фон Нојманове машине Системи са једном и више магистрала Инструкцијски сет Веза између архитектуре рачунара и програмских преводиоца Имплементација инструкција Управљачка јединица (ожичена и микропрограмска реализација) Аритметичка јединица Проточна обрада Трендови у архитектури рачунара: CISC, RISC, и VLIW Основи паралелизације на нивоу инструкција ILP (instruction-level parallelism) Хазарди у проточној обради

Овде је издвојене шест јединица код које свака има барем пет или више часова а припада језгру наведене области: Основни принципи архитектуре рачунара, Архитектура и организација меморијских система, Интерфејси и комуникација, Улазно/излазни уређаји, Дизајн система процесора, Организација процесора. Ове јединице су као и списак теме које покривају представљени су у Табели 1.

На Електротехничком факултету у Београду издвојене области везан за архитектуру и организацију рачунара се изучавају на неколико курсева на основним академским студијама кроз скуп предавања, вежбе на табли, лабораторијских вежби и скупа пројеката које студенти самостално или у групама раде. На основним студијама ови курсеви заузимају значајно место и прате студенте током прве три године студија. Наведене области се изучавају у оквиру курсева: Основи рачунарске технике 1 (ОРТ1), Практикум из основа рачунарске технике 1 (ПОРТ), Основи рачунарске технике 2 (ОРТ2), Архитектура рачунара (АР), Архитектура и организација рачунара 1 (АОР1) и Архитектура и организација рачунара 2 (АОР2). Преглед наставе на овим курсевима је дан на крају рада. Ови курсеви укључују основне логичког пројектовања дигиталних система, уводне и напредне курсеве из архитектуре и организације рачунара и као и курс пројектовања рачунарских система. Током курсева, од студената се очекује да разумеју понашање дигиталних система и да се сами баве пројектовањем и експериментисањем са добијеним решењима.

2.1.2 Преглед постојећих симулатора из области Архитектуре и организације рачунара

Системи који омогућавају симулацију дигиталних система представљају софтверска окружења која најчешће имају могућност за пројектовање, анализу и симулацију дигиталних система произвољне сложености. У области архитектуре и организације рачунара системи који се симулирају најчешће представљају поједине делове процесора, цео рачунар, али у неким случајевима представљају чак и сложене рачунарске системе. Основна сврха симулатора који се користе у овој области је да помогне студентима да путем симулације делова рачунара схвате поједине концепте са којима су се сусрели на предавањима и вежбама из поменутих предмета ([4] и [5]). У овој секцији су представљени симулатори који

се користе у настави из предмета повезаних са облашћу архитектуре и организације рачунара.

Отворен литература нуди различите симулатора погодне за наставу на курсевима из архитектуре и организације рачунара. Основне карактеристике одабраних симулатора приказане су у Табели 2 и укључују аутора, циљну групу корисника, оперативне системе на којима се симулатори извршавају, програмске језике који су коришћени приликом развоја симулатора, доступност симулатора у погледу лиценци, као и циљне симулиране архитектуре рачунара. Овим прегледом су обухваћени искључиво симулатори чији је изворни код доступан за анализу. Имајући у виду да се симулатори веома разликују у погледу карактеристика и дизајна, посебна пажња је посвећена процесу пројектовања симулатора који се користе у области архитектуре и организације рачунара.

Табела 2: Основне карактеристике одабраних симулатора

Назив	Аутор	Корисници	Оперативни систем	Програмски језик	Доступност	Циљна архитектура	Референце
ANT	Harvard University, USA	Студенти	Windows, Linux, и MacOS	C, Java	Free AC	Custom MIPS R2000 like architecture	[6] и [7]
CPU Sim	Colby College, USA	Студенти	Windows, Linux, и MacOS	Java	Free	RISC-like accumulator-based architectures	[8] и [9]
DigLC2	University Paris-Sud, France	Студенти	Windows и UNIX	C	GPLv2	Custom LC-2 architecture	[10] и [11]
DLXview	Purdue University, USA	Студенти	Solaris, SunOS, HP-UX, и Linux	C	GPLv2	DLX architecture	[12] и [13]
EDCOMP	University of Belgrade, Serbia	Студенти	Windows, Linux, и MacOS	Java	Free	Custom CISC architecture	[14] и [15]
HASE	University of Edinburgh, UK	Студенти, развој, тестирање, наставници	Solaris, Linux, и Windows	Java	Free AC	-	[16] и [17]
HASE-Dinero	University of Edinburgh, UK	Студенти	Solaris, Linux, и Windows	HASE++	Free	-	[18] и [19]
JCachesim	University of Siena, Italy	Студенти	Windows, Linux, и MacOS	Java	Free	MIPS R3000	[20] и [21]
JHDL	Brigham Young University, USA	Студенти, развој, тестирање, анализа перформансе	Windows, Linux, и MacOS	Java	OSSLA	-	[22], [23] и [24]
Logisim	Hendrix College, USA	Студенти	Windows, Linux, и MacOS	Java	GPL	-	[25] и [26]
M5	The University of Michigan, USA	Развој хардвера и софтвера, студенти, тест, анализа перформанси	Linux, MacOS X, Solaris, OpenBSD, и Cygwin	Python, C++	BSD	Alpha, SPARC, MIPS, and ARM ISAs	[27] и [28]
RM	University of Catalonia	Студенти	Windows и Linux	C, C++	Free	Custom RISC architecture	[29], [30] и [31]

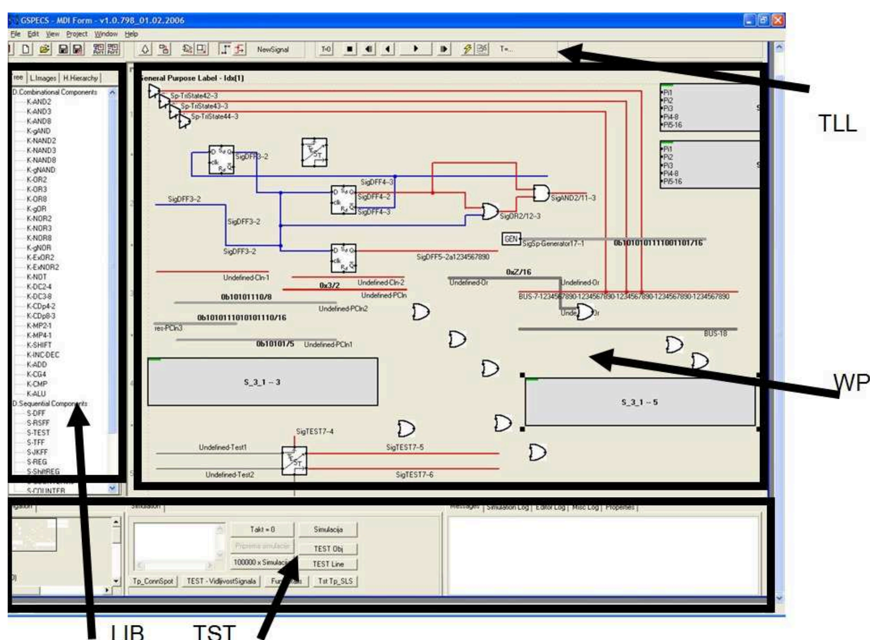
Назив	Аутор	Корисници	Оперативни систем	Програмски језик	Доступност	Циљна архитектура	Референце
	(UPC), Spain						
RSIM	Rice University Houston Texas, USA	Студенти, истраживачи	UNIX компатибилне	C, C++	UIUC/NCSA	Custom architecture	[32], [33] и [34]
SIMCA	University of Minnesota, USA	Студенти, развој и тестирање	SUN SunOS 5.6 и IRIX 6.2	C	Free	Custom superthreaded architecture	[35], [36] и [37]
SimFlex	Carnegie Mellon University, USA	Развој софтвера, студенти, тестирање, анализа перформанси	Linux	C++	Free	Architecture supported by Simics	[38], [39] и [40]
Simics	Virtutech AB Stockholm, Sweden	Развој софтвера, студенти, тестирање, анализа перформанси	Linux (x86, PowerPC, и Alpha), Solaris/UltraSparc, True64/Alpha, и Windows 2000/x86	Python, C	Commercial, Free AC	PowerPC, x86, ARM, and MIPS	[41] и [42]
SimOS	Stanford University, USA	Студенти, развој, анализа перформанси	Irix, Solaris, и Linux	C	Free NC	MIPS-based multiprocessors architecture	[43] и [44]
SimpleScalar	University of Wisconsin-Madison, USA	Студенти, развој и анализа перформанси	UNIX и Windows	C	Free NC	Alpha, PISA, ARM, and x86	[45], [46] и [47]
VSDS	University of Belgrade, Serbia	Студенти, развој	Windows	Visual Basic	Free NC	-	[48], [49] и [50]

Легенда: NA – Није доступно, Free AC – Слободно за академске институције, OSSLA – под Open Source Software License Agreement лиценцом, GPL – под GNU General Public License, GPLv2 – GNU General Public License, Version 2, BSD – Berkeley-Style Open Source License, UIUC/NCSA – University of Illinois/NCSA Open Source License, Free NC – бесплатно за некомерцијалне академске сврхе

Генерално гледано ови симулатори се могу поделити у две главне групе: на симулаторе намењене пројектовању нових система и симулаторе намењене анализи већ развијених система.

Прва група симулатора садржи одговарајуће алате и развијене методе које омогућавају кориснику да креира конфигурације специфичног рачунарског система, а затим да их симулира. Иако су неки симулатори пројектовани као симулатори општих система који се састоје од дигиталних логичких кола, док су други експлицитно развијене за моделовање и проучавање архитектуре и организације рачунара, ипак садрже велики број заједничких карактеристика које им дозвољавају да буду посматрани како јединствена група. На слици 1 је узет

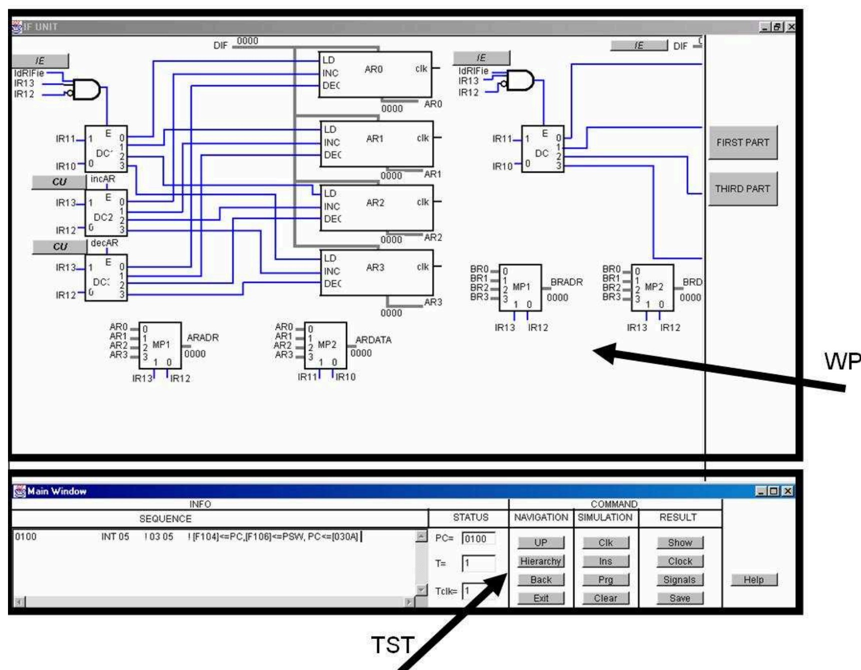
један симулатор из ове групе као илустрација могућности које обично нуде ове симулатори. Главни прозор симулатора је подељен у три области: LIB – Скуп доступних библиотека (горе лево), WP - Радна површина (горе десно), TST – Опције за тестирање (на дну), и TLL – Алати за конфигурацију (горе). Коришћење ових симулатора се може поделити у пет корака у којима корисник: Бира скуп компонената из доступних библиотека; Распоређује изабране компоненте по радној површини; Повезује одабране компоненте са другим компонентама у циљу стварања нових компонената; Креира кориснички интерфејс за тако ново креиране компоненте; Обавља тестирање компоненте креиране користећи доступну палату алата унутар симулатора. Након завршетка рада овако креирана компонента се појављује у библиотеци и може се користити приликом развоја нових компоненти.



Слика 1: Основни прозор симулатора намењених пројектовању

Најрепрезентативнији симулатора из ове групе су: HASE, JHDL, Logisim, M5, Simics, и VSDS. HASE (Hierarchical Computer Architecture design and Simulation Environment) представља сет алата за креирање хијерархијских модула, за конфигурисање тих модула, за симулацију развијених система, као и за анализу скалабилности архитектуре. JHDL (структурално засновани језик за опис хардвера реализован користећи програмски језик Јава) је скуп FPGA CAD алата

који омогућава кориснику да пројектује структуру и распоред логичких кола, да обави проверу постојања грешака и да обави комплетну синтезу хардвера. Logisim је образовни алат за пројектовање и симулацију дигиталних логичких кола. M5 обезбеђује функционалности неопходне за детерминистичку симулирају умрежених рачунара, укључујући рада комплетног система (full system), детаљан рад са подсистемом улаза-излаза, као рад више умрежених система. Simics је платформа за симулацију потпуног система која може да ради користећи стварне фирмвере, као и потпуно неизмењен кернел и везне програме. VSDS (Визуелни генератор дигиталних структура) је сет алата за креирање хијерархијски модула, за конфигурисање модула, симулацију и тестирање креираних система у области наставе из предмета везаних за архитектуру и организацију рачунара.



Слика 2: Основни прозор симулатора намењених анализи већ развијених система

На слици 2 је узет један симулатор из друге групе као илустрација могућности које обично нуде ове симулатори [56]. Главни прозор симулатора је подељен у две области: WP - Радна површина (горе) и TST - Опције за тестирање (на дну). Коришћење ових симулатора се може поделити у пет корака у којима корисник: Иницијализација симулатора са подразумеваним параметрима; Одређивање тест конфигурације; Стварање теста вектора и дијаграма чије

понашање треба посматрати; Обављање саме симулације; Анализирање резултата добијених за наведене конфигурације.

Овај тип симулатора обично има и неке конфигурабилне параметре који треба да буду постављени пре почетка извршавања симулације. Ови параметри обично обухватају дефинисање броја периферија, почетне вредности регистара и меморијских локација, и ширине појединих регистара. Коришћење ових конфигурационих параметара омогућавају да се поставе различите конфигурације рачунарских система за тестирање. Извештаји тестирања који се стварају на основу обављене симулације су обично дати у форми табела која садржи вредности регистара и меморије, временске дијаграме, текстуалне датотеке са разним контролним мерењима.

Најрепрезентативнији симулатора из ове групе се користе у евалуацији су: ANT, CPU Sim, DigLC2, DLXview, EDCOMP, HASE-Dinero, JCachesim, RM, RSIM, SIMCA, SimFlex, SimOS, и SimpleScalar. ANT је виртуелна машина заснована на основној RISC архитектури. CPU Sim представља пакет рачунарских симулација који омогућава кориснику да одреди детаље процесора који треба да се симулира. DigLC2 је симулатор дигиталних система на нивоу логичких кола који пружа детаљан опис свих компоненти симулираног процесора. DLXview је симулатор процесор са проточном обрадом који користи DLX скуп инструкција за три верзије DLX организације процесора. Easy CPU симулатор омогућава симулацију поједностављеног скупа инструкција који се јавља код Интел 80X86 архитектуре са анимираном презентацијом интерних операција које обавља рачунар. EDCOMP (EDucation COMPuter) је флексибилан симулатор рачунарског система CISC типа дат на нивоу трансфера између регистара. HASE Dinero представља окружење за симулацију рада кеш меморије. JCachesim је симулационо окружење за оцену перформанси рада рачунарског система заснованог на MIPS архитектури са кеш меморијом. RM (Rudimentary Machine) омогућава основну симулацију архитектуре рачунара засноване на инструкцијама RISC стила. RSIM (Rice Simulator for ILP Multiprocessor) је догађајима вођен симулатор за анализу дељене меморије код вишепроцесорских и једнопроцесорских система који поседују паралелизам на нивоу инструкција. SIMCA (The Simulator for Multi-threaded Computer Architecture) служи као

симулатор који се користи за процену перформансе супер-нитних архитектура као и за испитивање различитих алтернатива током дизајна рачунарског система. SimFlex је симулациони оквир који користи дизајн заснован на коришћењу компонената. SimOS је комплетно симулационо окружење рачунарског система дизајнирано за ефикасно и тачно изучавање система састављених од једног или више процесора. SimpleScalar је сет алата који обухватају компајлер, линкер, симулатор и алатке за визуелизацију рада симулатора.

Процена симулатора захтева коришћење релевантних критеријума. У одсуству било каквог познатим критеријумима који се користи за процену симулатора који се срећу на курсевима повезаним са облашћу архитектуре и организације рачунара, овде су усвојени критеријуми који се могу поделити у две групе: покривеност тема и карактеристике симулатора. Критеријум покривености тема треба да да одговор на то колики проценат области архитектуре и организације рачунара или неке њене под-области, покрива одабрани симулатор. Други критеријум треба да да одговор на то које све функционалне карактеристике што се тиче система који симулира и начина на који се обавља симулација има одабрани симулатор. Приликом избора критеријума за преглед ових симулатора постојала је могућност укључивања и других који би покриле начине и лакоћу коришћења симулатора на појединим курсевима, али они нису укључени због потребе за стварним коришћењем симулатора на истој групи студената што овакве критеријуме чине непрактичним. Изабрани критеријуми су ограничени на оне који се могу примењивати на основу података расположивих у отвореној литератури. Као додатни критеријуми узета су у обзир разматрања дата у [51]-[54].

2.1.3 Преглед карактеристика симулатора

Симулатори архитектуре и организације рачунара се могу посматрати са више аспеката. У овој секције се разматрају карактеристике одабраних симулатора са становишта карактеристика које су уочене приликом анализе датих симулатора. Ове карактеристике се могу поделити на карактеристике везане за оно шта се симулира, и на оно како се симулација обавља.

Сви симулатори на неки начин показују структуру симулираног система као композицију састављену од функционалних блокова. Постоје симулатори где

детаљи имплементације нису видљиви и подаци посматрани током симулације су на нивоу преноса информација између блокова највишег нивоа. Међутим, постоје симулатори, где се може посматрати структура блокова на нижим нивоима детаље имплементације улазећи у компоненте рекурзивно, све до најнижих детаља имплементације. На основу нивоа детаља видљивих током симулације, симулатори се могу поделити на оне са или без видљивих детаље имплементације рачунарског система. Критеријум усвојен је Хијерархијска представа. Нивои процене су Да и Не.

Једна група симулатора укључује алате за подршку дизајну компонената које се могу користити у више одвојених пројеката и њихову симулацију. Друга група симулатора омогућава само симулације система чија је архитектура и организација унапред дефинисана. Сходно томе, симулатори се могу поделити на оне који симулирају или кориснички дефинисане рачунарске системе или системе фиксне структуре. Критеријум усвојен је Дизајн компоненти. Нивои процене су Да и Не.

Приликом дефинисања нових компонената једна група симулатора омогућава дефинисање нових компонената користећи посебан језик за опис компонената. Коришћење посебног језика или коришћење неког постојећег језика је особина које може доста да олакша пројектовање симулатора сложених рачунарских система. Према овој категорији симулатори се могу поделити на оне симулаторе који дозвољавају коришћење посебној језика за опис компоненти. Критеријум усвојен је Језик за опис компоненти. Нивои процене су Да и Не.

Приликом извршавања симулације неки симулатори омогућавају да се компоненте које су уграђене у симулатор искористе у генеричком облику, то јест да се дефинишу компоненте које немају конкретну величину већ се коначна величина уноси тек у тренутку непосредно пред симулацију. Ово није само особина симулатора који омогућавају дизајн симулатора већ и симулатора са фиксном организацијом код којих је могуће постављањем неких параметар делимично изменити симулирани систем. Овде се обично ради о постајању величине појединих компонената или замена неког од алгоритама симулације без промене остатка симулације. Успостављени критеријум је Параметризација компоненти. Ниво процене је Да и Не.

Компоненте које се симулирају на курсевима Архитектуре и организације рачунара поред својих логичких карактеристика у неким случајевима захтевају и симулацију својих физичких карактеристика. Физичке карактеристике које се симулирају покривају најбитније параметре дигиталних система као што су напон, струја, потрошња и дисипација енергије. Неки симулатори дозвољавају и праћење и ових карактеристика у циљу детаљније анализе симулације. Критеријум који је овде усвојен је Физика компоненти. Ниво процене је Да и Не.

Многи симулатори имају веома добар визуелну презентацију, погодну да се покаже унутрашњи рад симулираног рачунарског система. С друге стране, постоје симулатори, без визуелну презентацију где су резултати дати у текстуалном облику са могућношћу за накнадне обраде добијених података. Симулатори се могу поделити на оне са или без визуелну презентацију. Критеријум усвојен је Визуелна презентација. Нивои процене су Да и Не.

Имплементација тока симулације се доста разликује међу симулаторима. У некима симулаторима омогућено је потпуна интеракција са корисником, тако да је кориснику омогућено да стартује и заустави симулација, да се враћа уназад, да сними до неког тренутка извршену симулацију и касније је покрене поново. Друга група симулатора омогућава само симулацију у пакетском моду где се задају почетни параметри симулације и онда се симулација пушта до задатог симулационог интервала. Симулатори се могу поделити на оне са интерактивном или пакетском контролом. Критеријум усвојен је Контрола симулације. Нивои процене су IC (интерактивна) и BC (пакетска).

Ниво грануларности на који се може поделити симулација може да буде на нивоу такта, инструкције, или програма. На основу најнижег нивоа гранулација који подржава неки симулатор, симулатори се могу поделити на оне који подржавају симулације на нивоу такта, инструкције или на нивоу целокупног програма који се извршава. Нивои процене су CL (ниво такта), IL (ниво инструкције) и PL (ниво програма).

Анализирани симулатори се доста разликују по начину на који је њихова интерна структура реализована и начину на који се симулација обавља. Један од основних параметара приликом имплементације симулатора је начин на који је реализован проток времена унутар симулације. Проток времена се најчешће

реализује како временом вођена симулација или догађајима вођена симулација. Поред ове две основне групе неки симулатори су реализовани и користећи под варијанте које захтевају познавање организације симулираног система. Такви симулатори могу да постигну боље перформансе, али се њихова примена ограничава само на конкретну имплементацију. Критеријум који је усвојен је Симулациони алгоритам, док је ниво процене TD (временом вођена симулација) и ED (догађајима вођена симулација).

Када се погледа сложеност система који се симулирају и времена потребног да се обави симулација уочава се потреба за конкурентним извршавањем симулација како би се искористили хардверски ресурси који постоје код доступних рачунара. Потреба за конкурентним извршавањем је изражена само код симулација веома сложених рачунарских система које се користе у области наставе из Архитектуре и организације рачунара, док се код једноставних система могућност паралелизације не може искористити јер и једнопроцесорски системи довољно брзо извршавају симулације. Критеријум који је успостављен је Паралелно извршавање, док су нивои процене Да и Не.

Када се посматрају курсеви на којима се ови симулатори користе може се приметити да је подршка за учење на даљину захтев који се поставља пред дизајнере симулатора како би се постигао већи успех приликом примене на студијама. Сходно томе, могућност коришћења симулатора преко Интернета постаје важан критеријум за процену симулатора. На основу доступности подршке за учење на даљину, симулатор може се поделити на оне са и оне без подршке за учење на даљину. Критеријум усвојен је Учење на даљину. Нивои процене су Да и Не.

Сходно претходним разматрањима, пожељно је да симулатор има могућност приказа хијерархијске структуре рачунарских система као и могућност прилагођења симулираног рачунарског система и дизајн нових компоненти користећи визуелни или текстуални опис нових компонената, пружање визуелне презентације тока симулације уз интерактивну контролу тока симулације на гранулитету такта, инструкције или програма, уз обављање симулације користећи већи број језгара или рачунара, као и да пружа могућност учења на даљину.

2.1.4 Евалуација симулатора са становишта покривања области архитектуре и организације рачунара

У овој секцији је приказана евалуација симулатора са становишта које области Архитектуре и организације рачунара дати симулатори покривају. Резултати евалуације су сажети у Табели 3, док је више детаља у вези са методологијом коришћеном за вредновање критеријума покривености дато у [55].

Табела 3: Евалуација симулатора са становишта покривања области

Назив	Основни принципи архитектуре рачунара	Архитектура и организација меморијских система	Интерфејси и комуникација	Улазно/излазни уређаји	Дизајн система процесора	Организација процесора	Укупан просек
ANT	81.25	22.22	20.00	0.00	10.00	25.00	28.26
CPU Sim	87.50	16.67	10.00	0.00	10.00	29.17	28.26
DigLC2	62.50	27.78	30.00	0.00	30.00	29.17	30.43
DLXview	87.50	22.22	0.00	0.00	20.00	58.33	36.96
EDCOMP	100.00	33.33	100.00	7.14	40.00	62.50	56.52
HASE	31.25	50.00	60.00	0.00	50.00	41.67	38.04
HASE-Dinero	56.25	33.33	20.00	0.00	10.00	25.00	26.09
JHDL	25.00	66.67	60.00	28.57	60.00	33.33	43.48
Logisim	18.75	11.11	0.00	0.00	20.00	20.83	13.04
M5	100.00	66.67	90.00	50.00	70.00	54.17	69.57
RM	75.00	5.56	10.00	0.00	10.00	33.33	25.00
RSIM	68.75	33.33	30.00	0.00	20.00	50.00	36.96
SIMCA	62.50	44.44	10.00	0.00	20.00	41.67	33.70
SimFlex	56.25	33.33	20.00	28.57	30.00	37.50	35.87
Simics	100.00	55.56	100.00	50.00	70.00	37.50	64.13
SimOS	68.75	22.22	50.00	7.14	20.00	33.33	33.70
SimpleScalar	100.00	11.11	10.00	0.00	10.00	54.17	35.87
VSDS	31.25	50.00	60.00	0.00	50.00	41.67	38.04

Теме које се односе на област Основни принципи архитектуре рачунара су доста добро покривене скоро свим посматраним симулаторима. Неки од симулатора, као што су EDCOMP, M5, Simics и SimpleScalar имају покривеност тема ове области од 100% па могу да представљају узор за реализацију

симулатора ове области. Што се поктивености ове области тиче изузетак чине симулатори, као што је Logisim (18,75%), који су развијени за специјализоване потребе и само се летимично баве темама из ове области.

Област Архитектура и организација меморијских система је доста широка и разуђена тако мали број симулатора покрива већи број тема ове области, а за разлику од области Основни принципи архитектуре рачунара ниједан од симулатора покрива све теме. Што се укупног количника тиче најбољу покривеност су постигли симулатори JHDL (66.67%), M5 (66.67%), и Simics (55.56%). Оно што се ипак може рећи је да се ови симулатори не баве директно темама различитих меморијских технологија и организација, као ни са поузданости и исправљањем грешака већ су системи намењени за развој симулатора и само узгредне реализације се баве овом области.

Теме које се односе на области Интерфејси и комуникација, Улазно/излазни уређаји, и Дизајн система процесора су у принципу подржани што се анализираних симулатора тиче, али тај проценат доста варира. Разлог за тако велико варирање карактеристика симулатора вероватно лежи у томе је да се већина анализираних симулатора концентрише углавном на темама везаним за основне компоненте рачунарског система који укључују процесор, главне меморије и једноставне улазно-излазне уређаје. Изузеци од овог тврђења су симулатора који су посебно развијени да се описују неку од тема из ова три области. Као резултат тога, област Интерфејси и комуникација има веома добру покривеност са симулаторима EDCOMP (100.00%), Simics (100.00%) и M5 (90.00%), подсистем Улазно/излазних уређаја са симулаторима M5 (50.00%) и Simics (50.00%) и област Дизајн са симулатора M5 (70.00%), Simics (70.00%), и JHDL (60.00%).

Унутар области Организација процесора велики број симулатора свој резултат има приближна резултату који је остварио у области Основни принципи архитектуре Овај резултат вероватно проистиче из чињенице да су ове две области комплементарне и, како је наведено у претходном ставу, баве основним компонентама рачунарског система, које су покривене од стране већине симулатора. Најбоља покривеност је постигнута са симулатора EDCOMP (62.50%), DLXview (58.33%), M5 (54.17%), SimpleScalar (54.17%) и RSIM

(50.00%).

Укупан резултат евалуације резултата датих у Табели 3 може да буде сагледан на два начина: општим показатељем покривености тема и покривеност тема по свакој јединици посебно. Најбоља покривеност тема се постиже са симулаторима M5 (69.57%), Simics (64.13%), EDCOMP (56.52%), и JHDL (43.48%). Међутим, као што је већ поменуто, постоје симулатори са нижом укупном процентом покривености тема, али са високим покривеност за одређену јединицу. Ова врста информација је важнија од укупног покривеност за оне курсеве који се баве појединим јединицама.

2.1.5 Евалуација симулатора са становишта карактеристика

У овој секцији је приказана евалуација симулатора са становишта које карактеристике уочене као битне током евалуације симулатора архитектуре и организације рачунара дати симулатори покривају. Резултати евалуације су сажети у Табели 4.

Симулатори основне архитектуре, као што су CPU Sim, DigLC2, и RM су развијени са примарним циљем да омогући кориснику да се упознају са функционисањем рачунарског система без залажења у дубље детаље имплементације. Анализирани симулатори у већини случајева покривају рад процесора, и показују како дохвата инструкција, као се декодује, како се одређују начини адресирања и операнди, како се дохватају операнди, како се обавља фаза извршења инструкције и смештање резултата. Симулатори JCachesim и Logisim се баве специфичним темама, као што су кеш меморија (JCachesim) и кола прекидачке логике (Logisim). Неки симулатори који имају сложену архитектуру (HASE, JHDL, M5, Simics и VSDS) омогућавају напредну подршку корисницима да дефинише сложене система, који се крећу од веома једноставних до веома сложених. Остали (ANT, DLXview, EDCOMP, HASE-Dinero, RSIM) обављају симулацију конкретних система велике сложености. Међу анализираним симулаторима (SIMCA, Simics, SimOS, SimpleScalar) су и они који омогућавају симулацију постојећих хардверских платформи.

Табела 4: Евалуација симулатора са становишта карактеристика

Симулатор	Хијерархијска представа	Дизајн компоненти	Језик за опис компоненти	Параметризација компоненти	Физика компоненти	Визуелна презентација	Контрола симулације	Корак симулације	Симулациони алгоритам	Паралелно извршавање	Учење на даљину
ANT	Не	Не	Не	Не	Не	Не	BC	IL	TD	Не	Не
CPU Sim	Да	Не	Не	Не	Не	Не	IC	CL	TD	Не	Не
DigLC2	Да	Не	Не	Не	Не	Да	IC	IL	ED	Не	Не
DLXview	Да	Не	Не	Не	Не	Да	IC	CL	TD	Не	Не
EDCOMP	Да	Не	Не	Не	Не	Да	IC	CL	TD	Не	Да
HASE	Не	Да	Да	Да	Не	Да	IC	CL	?	Не	Не
HASE-Dinero	Не	Не	Не	Не	Не	Да	IC	PL	?	Не	Не
JCachesim	Не	Не	Не	Не	Не	Да	BC	CL	TD	Не	Да
JHDL	Да	Да	Да	Да	Да	Да	IC	CL	ED	Не	Да
Logisim	Не	Да	Не	Не	Не	Да	IC	CL	?	Не	Не
M5	Да	Да	Не	Да	Да	Не	BC	IL	ED	Да	Да
RM	Да	Не	Не	Не	Не	Да	IC	CL	TD	Не	Не
RSIM	Не	Не	Не	Не	Не	Не	BC	CL	ED	Не	Не
SIMCA	Не	Не	Не	Не	Не	Не	BC	CL	ED	Да	Не
SimFlex	Не	Не	Не	Да	Не	Не	BC	CL	ED	Да	Не
Simics	Не	Да	Не	Да	Да	Не	BC	CL	ED	Да	Не
SimOS	Не	Не	Не	Не	Не	Не	BC	CL	ED	Не	Не
SimpleScalar	Не	Не	Не	Не	Не	Не	BC	PL	ED	Не	Не
VSDS	Да	Да	Да	Не	Не	Да	IC	CL	TD	Не	Не

Симулатори који омогућавају подршку дизајну међусобном се разликују по почетном скупу расположивих компоненти и процедура за њихово коришћење за креирање нових компоненти. Почетни скуп компоненти у случају Logisim симулатора обухвата основне логичка кола као што су И, ИЛИ, НЕ, Д флип-флопови и тростатички бафери, док је HASE, JHDL и VSDS укључују и сложене елементе као што су компаратори, бројачи, меморијских чипова, итд. Поред тога, ови симулатори омогућавају да се користе компоненте које је корисник дефинисао или користећи посебан језик за опис хардвера или посебне текстуалне датотеке.

Визуелна презентација, Контрола симулације, и Корак симулације су

међусобно повезани критеријуми. Значајан број симулатора са визуелном презентацијом (DLXview, EDCOMP, HASE, HASE-Dinero, JHDL, Logisim, RM, SMOK, VSDS) омогућавају интерактивни ток симулације не нивоу такта. DigLC2 и JCachesim омогућавају извршавање симулације на нивоу инструкције, јер њихов примарни циљ није презентација имплементационих детаља. Симулатори који пружају само текстуални интерфејс и пакетску обраду извршавају симулацију на нивоу читавог програма (SIMCA, SimOS), на нивоу инструкција (ANT, M5, SimFlex, Simics) и на нивоу такта (RSIM, SimpleScalar). Ови симулатори су чешће користи за анализе перформанси симулираних система него за наставу архитектуре и организације рачунара.

Хијерархијска репрезентација компонената је доступна код осам симулатора. Пет од њих (CPU Sim, DigLC2, DLXview, EDCOMP, RM) немају никакву подршку дизајну, док је преосталих пет (JHDL, M5, SMOK) пружају и подршку дизајну компоненти. CPU Sim, DigLC2 и RM дају хијерархијску представу имплементационих детаља RISC заснованог процесора, DLXview процесора са процесором са проточном обрадом, док EDCOMP даје опис детаља CISC заснованог процесора. Реализација симулатора који нуде овај вид подршке обично захтева знатне напоре како дизајна рачунарског система који треба да се симулира тако и самог симулатора који треба да га симулира. Може се закључити да је ова функционалност обично имплементирана само код оних симулатора где је тачно одређена архитектура и организација циљног симулираног система.

Учење на даљину је значајан тренд код савремених универзитетских курсеве. Као резултат тога, неколико симулатори (EDCOMP, HASE, JCachesim) су развијени као Веб базиране апликације. Ови симулатори су имплементирани користећи Јава програмски језик, који нуди различите технике погодне за приступ преко Интернета. У складу са овим трендом неки од раније развијених симулатора (HASE) су поново написани у Јави. Са развојем нових програмских оквира, пребацивање постојећих симулатора, као и развој нових симулатора доступних преко Интернета ће се повећати.

Евалуације резултата датих у Табели 4 показују да генерално сваки од симулатора нуди добар сет карактеристика које симулатор треба да поседује. EDCOMP, HASE, и M5 испуњавају већину критеријума. Овај ниво способности је

вероватно последица развоја на пољу софтверских технологија, које обухватају напредне визуелне алате, објектно оријентисану методологију, ефикасно конкурентно програмирање, Веб програмирање, итд.

2.2 Конкурентно и дистрибуирано програмирање

Да би се приступило прилагођењу симулатора архитектуре и организације рачунара за рад у конкурентном и дистрибуираном окружењу потребно је размотрити ову област као и начине на које је ова област изучавана на универзитетима широм света. У овој секцији се разматра преглед области конкурентно и дистрибуирано програмирање како би се дефинисао други домен проблема. Након прегледа области даје се преглед постојећих решења која се користе на лабораторијским вежбама из предмета конкурентно и дистрибуирано програмирање.

2.2.1 Преглед области Конкурентно и дистрибуирано програмирање

Приликом пројектовања симулатора из области Архитектуре и организације рачунара уочено је да се велики број симулатора извршава на једном рачунару унутар једног тока контроле, али је примећен и раст обима посла који симулатори треба да обаве. Тај обим посла је у директној вези са системима који се симулирају, који током година беже сталан раст. Да би се пратиле тенденције у развоју система који се симулирају потребно је студенте обучити и техникама које ће им помоћи да креирају симулаторе који могу да искористе потенцијале које пружају савремену рачунарски системи који омогућавају конкурентно и дистрибуирано извршавање како би се добио максимум перформанси. Због свега тога потребно је размотрити област конкурентног и дистрибуираног програмирања која студентима треба да пружи могућност развоја сложенијих симулатора Архитектуре и организације рачунара. Област конкурентно и дистрибуирано програмирање је одабрана из разлога што велики број алгоритама симулације захтева рад у конкурентном и дистрибуираном окружењу, па је потребно видети које су технике синхронизације и размене порука неопходне да би се развио одговарајући симулатор. Ово је посебно битно јер се посебна пажња даје на сам поступак пројектовања симулатора које студенти треба да науче.

Као и у случају области Архитектуре и организације рачунара удружења

ACM и IEEE су у оквиру своје заједничке радне групе дале предлог планова и програма наставе у оквиру студија рачунарске технике. Унутар ових планова и програма дате су и целине везане за конкурентне и дистрибуиране системе које би требало да се изучавају [2] и [3]. Ове препоруке су целине везане за конкурентно и дистрибуирано програмирање груписале у оквиру неколико области, и нису формирали само једну област. Ова расподела целина по областима је приказана у Табели 5.

Табела 5: **Преглед тема повезаних са конкурентношћу и дистрибуираним системима које су покривене различитим областима**

Тема		Област
Конкурентност	Процеси и нити	AL, CS, CAO, ES, OS
	Браве и баријере	CAO, ES, OS
	Синхронизација	CAO, ES, OS
	Алгоритми	AL, OS
	Семафори	OS
	Монитори	OS
	Условни региони	OS
Дистрибуирани системи	Синхронизација	AL, CS, CAO, ES, NC
	Алгоритми	AL, CS, CAO, CN, ES
	Размена порука	CS, CAO, CN, ES
	Удаљени позиви процедура	ES, NC
	Мрежно програмирање	NC
	Rendezvous	OS
	Grid Computing	CS, NC
	Веб сервиси	NC
	P2P	NC

Области: Алгоритми (AL); Рачунске науке (CS); Архитектура и организација рачунара (CAO); Рачунарске мреже (CN); Уграђени системи (ES); Рачунање засновано на мрежама (NC); Оперативни системи (OS).

Приликом пројектовања симулатора на неком од курсева самом том поступку је потребно посветити извесно време. То време је ограничено планом и програмом који је дефинисано или на нивоу предмета или на нивоу целог модула на коме се изучава. Ово ограничено време на курсу Конкурентно и дистрибуирано програмирање где студенти треба да се упознају са синхронизацијом и разменом порука које се могу искористити приликом пројектовања симулатора подразумева да скуп тема које се обрађују на лабораторијским вежбама треба да буду изабрани на такав начин да обезбеди што виши ниво када су у питању циљеви курса које студети треба да остваре [57]. Када се ради о курсу Конкурентног и дистрибуираног програмирања на Електротехничком факултету Универзитета у Београду, једна петина укупног времена на курсу је посвећена лабораторијским

вежбама. Да би се утврдило који теме из области Конкурентног и дистрибуираног програмирања, представљене у Табели 5 треба да буду покривене вежбама обрађени су курсеви на великом броју универзитета на којима је било могуће наћи наставне програме и детаље курсева. Универзитети су одабрани међу првих 100 универзитета са Академске листе рангирања светских универзитета (Academic Ranking of World Universities list) [58] која треба да процентуално одржава однос између региона света (Америка, Европа и Азија / Пацифик). Једини изузетак у одабиру универзитета је Универзитет у Београду. Преглед обрађених курсева са основних студија са детаљима у вези са покривене тема, карактеристикама и оцењивањем је дат у Табели 6. За универзитете који не нуде курс конкурентног програмирања одговарајући курс који се односе на оперативне системе је укључен. Детаљи курсева су прикупљени из брошуре и сајтова и представљају стање у области у тренутку претраге. Више података о овим курсевима се може наћи у секцији „Прилог В: Преглед наставе из конкурентног и дистрибуираног програмирања на првих 100 универзитета Шангајске листе“ (део материјала је био прикупљен у оквиру пројекта [59]).

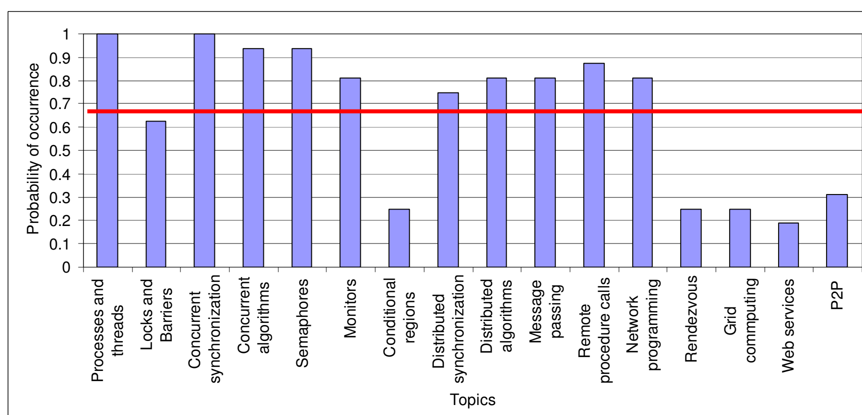
Уколико се посматрају теме коју се изучавају на курсевима који се баве облашћу Конкурентног и дистрибуираног програмирања на поменутим универзитетима може се претпоставити да теме које се чешће изучавају на тим курсевима треба сматрати важним за лабораторијске вежбе. Претпоставља се да се тема изучава на универзитету, ако постоји бар један курс који се односе на конкурентности и дистрибуиране системе на универзитету који има ту тему. На слици 3 дата је вероватноћу да је тема покривена на анализираним универзитетима. Приликом избора тема као гранична вероватноћа појављивања узет је праг на две трећине који као резултат избора даје пет тема из области конкурентног програмирања и пет тема из области дистрибуираног програмирања. Одабране теме испуњавају хоризонтални и вертикални критеријум, као и критеријум времена и смисла које су дате у оквиру принципа основних идеја [60]. Овако одабране теме омогућавају стицање знања неопходног за развој симулатора архитектуре и организације рачунара, али не прате трендове у развоју симулатора. Као једини изузетак од правила две трећине је узета област обраде у мрежи рачунара (grid computing). Тема везана за мреже рачунара је

укључена и због чињенице да су протоколи и технологије који се овде примењују искоришћене у великом броју академских, владиних и индустријских окружења, а и због чињенице да се је све већа количина литературе везаних за ову тему сакупљена током протеклих година [61].

Табела 6: Преглед курсева који покривају област конкурентности и дистрибуираних система на основним студијама на одабраним универзитетима

			Конкурентност							Дистрибуирани системи																		
Универзитет	Име курса	Укупно часова	Процеси и нити	Катанци и баријере	Синхронизација	Алгоритми	Семафори	Монитори	Условни региони	Синхронизација	Алгоритми	Размена порука	Удаљени позиви процедура	Мрежно програмирање	Rendezvous	Grid Computing	Веб сервиси	P2P	Програмски језици	Формални језици	Пројекти	Домаћи задаци	Лабораторија	Квизови	Присуство настави	Колоквијуми	Испит	Литература
Australian National University	Concurrent and Distributed Systems	38	+	+	+	+	+	+	+	+	+	+	+		+				ADA		0	30	0	0	0	10	60	1
Case Western Reserve University	Distributed Systems	43								+	+	+	+	+		+	+	+	C		20	30	0	0	10	20	20	2
	Introduction to Operating Systems	39	+		+	+	+	+	+	+	+		+	+					C		0	40	0	0	0	25	35	3
Hebrew University of Jerusalem	Operating Systems	24	+	+	+	+	+			+	+	+	+	+					C,C++		0	40	0	0	0	0	60	3
Massachusetts Institute of Technology	Distributed Computer Systems Engineering	20								+	+	+	+	+					C		40	25	0	25	10	0	0	4
	Operating System Engineering	8	+	+	+														C		0	20	50	30	0	0	0	5
Stanford University	Distributed Systems	16								+	+	+	+	+				+	C,C++		0	40	0	0	0	15	45	6
	Operating Systems	13	+	+	+	+	+					+	+						C		50	0	0	0	0	0	50	3
Swiss Federal Institute of Technology, Zurich	Concepts of Concurrent Computation	33	+		+	+	+	+											Java, Eiffel		50	0	0	0	0	0	50	7
	Distributed Systems	57	+		+					+	+	+	+	+	+	+	+	+	Java		0	20	0	0	0	0	80	6
University College London	Concurrent Programming	30	+		+	+	+	+			+	+	+	+	+				Java	+	0	15	0	0	0	0	85	8
University of Arizona	Parallel and Distributed Computing	40	+	+	+	+	+	+	+	+	+	+	+	+	+			+	C, Java, Linda	+	25	30	0	0	10	15	20	9
Универзитет у Београду	Конкурентно и дистрибуирано програ.	50	+	+	+	+	+	+	+	+	+	+	+	+	+	+			Java	+	20	0	10	0	0	35	35	9
University of Bristol	Concurrency with Java	40	+		+	+	+	+											Java, Occam	+	30	0	0	0	0	0	70	8
University of California, Berkeley	Operating Systems and Systems Programming	24	+	+	+	+	+	+					+	+					C		50	0	0	0	5	30	15	3
University of California, Davis	Operating Systems	14	+	+	+	+	+	+											C		25	25	0	0	0	25	25	10
University of Manchester	Concurrency and Process Algebra	22	+		+	+	+	+											Java	+	0	0	0	0	0	0	100	8
	Distributed Computing	32								+	+	+	+	+		+	+		Java		0	30	0	0	0	10	60	2
University of Rochester	Operating Systems	8	+	+	+	+	+	+											C,C++		20	30	0	0	10	20	20	3,4
	Parallel and Distributed Systems	15	+							+	+	+	+	+					C,C++, Java		0	40	0	0	0	25	35	-
University of Texas at Austin	Concurrent and Distributed Systems	47	+		+	+	+	+		+	+	+	+	+					Java		0	40	0	0	0	0	60	11, 12
Uppsala University	Distributed Systems	20								+	+	+	+	+				+	Erlang		40	25	0	25	10	0	0	6
	Operating Systems	14	+	+	+	+	+	+											C, Java		0	20	50	30	0	0	0	3

Укупно сати: Број сати на курсу за предавања, вежбе, туторисале, и лабораторијске вежбе посвећене областима означеним у табели.
Формални језици:Курсеви означени користе CSP или FSP формалне језике.
Литература: 1. M. Ben-Ari, Principles of Concurrent and Distributed Programming; 2. A. S. Tanenbaum and M. Van Steen, Distributed Systems: Principles and Paradigms; 3. A. Silberschatz, P. B. Galvin, and G. Gagne, Operating System Concepts; 4. A. S. Tanenbaum, Modern Operating Systems; 5. J. Lions. Lions' Commentary on UNIX 6th Edition with Source Code; 6. G. Coulouris, J. Dollimore, and T. Kindberg, Distributed Systems: Concepts and Design; 7. B. Meyer, S. Nanz, Concepts of Concurrent Computation; 8. J. Magee and J. Kramer, Concurrency: State Models & Java Programs; 9. G. R. Andrews, Foundations of Multithreaded, Parallel, and Distributed Programming; 10. A. S. Tanenbaum and A. S.Woodhull, Operating Systems Design and Implementation; 11. V. K. Garg, Concurrent and Distributed Computing in Java; 12. D. Lea, Concurrent Programming in Java.



Слика 3: Вероватноћа појављивања појединих области конкурентног и дистрибуираног програмирања на анализираним универзитетима који имају барем један курс повезан са наведеном области. Праг је постављен на 2/3

2.2.2 Преглед постојећих решења из области Конкурентно и дистрибуирано програмирање

У наставни плановима на курсевима који се баве конкурентним и дистрибуираним програмирањем скуп лабораторијских вежби је развијен тако да се студенти на дискретан и ограничен начин сусрећу са практичне проблеме [62]. Много труда је било уложено у приступима да се прикаже скуп формалних концепата на примеру CSP (Communicating Sequential Processes) или његова варијације FSP (Finite State Processes [63]) и као и концепти који се примењују у програмским језицима опште намене. Као најраспрострањенији алат који подржава FSP нотацију издваја се LTSA (Labelled Transition System Analyser) који омогућава концизан опис конкурентног понашања као и коришћење графичког приказа понашања компоненти [64]. За рад са традиционалном CSP нотацијом користи се CSPsim симулациони алат који омогућава кориснику да како уноси програм тако обавља оцену CSP процеса превођењем формалног описа користећи програмски језик Ада [65].

Приступ заснован на коришћењу формалних концепата подржава праволинијски развој апликација, али не нуди решења за ситуације које могу настати током саме имплементације. Решења која претежно имају за циљ да буду помоћ у настави основа објектно-оријентисаног програмирања, као што је Zero [67], обично су тако конципирана да им недостају аспекти који покривају теме из

конкурентног и дистрибуираног програмирања. Могуће решење за алате који покривају и ове концепте може наћи у алату као што је JThreadSpy који пружа могућности за прикупљање и приказивање редоследа по коме је обављено извршавање нити [66]. Други приступи за унапређење ефикасности наставе и учења конкурентног и дистрибуираног програмирања сугеришу коришћење интегрисаног окружења, као што су HiSAP [68]. Уз обезбеђивање високог нивоа интеракције, интегрисаним окружењима може недостајати довољан ниво флексибилности приликом увођења нових примера у скуп лабораторијских вежби. Сви наведени приступи лабораторијским вежбама се заснивају на коришћењу малих, независних задатака, где сваки задатак треба да демонстрира посебан концепт. Овако развијени мали задаци су често вештачки, имају за циљ да јасно истакну концепт, али се ретко могу применити у већој мери приликом развоја практичних и сложених решења. Начини да се помогне студентима да достигне виши ниво стручности у програмирању кроз лабораторијске вежбе се доста често огледа у развоју клијент-сервер апликација средњег нивоа сложености [69]. Други начин за развој лабораторијских вежби је да се користе веће апликација које комбинују већи број концепата из области конкурентног и дистрибуираног програмирања и показују како ови концепти интерагују заједно, као што је приказано са ShareMe апликацијом [70].

2.3 Предлог решења на глобалном нивоу

У овој секцији се разматра оквирни предлог развоја симулатора архитектуре и организације рачунара погодног за рад у конкурентном и дистрибуираном окружењу. Решење које се овде предлаже налаже коришћење слојевите архитектуре код које је сваки слој одговоран за различите облике комуникације и обраде. Предложено решење треба да се састоји из пет слојева: логичког, извршног, презентационог, симулационог, и слоја физике ([71]-[78]). Приликом формирања слојева треба водити рачуна о интеракцији између слојева.

Логички слој симулатора се односи на опис логичког понашања компоненти које се симулирају и њихове међусобне везе на нивоу понашање. Први корак у пројектовању симулатора представља одабир компонената и нивоа детаља са којим ће компоненте бити представљене. Компоненте су засноване на принципу коришћења приступних тачака за комуникацију. Користећи

компоненте, са својим приступним тачкама и везама које постоје између њих описује се понашање система. Компоненте обављају одређену функцију примањем улазних сигнала преко приступних тачака, трансформишу их, и генеришу нове сигнале који преко излазних тачака дистрибуирају даље. Дистрибуирање сигнала се обавља користећи везе између компоненти које омогућавају транспорт сигнала између од једне компоненте до друге. На овај начин је омогућено преносити поруке, догађаје и сигнале између логичких компонената.

Извршење слој симулатора треба да омогући кориснику да створи различите алгоритме извођења симулације. Овај слој треба да буде флексибилно направљен тако да омогући и догађајима вођену симулацију, као и временски вођену симулацију, као и њихове варијанте и подваријанте. Овај слој треба уско да сарађује са осталим слојевима како би се симулација непрекидно одвијала. Овај слој води рачуна о протеклом логичком времену симулације, као и обрађеним ситуацијама.

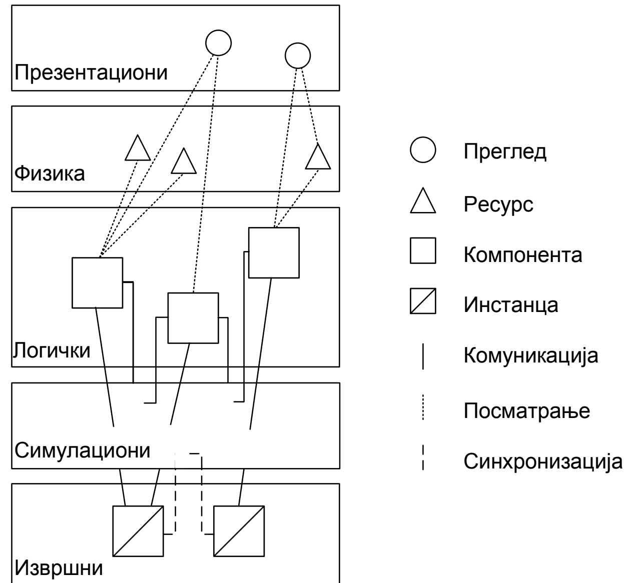
Презентациони слој симулатора треба да представља везу онога шта се симулира са корисником система. Овај слој треба да буде покретач свих акција које се догађају у симулатору. Преко овог слоја се креирају логичке компоненте и њихови интерфејси, повезује се компоненте и одређује њихов распоред, врши се одабир алгоритма симулације, интерактивно се уносе параметри симулације и прати сам ток симулације треба водити рачуна о томе да је време извршавања корака симулације упоредиво са временом потребним за приказ резултата, те је потребно имати што мању синхронизацију и међузависност између ових слојева.

Симулациони слој симулатора треба да обезбеди слој апстракције између инстанци извршног слоја симулатора и презентационог слоја симулатора. Ова апстракција се огледа у томе да компоненте логичког слоја не треба да буду свесне на ком се рачунару извршава код те компоненте већ симулациони слој треба да обезбеди виртуелну повезаност свих логичких компоненти. Ово је веома битно у ситуацијама када је потребно извршавати сложене синхронизационе алгоритме и интерактивно приказивати резултате симулације без довођења корисника у заблуду око спекулативно израчунатих вредности симулације.

Слој физике симулираних компоненти треба да обезбеди могућност

провере и израчунавања потребних физичких карактеристика које логичка симулација може да произведе. Ове карактеристике се обносе на описе физичких параметара које би реални системи имали, ови параметри се односе на опис напона, струје, напонског и струјног капацитета, дисипације и потрошње енергије, и конкретним случајевима симулације. Овај слој своја израчунавања заснива на параметрима логичког слоја симулације и може се сматрати проширењем овог слоја у смислу генерисана додатних резултата симулације.

Комуникација, синхронизација и интеракција између предложене архитектуре састављене од слојева намењене симулатору архитектуре и организације рачунара погодном за извршавање у конкурентном и дистрибуираном окружењу је представљена на слици 4. Компоненте логичког слоја међусобно комуницирају користећи симулациони слој. Овакво раздвајање омогућава да се компоненте логичког слоја распореде на већи број дистрибуираних рачунара а да саме компоненте не буде свесне дистрибуираности. На рачунарима се унутар инстанци извршног слоја обавља обрада компонената логичког слоја. Компоненте извршног слоја међусобно обављају синхронизацију на исти начин као што компоненте логичког слоја комуницирају користећи услуге симулационог слоја. На овај начин је остварено да компоненте извршног слоја не буду свесне да ли се ради о дистрибуираном или конкурентном извршавању. На основу резултата обраде и промењеног стања компонената логичког слоја обавља се обрада у слоју физике компонената пресликавањен стања логичке компоненте у физичке ресурсе. Резултат обраде у логичком слоју и слоју физике се приказује користећи презентациони слој.



Слика 4: Структура предложеног решења и зависност између слојева

3 ПРОЈЕКТОВАЊЕ СИМУЛАТОРА АРХИТЕКТУРЕ И ОРГАНИЗАЦИЈЕ РАЧУНАРА

Ова глава је подељена у две целине. Прву целину чине детаљни описи пројектовања сваког од слојева симулатора који би се користио у настави из пројектовања симулатора Архитектуре и организације рачунара као и настави из Конкурентног и дистрибуираног програмирања приликом паралелизације секвенцијалних апликација. Другу целину чине аналитички модели понашања описаног симулатора у зависности од изабраног алгоритма симулације и карактеристичних параметара симулираног система, на основу чега су идентификоване карактеристике симулатора са становишта могућности паралелизације.

3.1 ЛОГИЧКА СТРУКТУРА СИМУЛАТОРА

Архитектура и организација рачунара представља доста широку област рачунарске технике. Целине које су саставни делови ове области су идентификоване на основу IEEE/ACM препорука и представљене су у глави 2.1 „Преглед области Архитектура и организација рачунара“. На основу ових издвојених целина је базиран комплетан поступак пројектовања симулатора Архитектуре и организације рачунара. Поступак пројектовања се заснива на постојању извесног броја слојева унутар симулатора који треба да омогуће повезивање ових делова у једну конзистентну целину. Скуп слојева које симулатор поседује треба да чине целину приликом чијег пројектовања треба дати одговоре на неколико питања о понашању система. Приликом пројектовања логичког слоја треба дати одговор на то шта симулатор треба да ради, извршни слој треба да одговор на то како симулатор треба да ради, приликом пројектовања симулационог слоја треба даје одговор на питање где симулатор треба да ради, приликом пројектовања презентационог слоја треба дати одговор на питање ко је објекат који се симулира односно ко ће посматрати дати објекат, и приликом пројектовања физичког слоја треба дати одговор на питање зашто се систем понаша тако како се понаша и имали то неких последица на извођене симулације.

Слојеви који чине симулатор чине један целину, али у зависности од примене неки од слојева се могу изоставити. Језгро симулатора који се користе у настави из Архитектуре и организације рачунара чине логички извршни и презентациони док се преостала два слоја користе само у посебним случајевима. Симулациони слој се користи када је потребно извршити временски захтевну симулацију на великом броју рачунара, и физички слој када је потребно утврдити да ли је то што се симулира могуће реализовати користећи компоненте које имају одговарајуће физичке карактеристике.

3.1.1 Пројектовање логичког дела симулатора

Логички слој симулатора представља основну везу симулатора према одређеној области. У овом случају ради се о области архитектуре и организације рачунара. Приликом пројектовања логичког слоја симулатора треба дати одговор на питања шта симулатор треба да симулира. Одговором на ово питање дефинише се домен проблем. Домен проблема се обноси на то које компоненте треба да чине симулатор. Одговор на ово питање треба потражити у анализи која је начињена на почетку овог рада где је разматрана област архитектуре и организације рачунара и већ развијених симулатора. Приликом ове анализе уочене су две веће групе симулатора које описују понашање симулираног система користећи два различита приступа. Један приступ даје одговор на питање шта симулирана компоненте ради, не улазећи у детаље имплементације док други приступ даје одговор на питање како одређена компонента ради. Одговори на ова питања су релативни јер се у сваком кораку може поставити исто питања шта и како и прећи на следећи ниво апстракције. Ове уочене групе се могу поделити и на већи бој подгрупа у зависности од траженог нивоа детаља. Прелазак на ниже нивое апстракције је могућ и остварује се увођењем посебних слојева који се ослањају на постојеће нивое, што је касније у овом раду показано на примеру увођења слоја физике компонената.

Симулатори који су били разматрани у овом раду су посматрали компоненте које се користе у области архитектуре и организације рачунара са два нивоа апстракције приликом извршавања. Ови приступи су примењивани како би се од корисника сакрио известан ниво детаља, али и да би се време симулације учинило што краће. Симулација код које су градивни блокови реализовани на

високом нивоу апстракције су се веома брзо извршавале, али је ниво детаља које пружају кориснику мали, док су симулације са великим бројем детаља биле доста споре за извршавање. Може се приметити да се у извесном броју случајева примењивала и хибридна техника која је користила функционални опис компонената у неком временском периоду дајући том приликом статистички поуздане податке и могућност преласка на детаљан мод рада у коме се добијају тачни симулациони подаци, али у овом случају је време обраде било неупоредиво дуже. У овом раду је фокус на ова два приступа, уз предложен механизам за прелазак на неки наредни слој апстракције.

3.1.1.1 Компоненте логичког слоја

У овој секцији се разматрају компоненте логичког слоја које се појављују у области Архитектура и организација рачунара. На основу IEEE/ACM радне групе ова област је подељена на следеће образовне јединице: Основни принципи архитектуре рачунара, Архитектура и организација меморијских система, Интерфејси и комуникација, Улазно/излазни уређаји, Дизајн система процесора, и Организација процесора. На основу анализе расположивих симулатора и покривености који ону дају према одређеним образовним јединицама уочено је да они приликом пројектовања компонената имају следеће нивое апстракције: Ниво логичких компонената, Ниво стандардних модула, Ниво функционалних јединица, и Ниво извршних јединица. Ови нивои као и компоненте расположиве код ових типова симулатора су приказани у Табели 7.

Табела 7: Нивои апстракције компонената логичког слоја

Нивои компонената	Расположиве компоненте
Ниво логичких компоненти	Логичка кола Флип-флоп
Ниво стандардних модула	Стандардни комбинациони модли Стандардни секвенцијални модули
Ниво функционалних јединица	Функционалне јединице (ALU, CPU)
Ниво извршних јединица	Машинске инструкције
Ниво структурне јединице	Систем

Компоненте које се корите на нивоу логичких компоненти треба да по својој структури одговарају најмањим градивним блоковима који се користе у

области архитектуре и организације рачунара. Ова коле која се корите за логичко пројектовање сложенији рачунарских система би треба да укључују логичка кола је два и више улаза (And2, Or2, Nand2, Nor2, Xor2, Xnor2, And3, Or3, Nand3, Nor3, Xor3, Xnor3, And4, Or4, Nand4, Nor4, Xor4, Xnor4, Not1, Buff1, Tri1), као и стандардне меморијске елементе флип-флопове (DFF, RSFF, TFF, JKFF, DMSFF, RSMSFF, TMSFF, JKMSFF и DEFF).

Преласком на сваки следећи ниво апстракције треба се обезбедити делимично сакривање имплементациоих детаља. Ово подразумева да постоји поступак како се користећи градивне блокове дефинисане на једном нивоу може формирати градивни блок следећег нивоа. Градивни блокови следећег нивоа не морају бити реализовани на поменути начин све докле постоји поступак како се могу дефинисати. Ово је битно на гласити зато што време потребно за развој и извршавање зависи од нивоа на коме је поједина компонента дефинисана. Следећи ниво који се користи у симулаторима у архитектури и организацији рачунара описује понашање система на нивоу трансфер података међу регистрима. Компоненте које се овде користе су састављене од основних компонената и могу се даље користити као основни блокови у развоју сложених целина. Међу ове компоненте спадају стандардни комбинациони (мултиплексер, демултиплексер, кодер, декодер, елементарни сабира) и стандардни секвенцијални модули (регистри, бројачи, померачи, регистри са више функција).

Користећи овако формиране градивне блокове могуће је реализовати поједине сложене јединице које могу да обављају доста сложене функције. На овај начин се реализују компоненте следећег слоја које обављају појединy функцију унутар сложених рачунарских система. Та функција може да реализована као аритметичко логичка јединца или појединих јединца процесора, или читавог процесора, или компонента из хијерархијског меморијског подсистема.

Највиши ниво у хијерархији када се ради о облати архитектура и организација рачунара заузимају компоненте које реализују инструкције и описе читаве архитектуре рачунарског система. На овом нивоу процесор се посматра само на нивоу архитектуре рачунара занемарујући организацију и времена извршавања тако да корисник најчешће вид само машинске или асемблерске

инструкције. Овај ниво се обично користи за симулирање комплексних система, али се симулатори на основу статистичких параметара труда да дају прецизну симулацију у погледу времена извршавања.

3.1.1.2 Везе логичког слоја

У овој секцији се разматрају везе између компонената логичког слоја симулатора које се појављују у области Архитектура и организација рачунара. Везе треба да обезбеде комуникацију и размену порука између компонента на одређеном нивоу апстракције и треба да пронађу како је реализована размена порука и шта та порука од информација носи. За разлику од компонента које обрађују примљене податке везе служе за транспорт података оне не обављају никакву обраду над примљеним подацима. На основу поделе на нивое сложености компонената и везе између њих се могу поделити на исте нивое. Ови нивои, везе између компонената расположиве код ових типова симулатора су представљени у Табели 8.

Табела 8: Нивои апстракције веза логичког слоја

Нивои компонената	Расположиве везе
Ниво логичких компоненти	Размена сигнала
Ниво стандардних модула	Размена група сигнала Коришћење компоненте везе
Ниво функционалних јединица	Коришћење дељених променљивих
Ниво извршних јединица	Позиви функција
Ниво структурне јединице	Коришћење готових симулатора

Везе које повезују логичке компоненти треба да обезбеде комуникацију између дигиталних компонента. Ове вредности најчешће спадају у скуп {0, 1}, {0, 1, X}, као и {0, 1, X, Z}. Везе које се формирају између компоненти логичког слоја полазе од излаза једне компоненте до улаза друге компоненте и типови података који се размењују на оба краја су најчешће исте. Уколико се типови изворишта и одредишта разликују онда је могуће одрадити имплицитну конверзију типа излаза једне компоненте у тип улаза у другу компоненту.

Преласком на сваки следећи ниво апстракције симулатори реализују делимично сакривање детаља имплементације и структуре порука који се

размењују. Ово подразумева да постоји поступак за конверзију и размену порука дефинисаних на различитим нивоима. Нивои апстракције порука које се размењују између компонената полазе од типова који су блиски хардверу и полако прелазе на типове података који с могу лако имплементирати у софтверу помоћу кога се симулација обавља. Следећи ниво који се користи у симулаторима у архитектури и организацији рачунара описује понашање система на нивоу трансфер података међу регистрима. Ови трансфери очекују често коришћење магистрале преко које се подаци транспортују, као и правила за разрешавање конфликтних ситуација на магистрали. Вредности које се најчешће размењују спадају у скуп $\{0, 1\}^n$, $\{0, 1, X\}^n$, као и $\{0, 1, X, Z\}^n$ где n представља ширину податка који се транспортује. Приказ ових података се обавља на нивоу датог скупа вредности, али се врло често вредности исказују као бинарне, децималне или хексадецималне како би се кориснику олашкао рад унутар симулатора. Пошто се компоненте усложњавају овде се користе и бидирекционе везе и то реализоване као магистрала и веза елемената са отвореним колектором. Симулатори који користе компоненте дефинисане на овом слоју поред тога што користе сложене компоненте морају да формирају и правила интерпретације у случају вишеструког постављања вредности на исту везу. Резолуција конфликата се интерно решава коришћењем посебних компоненте или прекидањем симулације о обавештавањем корисника да је дошло до грешке.

Увођењем ових посебних компоненте симулатор почиње да добија делове који престају да описују само шта симулирани систем треба да ради, већ како је нешто реализовано. Са преласком између слојева хијерархије све више се смањује број компонената које описују како поједини делови система функционишу, а појављују се компоненте које омогућавају да се сама симулација обави што брже. Преласком на коришћење градивних блокова који обављају сложене функције ниво сложености порука које се размењују расте тако да компоненте више не користе директне везе преко којих се поруке размењују већ користе заједничке дељене променљиве које описују комуникацију. Вредности које се размењују носе информацију о томе која функционална јединица има коју вредност, односно могуће је приступити променљивама које описују комплетно њено стање.

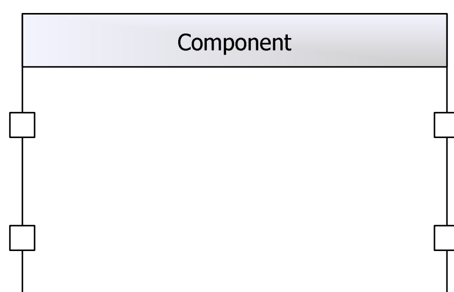
Највиши ниво заузимају компоненте које реализују инструкције и описе

читаве архитектуре рачунарског система тако да комуникација између ових компоненти реализована на позиву функција које размењују већи број комплексних структура података. Пошто се на овом нивоу компоненте посматрају на нивоу архитектуре поруке које компоненте размењују описују како је симулатор развијен а не како се понаша систем који се симулира.

Везе које се користе унутар симулатора могу да повезују већи број компонената у хијерархији. Да би се кориснику олакшао рад потребно је дозволити да се вези може доделити име. Приликом ове доделе имена треба водити рачуна о имену које се једна иста логичка веза може протезати између већег броја компонената унутар хијерархије. Ово је такође битно у ситуацијама када се унутар једне компоненте исти сигнал може прослеђивати са улаза на излаз без икакве обраде.

3.1.1.3 Спајање логичких компонената и логичких везе

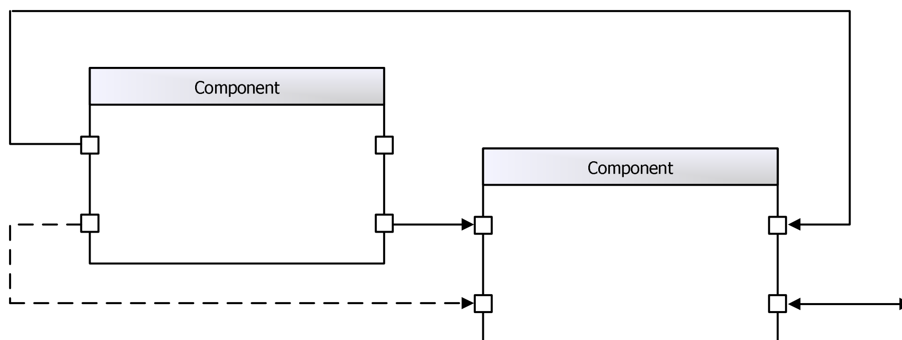
Приликом пројектовања симулатора архитектуре и организације рачунара потребно је водити рачуна о томе шта симулатор треба да симулира, односно о томе како треба да изгледа симулирани систем посматран из угла корисника. Овде се под изгледом компонената подразумева визуелна презентација компонената, али и могућност праћења и приступа хијерархијској репрезентацији компонената. Да би се обезбедило да компоненте симулираног система могу да комуницирају користећи везе између компонената потребо је представити како изгледа принципска шема симулиране компоненте. Принципска шема логичког слоја компоненте је представљена на слици 5 и састоји се дела који ради обраду, односно остварује понашање компоненте и приступних тачака помоћу којих се врши размена порука између компонената.



Слика 5: Принципска шема компоненте логичког слоја

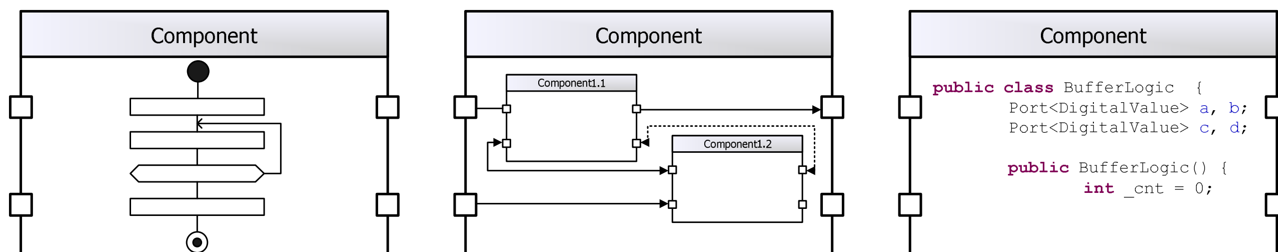
Компонента на основу података (порука, сигнала, вредности) примљених преко приступних тачака израчунава како ће се дата компонента понашати, односно како ће се променити њено интерно стање и како ће утицати на остале компоненте у симулираном систему користећи приступне тачке. Поступак израчунавања стања компоненте зависи од начина на који је компонента пројектована, и функције коју дата компонента обавља. Унутрашње стање компоненте може да обухвата делове који се односе на опис хардвера симулираног система, али и поступак симулације. Приликом поступка симулације у неким случајевима је потребно чувати комплетну историју понашања компоненте, али и вођење рачуна о деловима који описују текуће стање система које кориснику омогућава да лакше прати и посматра ток симулације. Уколико се ради о компоненти најнижег слоја хијерархије онда је опис компоненте дат неким високим програмским језиком и постоји само у симулатору. Уколико се ради о компоненти на неком од виших слојева хијерархије онда се комплетан поступак израчунавања обавља на основу задате структурне шеме која се састоји из компонента логичког слоја и веза између њих. Компоненте дефинисане на овом нивоу су лишене информације о протеклом логичком и физичком времену тако да је тај податак сакривен.

Компоненте се међусобно повезују користећи везе које спајају приступне тачке. Приликом поступка повезивања дефинишу се типови порука које компоненте на логичком нивоу међусобно размењују, као извориште и одредишта поруке. Карактер приступних тачака доста зависе од нивоа апстракције примењеног за пројектовање дате компоненте, али овај карактер у већини случајева доста верно прати понашање хардвера тако да се може уочити више типова приступних тачака: један број који представљају скуп улаза у компоненту, један број који представљају излазе из компоненте, известан број који су и улазни и излазни. На слици 6 је дат скуп логичких компонента повезаних логичким везама. Компоненте су представљене на апстрактном нивоу без улажења у њихову имплементацију. Овај поступак повезивања компонената треба да обезбеди креирање листе и начина повезивања компонената (нет листа) коју могу да користе остали слојеви симулатора. Симулатори треба да омогуће и чување и учитавање логичких компонената и веза на основу дате листе.



Слика 6: Приказ повезаних логичких компонената користећи логичке везе

Да би се омогућило креирање сложених компонената симулатори треба да омогуће да се независно обави креирање интерфејса компоненте, који је видљив осталим компонента, и његове унутрашње имплементације. Потребно је обезбедити да компонента може да има већи број различитих имплементација, али тако да њена спољашност остане непромењена. Уколико се развој компонената обавља користећи објектно оријентисани концепт приликом реализације различитих имплементација компонената може се применити особина наслеђивања. Користећи ову особину постоји могућност мењање неких особина оригиналне компоненте уз задржавање потписа метода класа, односно интерфејса према другим компонентама. На слици 7 је дат пример компоненте која поседује интерфејс који чине четири приступне тачке према другим компонентама, а притом поседује више различитих имплементација. Пошто је унутрашња структура сакривена од осталих компонената могуће је направити већи број реализације датог интерфејса. На истој слици су приказане три имплементације истог интерфејса који су имале компоненте сачињене према истој спецификацији, али користећи различите поступке. Прва реализација компоненте је креирања користећи посебан језик за опис хардверских структура и не поседује унутрашњу хијерархију. Друга имплементација се заснива на хијерархијској репрезентацији према истом опису користећи стандардна логичка кола. Трећи опис се заснива на оптимизацији другог решења и према логичким карактеристикама представља идентично решење. Сва три решења су међусобно еквивалентна на логичком нивоу, али се разликују и на нивоу имплементације, али и на нивоу других карактеристика које нису директно видљиве унутар овог слоја.



Слика 7: Раздвајање интерфејса понашања од реализације

Креирање компонената које користе симулатори архитектуре и организације рачунара се може поделити на три фазе. Прва фаза представља креирање основних градивних блокова користећи описе компонената који се појављују у области архитектуре и организације рачунара. Приликом формирања компонената у овој фази треба водити рачуна о томе да се креирани градивни блокови могу користити у даљем поступку интеграције унутар симулатора, односно да могу да чине библиотеку компонената. Друга фаза представља груписање компонената тако да чине извесну целину која се могу користити у појединим подобластима. Овде је потребно користећи основне блокове формиране у првој фази креирати сложеније, хијерархијске компоненте које директно одговарају већим целинама које се примењују у овој области. Овај поступак може да иде кроз више нивоа хијерархије сходно могућим описима компонената и веза који су дати у Табелама 7 и 8. Као трећа фаза јавља се потреба за тестирање и верификацијом понашања које развијене компоненте имају. У овом кораку се проверава да ли компоненте чији је опис задат користећи више различитих имплементација има конзистентно понашања, односно колико износе губици у нивоу информација које поједини слојеви могу да пруже.

3.1.1.4 Верификација компонената логичког слоја

Приликом рада са компонентама које се користе у области архитектуре и организације рачунара потребно је извршити верификацију да ли задате компоненте обављају специфицирану функцију и да ли је та функција описана са довољним нивоом детаља. Приликом верификације треба водити рачуна о вредностима које улазни сигнали/поруке могу да имају, односно ком домену припадају. Поступак верификације компонената логичког је један од основних поступака у креирању дигиталних уређаја који се срећу у области архитектуре и организације рачунара. Циљ верификације је да осигура функционалну

исправност предложен компоненте логичког слоја, али и потпуног поштовања свих детаља спецификације.

Поступак симулационе верификације има за циљ да омогући откривање грешака насталих у дизајну компонената тако што се посматра одзив компоненте на ограничени скуп улазних вектора. Поступак одређивање улазног скупа вектора је сложен поступак уколико се жели пронаћи минималан скуп улазних вектора. Приликом одређивање улазних вектора треба водити рачуна о томе да ли постоји грешка приликом пројектовања софтвер симулатора или приликом пројектовања компоненте логичког слоја. Као и у области тестирања софтвера и у овој области је развијено више поступака верификације заснованих на коришћењу метрика тестирања. Ове метрике се могу сврстати у следеће категорије: метрика заснована на покривању комплетног скупа улазних вектора, метрика делимичног покривања заснована на коришћењу коначних аутомата, метрика заснована на посматрању понашања система, као и метрика зависна од конкретног пројекта. Да би се омогућила симулациона верификација потребно је на неки начин омогућити да се током симулације могу посматрати интерна стања свих компонената логичког слоја, као и стања на логичким везама које транспонује сигнале између компонената. Поступак тестирања је доста сличан тестирању хардверских компонената, али са том разликом што није потребно убацивати посебне компоненте логичког слоја које обављају тестирање већ се уместо тих компонената користе особине симулатора.

Поступак симулациона верификације се заснива на посматрању излаза и стања које даје симулација са очекиваним излазима и стањима. Ови очекивани изрази се код представљених метрика задају користећи скупове логичких услова или вредностима који описују стања компонената која у поступку симулације не би требала да не могу да буду достигнута. Ово значи да се унутар кода логичких компонената уместо хардверских компонената постављају софтверски услови који проверавају да ли се симулатор налази у неком недозвољеним стању, односно да ли се појавила нека недозвољена комбинација улазних вектора и вектора који описују стање. Уколико се таква вредност појави најчешће је потребно прекинути симулацију и корисника обавестити да је се појавила грешка у дизајну. Поступак симулације се не мора прекидати приликом откривања

грешака, већ се за дату логичку компоненту може формирати посебна секција кода које обавља опоравак од грешака. Ове секције, као и делови кода који се користе приликом тестирања компонената треба да буду постављени у симулатору независно од кода компоненте, ово значи да код компоненте не треба да садржи делове за њено тестирање већ симулатор треба да омогући креирање посебних тест компонента које могу да посматрају интерно понашање компонената логичког слоја.

Поред испитивања унапред задатог скупа улазних вектора и очекиваних одзива на дате векторе поступак верификације се може засновати на поређењу резултата добијених за користећи описе исте компоненте дате на различитим нивоима апстракције. Поступак верификације се у овом случају заснива на поређењу резултата које свака од компонената даје у појединим ситуацијама. Овај поступак полази од претпоставке да је имплементација рачунарског система дата користећи хијерархијску репрезентацију компонената, док је спецификација система дата на нивоу функционалног описа. Пошто се код компонената логичког слоја не посматра кашњење већ се компоненте посматрају као идеалне онда овакав поступак не представља деградацију општости.

3.1.1.5 Чување компонената логичког слоја

Скуп компонената логичког слоја које се користе у области архитектуре и организације рачунара је потребно чувати у посебне библиотеке компонената како би могле да се користе унутар већег броја симулатора. Библиотеке компонената треба да омогуће чување често коришћених компонента, претходно формираних компонента, као и скупове комерцијално доступних компонената. Коришћење готових библиотека компонената има за циљ да убрза поступак пројектовања сложених хијерархијских компонената, као и да убрза поступак тестирања и валидације јер у већини случајева библиотечке компоненте није потребно независно тестирати већ је потребно обављати интегрално тестирање.

Компоненте које се чувају унутар библиотека моду да имају различиту структури и функцију тако да се могу поделити на више различитих категорија. Ове компоненте по својој сложености треба да омогуће описивање постојећих компоненти архитектуре и организације рачунара и варирају од једноставних логичких кола до описа читавих система. Категорије компонента су тако

постављене да могу описују понашање логичких компонената на различитим нивоима хијерархије сходно Табелама 7 и 8.

Прву категорију чини основни градивни блокови који се користе у области логичког пројектовања у архитектури и организацији рачунара. Логичке компоненте које се могу наћи унутар ове категорије припадају основним логичким колима. Поступак синтезе основним логичких кола је код већине анализираних симулатора био такав да су њихови описи потпуно сакривени од корисника. Ове компоненте се обично реализују тако да буду директно имплементиране у неком високом програмском језику, а да описи тог понашању буду недоступни крајњем кориснику.

Другу категорију чине хијерархијски дефинисане компоненте које су међусобно повезане са другим логичким компонентама како би формирале више нивојске структуре сходно својој сложеност. Ова група компонената је најбројнија јер симулатори архитектуре и организације рачунара имају за циљ да корисницима прикажу како неки дигитални систем функционе, односно у којој вези стоје његови основни делови. Компоненте које се користе унутар хијерархијске репрезентације могу да припадају свим категоријама, али су обично само најнижи слојеви дефинисани користећи неки више програмски језик, док су остали слојеви формирани хијерархијским повезивањем.

Трећу групу компонента чине конфигурабилне компоненте чије понашање зависи од задатог параметра. Ова група компонента може да обухвата најједноставније компоненте логичког слоја као што су логичка кола са више улаза, али и читаве сложене организације рачунара. Ове компоненте захтевају посебне начине чувања јер поред само описа компоненти оне у неким случајевима могу бити и иницијализоване тако да је потребно сачувати и оригиналну компоненту, али и параметре која та компонента може да прими. Од наведених параметара понашање и изглед компоненте може у многоме да зависи. Овде се под изгледом компоненте подразумева листа повезаних компонената унутар логичке мреже, која такође може да зависи од ових параметара. Компоненте овог слоја могу да чине и компоненте претходних слојева дате текстуалним и хијерархијским описом.

Као четврта група компоненти логичког слоја издвајају се високо

параметризоване компоненте код којих параметар који треба поставити није једноставан већ захтева известан скуп других логичких компонената. Ова група компоненти се користе за моделовање сложених архитектура и организација рачунара и представљају шаблонске архитектура. Коришћење шаблонских архитектура помаже приликом формирања конкретних инстанци архитектура на тај начин што је за поступак даљег пројектовања довољно узети постојећи шаблон и у њега унети све потребне параметре. На та начин је формирана нова хијерархијска компоненте које се надаље може самостално користити.

Као посебна врста параметризовани компоненти издваја се пета група компоненти која обезбеђује коришћење спољашњих симулатора у поступку симулације. Ови спољашњи симулатори могу да буду софтверска и хардверска решења. Ова група компоненти омогућава коришћење већ развијених компоненти у неком другом систему и њихово коришћење у ономе који се развија. Приликом пројектовања компонената ове групе потребно је обавити пројектовање адаптера ка одговарајућем симулатору код којег постоји опис дате компоненте. Адаптер треба да обезбеди да компонента има све спољашње особине компоненте која се симулира а позиве прослеђивати наменском софтверском или хардверском симулатору.

3.1.2 Пројектовање извршног дела симулатора

Извршни слој симулатора представља средишњи део симулатора који води рачуна о извршавању симулације и о протоку времена унутар симулације. Алгоритми симулације се развијају независно од области у којој се могу примењивати, али њихове перформансе доста зависе од конкретне области. Приликом пројектовања извршног слоја симулатора треба дати одговор на питања како симулатор треба да обавља симулацију. Одговором на ово питање дефинише се домен проблем. Домен проблема се обноси на то који алгоритми симулације треба да буду реализовани унутар симулатор. Одговор на ово питање треба потражити у анализи која је начињена на почетку овог рада где су разматрани симулатори у област архитектуре и организације рачунара код којих је доступан изворни код. Приликом анализе симулатора уочене су две веће групе симулатора које описују начин на који симулатори обављају симулацију а користе два различита приступа. Један приступ даје одговор на питање шта симулирана

компоненте ради у неком, произвољном временском тренутку, не улазећи у детаље имплементације протока времена док други приступ даје одговор на питање како одређена компонента реагује када се на неком од њених улаза догоди нека промена. Ове уочене групе се могу поделити и на већи број подгрупа у зависности од траженог нивоа детаља и конкретне имплементације компонената логичког слоја.

Основни задатак извршног слоја симулатора је одређивање тренутка када је потребно израчунавање новог стања система, односно вођењу рачуна о протеклом времену од започињања симулације. Понашање пројектованих логичких компонената и веза се испитује симулацијом анализом у коју су унети параметри. Да би се покренула симулација неопходно је формирати скуп улазних вектора и њихову временску зависност којима се описује понашање улаза у систем. Оно нашта се извршни део симулатора концентрише се односи на начин моделовања протеклог времена, алгоритмима симулације, њиховој оптимизацији да би се омогућило паралелно извршавање симулације, као и пројектовање комуникације са интерактивним корисником.

3.1.2.1 Моделовање времена

Симулатори који се користе у области архитектуре и организације рачунара се разликују по својој сложености, али када се ради о моделовању времена могу се поделити на два основне групе. Прву групу чине они симулатори који немају појам о симулационом времену већ само о томе да ли се ради о обрада комбинационих или секвенцијалних мрежа, и на оне који време моделују дискретним временским тренутке. Свака од овних група се дели на две подгрупе у зависности од нивоа детаља који дати модел пружа.

- Модел јединичног кашњења кроз компоненте полази од претпоставке да су логичке компоненте тако моделоване да је промени сваког улазног сигнала потребно исто, јединично, времена да постане видљива другим логичким компонентама. Ово значи да се свака промена улазних сигнала тачно након једне јединице времена рефлектује на излаз те компоненте. Овај апстрактан модел се користи у случајевима када постоји јединствен начин обиласка графа логичке мреже. Симулатори који имплементирају овај модел не захтевају информацију о протеклом времену јер претпостављају да се комплетан посао могао реализовати

у представљеном временском интервалу. Информација о узрочно последичној повезаности је саставни део графа обиласка мреже.

- Модел одвојеног јединичног кашњења кроз компоненте полази од претпоставке да све комбинационе мреже имају исто кашњење и да све секвенцијалне мреже имају исто кашњење. Ово значи да се одвојено разматрају промене на улазима комбинационих и на улазима секвенцијалних мрежа. Ово захтева да симулациони алгоритам разликује комбинационе и секвенцијалне мреже. Као и модел јединичног кашњења и овај модел захтева доста апстракције и користи се у случајевима када је познат јединствен начин обиласка графа комбинационих мрежа и графа секвенцијалне мреже. Код ових симулатора обрада се ради тако што се прво обраде излази свих комбинационих мрежа у одговарајућем поретку, па онда обрада секвенцијалних мрежа, у такође стриктном поретку.

- Модел фиксног кашњења кроз компоненту полази од претпоставке да свака компонента између било ког улаза и излаза има своје време пропагације. Ово време може бити различити за сваку компоненту, или сваки пар улаз-излаз, али је фиксно у току саме симулације за једну логичку компоненту. Овакво моделовање кашњења кроз компоненту омогућава доста прецизнију анализу понашања система у односу на претходне моделе. Овакав модел захтева другачије симулационе алгоритме јер се обрада заснива на догађајима, а не на униформном протоку времена.

- Модел промењљивог кашњења кроз компоненту полази од претпоставке да време пропагације не зависи само од улаза и излаза већ и од вредности која се на тим улазима и излазима поставља, као и броја компонената које су повезане на те улазе и излазе. Овај модел пружа могућност да се на прекидачком нивоу моделују понашања компоненти у зависности са које вредности на коју прелазе и колико је компонената том тренутку везано на излаз извора. Модел служи за прецизније моделовање паразитивне капацитивности које постоје у дигиталним систему као и карактеристике мреже када једна логичка компонента погони више других (fan-out). Овај начин моделовања се најчешће користи приликом моделовања компонената физичког слоја.

Посебан модел који се ређе користи у симулаторима архитектуре и организације рачунара је модел континуалног времена. Овај модел се најчешће

употребљаван приликом анализе аналогних компоненти на најнижем нивоу апстракције. У овом моделу се води рачуна о аналогним величинама као што су напона и струје током времена до чијих се вредности долази решавањем скупа диференцијалних једначина. Везе које владају између елемената су најчешће описи неких природних закона попут Кирхофових закона за електрична кола, или Њутнових закона механике за динамику кретања тела. Овај модел се најчешће користи приликом моделовања компонената физичког слоја. Овај модел се користи у завршним фазама тестирања компонената јер захтева велико време да би се израчунали сви излази компонената и решиле све диференцијалне једначине за довољно велики скуп излазних вектора.

3.1.2.2 Алгоритми секвенцијалне симулације

Алгоритми симулације се могу пројектовати независно од области у којој се примењују, али се ти алгоритми могу додатно модификовати како би се прилагодили области. Основна ствар о којој алгоритми треба да воде рачуна је да обезбеде проток времена унутар симулације. Проток времена се може посматрати на основу количине посла коју је потребно обавити не улазећи у специфичности појединих компонената, и други начин је да се време посматра само у карактеристичним тренуцима када има потребе да се уради неки посебан посао. Уколико се симулација заснива на униформном протоку времена када је потребно урадити неки посао онда се ради о временом вођеној симулацији (Time Driven Simulation), а у случају да није потребно пратити униформно протекло време већ је потребно посматрати само тренутке када је потребно урадити известан посао онда са ради о догађајима вођеној симулацији (Event Driven Simulation). У секцијама које следе биће представљена ова два основна алгоритма симулације.

3.1.2.2.1 <u>Временом вођена симулација</u>

Један од првих алгоритама симулације који је развије је временски вођена симулација где се симулације може посматрати као скуп сукцесивних временских интервала у којима је потребно обавити извесне симулационе кораке. Овај облик симулација је проистекао из симулације физичких процеса код кога се инкрементално прелазило на следеће стање симулације уз израчунавање нових вредности унутар датог корака. Код овог типа симулације се подразумева да се улази односно посматране вредности неће мењати између корака симулације већ

само у завршним фазама сваког корака у којима се обавља чување промена. Овај алгоритам обухвата две фазе. У првој, припремној фази се ради постављање почетних параметара симулације док се у другој, итеративној, фази симулације обавља израчунавања нових вредности стања компонената. При израчунавању стања сваке компоненте одређује се стање у које ће прећи, та нова стање се уписује у други скуп променљивих, и тек се на крају обраде чувају унутар компоненти.

```
algorithm TimeDrivenAlgorithm(p : list of components);
initialize(p);
preCalculate(p);
executionTime := startTime;
while (executionTime < endTime) do
      foreach component in p do
            call component execute;
      end_for;
      foreach component in p do
            save component new state;
            distribute component state to connected components;
      end_for;
      executionTime := executionTime + interval;
end_while;
```

Слика 8: Алгоритам временски вођене симулације код јединичног кашњења

Симулациони алгоритам који описује временски вођене симулације у симулаторима који користе јединично кашњење је приказан на слици 8. У првој припремној фази се обављају две активности везане за иницијализацију симулације. Прва активност се односи на учитавање улазних вектора и постављање свих почетних параметара симулације. Након ове активности обавља се активност почетног израчунавање свих излаза датих компонената како би се успоставило почетно стање симулације. Ово је неопходно јер почетни улазни вектори не садрже све вредности улаза компонената које су потребне да би симулација започела већ само иницијални скуп вредности након тога је потребно обавити иницијалну евакуацију излаза. Након припремне фазе прелази се на петљу у којој се извршава симулација. У свакој итерацији симулационе петље се могу уочити три корака обраде. У првом кораку се на основу улазних вредности израчунава ново стање у које систем прелази. Ново израчунато стање компоненте може да утиче на промене излазних вредности из дате компоненте. Промена излазних вредности захтева и евалуацију свих модула са којима је дати излаз

повезан. Да би се до овога дошло прво је потребно променити интерно стање свих веза и освежити графички приказ са новим промењеним вредностима, а тек у наредној итерацији одради израчунавање у тим компонентама. Услов за напуштање петље је случај када симулатор достигне временски лимит у коме је потребно да се симулација извршава.

Описани алгоритам се може додатно убрзати уколико се узме у разматрање чињеница да већина компонената које се користе у области архитектуре и организације рачунара има особину да јој се излази не мењају уколико јој се улази не мењају. Ово значи да се у петљи обрађују само оне компоненте на чијим улазима је дошло до промена у некој претходној итерацији или које морају да буду евалуиране у свакој итерацији. На исти начин се могу обрађивати и компоненте које воде рачуна о протеклом времену и кашњењу кроз компонената. Овако модификован алгоритам је приказан на слици 9.

```
algorithm TimeDrivenAlgorithmWithDelays(p: list of components);
initialize(p);
preCalculate(p);
executionTime := startTime;
while (executionTime < endTime) do
      foreach component in p do
            if component should be evaluated do
                  call component execute;
      end_for;
      foreach executed component in p do
            save component new state;
            distribute component state to connected components;
            schedule connected components for iteration;
      end_for;
      executionTime := executionTime + interval;
end_while;
```

Слика 9: Алгоритам временски вођене симулације заснован на кашњењу кроз компоненте

У сваком симулационом циклусу очитавају се улазни вектори проверавају се компоненте да ли је предвиђена њихова обрада у датом кораку. Уколико се приликом обраде догоди да дође до промене излазних сигнала онда се ажурира листа компонената која треба да буде обрађена као и тренуци када треба да обраде буду обављене. Овај алгоритам се може применити на компоненте које имају коначно кашњење.

Описани алгоритам се у области архитектуре и организације рачунара често примењује у мало измењеним облику који приликом обаде компонената захтева разликовање комбинационих и секвенцијалних мрежа. Тај модификовани алгоритам прво обавља обраду свих комбинационих мрежа сходно датом алгоритму, па се тек на крају ради обрада секвенцијалних мрежа. Обрада комбинационих мрежа се обавља тако што се пре почетка симулације направи тополошки сортирано графа комбинационих мрежа и одреди се редослед по коме ће се обрада обављати. Овако измењен алгоритам може да има доста боље перформансе, али захтева познавање графа мреже. Недостатак овог измењеног алгоритма је то што не може да се примењује приликом обраде комбинационих мрежа са повратном спрегом које могу да се понашају као секвенцијалне мреже.

3.1.2.2.2 Догађајима вођена симулација

Други тип симулатора су симулатори који се усредсређује на догађаје који су битни некој компоненти да би могла да израчуна своје стање, уместо саме компоненте које је потребно обрађивати. Овај тип симулатора обрађује догађаје у тачно одређеном временском тренутку. Основна особина ових симулатора је да се интерно стање компоненте модификује само у случајевима да се десио неки догађај који проузрокује ту промену. Алгоритам се састоји у томе да се сви догађаји убацују у листу сортирану према времену када је потребно извршити обраду тих догађаја, а онда се обрада обавља над првим елементом те листе. Када обави обраду једног догађаја компоненте креира листу нових догађаја која садржи којим је све компонентама упућен тај нови догађај и када је потребно извршити тај нови догађај. Дата листе се убацује у листу сортираних догађаја које је потребно обрадити. Поступак симулације се завршава оног тренутка када листа постане празна или када се достигне тражено симулационо време. Основе догађајима вођеног алгоритма су представљене на слици 10.

```
algorithm EventDrivenAlgorithm (p: list of components);
initialize(p);
preCalculate(p);
executionTime := startTime;
while (eventsExist and executionTime < endTime) do
      event := getEventFromList();
      call event.component execute;
      foreach component new created event do
            putEventToList(event);
      end_for;
      executionTime := event.executionTime;
end_while;
```

Слика 10: Алгоритам догађајима вођене симулације

Овај алгоритам обухвата две фазе. У првој, припремној фази се ради постављање почетних параметара симулације и генерисање почетних догађаја, док се у другој итеративној фази на основу догађаја обавља израчунавања новог стања компоненте као и креирање нових догађаја. Догађаји се узимају из листе у којој се налазе догађају сортирани према времену доспећа догађаја. Време доспећа догађаја представља временски тренутак када је потребно обрадити тај догађај. Узима се први елемент из сортиране листе и прелази се на његову обраду. Тренутно симулационо време се поставља на основу времена доспелог догађаја. Приликом обраде догађаја могуће се створити већи број нових догађаја које је потребно да неке компоненте обраде. Све ове догађаје је потребно креирати и поставити да их требају обрадити компоненте са којима је посматрана компонента у директној вези. Директна веза између компонената се може простирати кроз више нивоа хијерархије тако да је све те компоненте потребно обавестити о догађају. Креирани догађају се убацују у већ сортирану листу догађаја које треба обрадити. Догађај садржи информације о томе која је нова вредност податка, када се појавио догађај, када треба обрадити догађај, порт и компоненту којој је догађај намењен. Након тога се обрађени догађај премешта у листу одрађених догађаја и прелази се на нови циклус. Симулациона петља се извршава у сваком кораку итерације и из ње се може изаћи само у случају да више не постоје скуп догађаја које треба обрађивати или је истекло време симулације.

Догађајима вођена симулација је развијена са циљем да се умањи временска сложеност временски вођених симулација код који је могуће да обави више циклуса симулације а да све то није било потребно. Догађајима вођена

симулација да би уштедела на времену потребном за обављање поступак симулације прелази са фиксног времена провере сваке компоненте на евалуацију стања компонената само у случајевима да је дошло до неких промена на улазу. Само у тим случајевима се прелази на израчунавање новог стања у које треба прећи компонента, али и израчунавање временског тренутка када ће промена постати уочљива на излазима. Овај тип симулатора омогућава да се постигну боље перформансе избегавање непотребне обраде у сваком кораку симулације. Недостатак овог типа симулатора лежу у томе да је у сваком кораку потребно сортирати листу догађаја, што у случају да већина компонената захтева обраду у сваком тренутку може да има лошије перформансе у односу на временом вођене симулације.

3.1.2.3 Алгоритми паралелних симулација

Развој дигиталних рачунарских технологија и система, посебно у области архитектуре и организације рачунара, захтева непрестано прилагођавање и унапређивање могућности које пружају симулатори. Са повећањем сложености система чије се понашање симулира повећавају се и захтеви према ресурсима који су потребни да би се симулација обавила. Ови ресурси, посебно по питању времена потребног да се обави симулација, се могу пронаћи у паралелизацији алгоритама и паралелног коришћења већег броја ресурса.

Један од начина да се време симулације скрати је да се обрада појединих компонената дистрибуира на више процесорских јединица и на тај начин повећа укупна процесорска снага система на коме се врши симулирање. Да би се остварио овакав приступ модел који се симулира се дели на коначан броји дисјунктних делова. Тако формирани скуп дисјунктних делова се распоређује на више чворова који ће даље обављати њихову обраду. Постоји више начина да се искористи могућност паралелизације, а први начин је пројектовање наменских хардверских система који ће хардверски подржати рад симулација. Овај приступ има лоших особина, које се огледају у цени компоненти, као и времена потребног за развој ових симулатора. Други приступ решењу проблема је дистрибуција на више процесора опште намене.

Секције које следе имају за циљ да појасне методе и појмове везане за паралелно извршавање симулације. У њима ће бити представљена два основна

алгоритма симулације прилагођена за рад са већем бројем токова контроле. Поступак дистрибуције компонената као и синхронизација између компонената у дистрибуираном окружењу ће бити представљени у глави која се односи на Симулациони ниво симулатора.

3.1.2.3.1 Паралелна временом вођена симулација

Временом вођена симулација има веома једноставан алгоритам симулације али је временска сложеност датог алгоритма веће него код догађајима вођене симулације. Пошто временом вођена симулације не захтева никакву централизовану структуру података у којој би се чувале информације о стању симулације могуће је веома једноставно паралелизовати извршавање датог алгоритма. Паралелизацијом би се редуковала временска сложеност алгоритма. Основана идеја код синхронизације паралелног извршавања временски вођених симулације лежи у томе да су компоненте чврсто спрегнуте у погледу времена извршавања. Код овог приступа свим компонентама се прослеђује јединствена синхронизациона порука упућена делу за извршавање симулације као основни синхронизациони параметар. Симулациони алгоритам који описује паралелне временски вођене симулације је приказан на слици 11.

```
algorithm TimeDrivenParallelAlgorithm(p : list of components
          i : node id);
initialize(p, i);
preCalculate(p, i);
executionTime := startTime;
while (executionTime < endTime) do
      barrier synchronization
      foreach component in p do
            call component execute;
      end_for;
      barrier synchronization
      foreach component in p do
            save component new state;
            distribute component state to connected components;
      end_for;
      executionTime := executionTime + interval;
end_while;
```

Слика 11: Алгоритам паралелне временски вођене симулације

Симулациони алгоритам који описује паралелне временски вођене симулације се састоји из две фазе. У првој припремној фази се обављају активности везане за иницијализацију симулације. У поступку иницијализације

компоненте се распоређују на више процесора како би се њиховом паралелном обрадом добило на перформансама читавог симулатора. Након припремне фазе прелази се на петљу у којој се извршава симулација. На свим процесорским јединицама у паралели се ради обрада компоненти, при обради се ради евалуације интерних стања компоненте и израчунавање нових излаза из система. Када се заврши обрада свих компонената не некој процесорској јединици ради се синхронизација са свим осталим паралелно покренутим алгоритмима. Скуп израчунатих излаза се тада обједињује и врше се припреме за следећи корак обраде. Ради се прослеђивање нових улазних параметара свим рачунарима који врше обраду. Након прослеђивања параметара поново се врши синхронизација на баријери између паралелних симулатора. Након тога се поновно испитује које компоненте треба обрадити јер су јој се променили параметри. Као што се са дијаграма може закључити синхронизација процеса се постиже након сваког корака извршавања алгоритма. Услов за напуштање петље је случај када симулатор достигне временски лимит у коме је потребно да се симулација извршава.

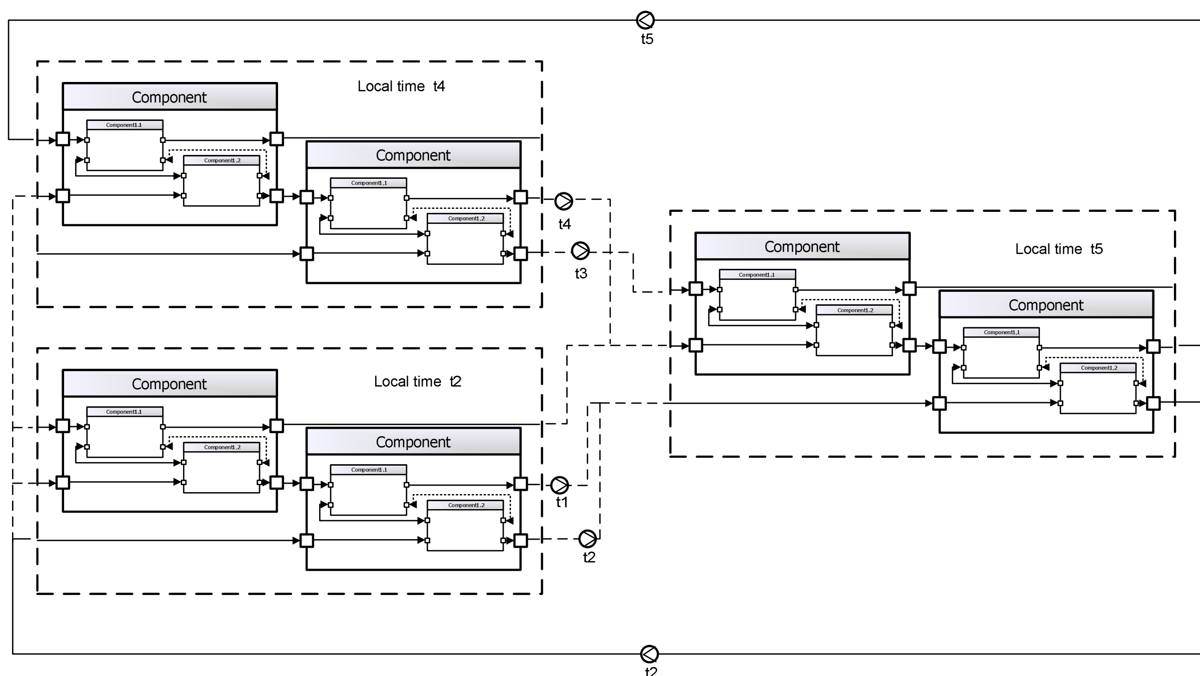
Паралелна временом вођена симулација се може доста добро користити код симулације дигиталних система код којих се у сваком кораку врши промена великог броја улазних параметара. Да би се проценила добит овог симулатора потребно је извршити равномерну поделу компонената на већи број рачунара. Компонента за чији је време обраде потребно највише времена одређује време које је потребно за се обави циклус извршавања алгоритма. Ово може да представља проблем код процене времена извршавања за компоненте које су дефинисане на различитим нивоима апстракције јер време извршавања обраде сложених компоненте може вишеструко да надмаши времена потребна за извршавање обраде једноставнијих компоненти.

3.1.2.3.2 Паралелна догађајима вођена симулација

Као алтернативни приступ паралелним временом вођеним симулацијама јавља се концепт се заснива на паралелној дискретној догађајима вођеној симулацији. Код временом вођених симулација сви паралелни процеси имају исто логичко време, што се постиже тиме што се сви паралелни процеси у свакој итерацији синхронизују на баријери. Ово је остварено тиме што код свих

паралелних процеса постоји униформан проток времена. Када се ради о догађајима вођеним симулацијама код њих не постоји униформан проток времена и постоји проблем синхронизације између паралелних процеса у којима се обавља симулација.

Код паралелних догађајима вођених симулација сваки паралелни процес поседује своје независно вођење рачуна о протеклом логичком времену. Сваки од проса догађаје обрађује у растућем редоследу додељених временској тренутака. Код овог концепта, за разлику од паралелних временски вођених симулација, не постоји концепт заједничког времена, већ само поступка за локално мерење протеклог логичког времена, као што не постоји ни концепт заједничког стање симулатора јер се све компоненте обрађују независно. Да би се очувао глобални редослед извршавања све додељени временски тренуци се чувају на сваком паралелном процесу у одвојеним редовима, а сваки процес прво прихвата догађај са најмањим временом доспећа да би њега прво извршио, па се онда ради провера да ли је дозвољено извршити дати догађај. Приказ рада једног оваквог симулатора је дат на слици 12.



Слика 12: Паралелна догађајима вођена симулација

Приликом рада са паралелним симулаторима дискретних догађаја потребно је обезбедити асинхроно паралелно извршавање где је дозвољено да

сваки паралелни процес има различито логичко време. Основна карактеристика је могућност потпуног одсуства централног сервера или централне контроле заједничког протока времена, мада постојање датог сервера не мора да буде искључено. Непостојање глобалног времена производи проблем синхронизације свих паралелних процеса и обраду догађаја у њиховом хронолошком претку. Оно што симулатор треба да обезбеди је паралелна симулација мора да да идентичан резултат као и симулатор са секвенцијалном обрадом.

Када се ради о секвенцијалној обради било који догађај x1 са тренутком почетка извршавања t1 ће се извршити пре догађаја x2 са тренутком почетка извршавања t2 уколико важи t1<t2. Ово се код симулатора са секвенцијалном обрадом постиже тако што се распоређивање који ће догађај прво обрађивати врши на основу тренутака извршавања свих преосталих догађаја. Уколико постоје два или више догађаја који имају исти тренутак започињања мора постојати јединствен детерминистички редослед заснован на графу симулиране мреже по коме ће се обрада вршити. Да би за паралелну симулацију могло да се каже да се понаша исправно излази које симулатор генерише морају да буду у складу са наведеним принципом.

Када се врши обрада логичких процеса који се обављају у паралели обично не постоји никаква информација о глобалним карактеристикама симулације у том тренутку, већ само парцијални подаци. Проблем који се јави се односи на случај када процес p1 обрађује догађај x1 са тренутком почетка извршавања t1, након кога му приспе порука од процеса p2 о догађају x2 са тренутком почетка извршавања t2 при чему важи да је t2<t1. Да би се наведени проблем решио паралелни процес мора да има измењен основни симулациони алгоритам који треба да обави извесне припреме радње и изврши одређен скуп акција. Постоје два основна принципа у решавању овог проблема. Један је конзервативни приступ, а други је оптимистички приступ.

Конзервативни приступ решавању проблема синхронизације захтева да се неки догађај обрађује само у случају да постоји потпуна извесност да ни један други догађај неће доћи на ред пре датог догађаја. Да би се обезбедио конзервативни приступ потребно је статички успоставити везе између процеса који међусобно комуницирају. Процеси који међусобно комуницирају треба да

обезбеде да догађаји које размењују буду у растућем редоследу временских тренутака када је потребно да се дати догађаји обраде. Свака веза између процеса који комуницирају такође треба да води рачуна о протеклом времену, тако да свака веза води рачуна о логичком времену и поседује информацију о томе који је тренутак доспећа следећа поруке у реду или временски тренутак последње примљене поруке ако ред је празан. На слици 13 је дат алгоритам извршавања заснован на конзервативном приступу синхронизације симулатора дискретних догађаја.

```
algorithm EventDrivenConservativeAlgorithm (p: list of components);
initialize(p);
preCalculate(p);
executionTime := startTime;
while (eventsExist and executionTime < endTime) do
      do
            x := getNextSafeMoment();
      while ( x < getFirstExecutionTime());
      event := getEventFromList();
      call event.component execute;
      foreach component new created event do
            if (event.component is in group(p, i)) then
                  putEventToList(event);
            else
                  sendEventToScheduler(event);
            end_if;
      end_for;
      executionTime := event.executionTime;
end_while;
```

Слика 13: Алгоритам конзервативне паралелне догађајима вођене симулације

Оптимистички приступ решавању проблема синхронизације се заснива на томе да се сви догађаји обрађују у редоследу према својим временима доспећа. Уколико пристигне закаснела порука чија је време доспећа ниже од тренутног логичког времена у посматраном процесу, симулација на датом процесу се враћа на тренутак који може да гарантује исправно извршавање свих догађаја, било тог закаснелог било свих они који су требали да се догоде после њега, иако је тада већ била завршена њихова обрада. Да би се ово остварило потребно је чувати информације о текућем стању да би се омогућио повратак на пређашње стање, тј извршен rollback трансакције на временски тренутак нове поруке. Информације о променама се могу чувати инкрементално или се периодично формирати сигурно место за повратак (restor point). Чување контекста се може остварити на тај начин

што се обезбеди да се за сваки атрибут поред своје тренутне вредности мора сачувати и информација о историји, тј о променама које су се догодиле и њиховим временским тренуцима. Ова информација може да буде инкрементално или периодично чувана. На овај начин имамо комплетну историју помоћу које можем враћати симулацију на произвољан тренутак због потребе синхронизације. Да би се поступак враћања симулације убрзао постоји известан број оптимизационих алгоритама који ограничавају број стања/корака на које се симулације може вратити уназад да би систем прешао у конзистентно стање. Поред овога је неопходно опозвати све поруке које је дати процес послао другим процесима у интервалу који се поништава. Ово може да произведе каскадно враћање на претходно стање код других процеса. На слици 14 је дат алгоритам извршавања заснован на оптимистичком приступу синхронизације симулатора дискретних догађаја.

```
algorithm EventDrivenOptimisticAlgorithm (p: list of components);
initialize(p);
preCalculate(p);
executionTime := startTime;
while (eventsExist and executionTime < endTime) do
      event := getEventFromList();
      if (executionTime <= event.executionTime and event <> cancel) then
            call event.component execute;
            foreach component new created event do
                  if (event.component is in group(p, i)) then
                        putEventToList(event);
                  else
                        sendEventToScheduler(event);
                  end_if;
            end_for;
            executionTime := event.executionTime;
      else
            undoAllActions(event.executionTime);
            sendCancleForRestartedPeriod(i, event.executionTime);
            removeUnconfirmedEvents(executionTime, event.executionTime);
            putEventToTheExecutedList(event);
      end_if;
end_while;
```

Слика 14: Алгоритам конзервативне паралелне догађајима вођене симулације

Да би обезбедили исправно функционисање симулације сваки од наведених алгоритама мора да утроши додатно време и ресурсе. Овај утрошак времена може да буде такав да дође до делимичног губитка ефекта које доноси паралелног извршавања. Сваки од алгоритама даје оптималне резултате у случају

да је посао који сваки паралелни процес обавља приближно једнак. Наведени алгоритми имају својих добрих особина али и ограничења, о којима се мора водити рачуна приликом одабира како се не би догодило да са повећањем броја доступних рачунара не добије никакав напредак у перформансама у односу на секвенцијалне алгоритме симулације. Да би се дошло до позитивних резултата мора се водити рачуна о следећим карактеристикама: дељењу модела на мање целине, оптимизацији протока, о нивоу детаља на коме су компоненте приказане, о комуникационим протоколима, о расположивим ресурсима, потреби да се симулациони модел интерактивно приказује кориснику. Код наведених алгоритама уколико симулатор користи презентациони слој, који се налази на једном рачунару онда је потребно посебно размотрити пријем корисничких асинхроних интерактивног сигнала.

3.1.2.4 Асинхрона интеракција

Симулатори у области архитектуре и организације рачунара спадају у симулаторе са великим степеном интеракције са корисником. Приликом ове интеракције корисник може да креира догађаје који у симулатору треба да произведу пријем спољашњих сигнала. Приликом рада са овим асинхроним сигналима потребно их је некако увести у симулацију.

Временски тренуци када асинхрони сигнали могу да приспеју у симулатор се најбоље могу моделовати користећи континуално време. Модел континуланог времена је погодан за рад само у система са малим бројем компонената. Да би се избегло ово ограничење асинхроне сигнале је потребно увести у симулатор у дискретним временским тренуцима. Приликом одређивања тренутака у којима треба обавити пријем асинхроних сигнала треба водити рачуна о удаљености ових тренутака. Уколико су тренуци сувише блиски добија се могућност за прецизнијим описом сигнала, али то може додатно да успори поступак симулације, и обрнуто.

Да би се обезбедио интерактивни рад са корисником осим коришћења компоненте која омогућава пријем асинхроних сигнала ни код временом ни код догађајима вођене симулације поступак симулације не захтевају додатне модификације алгоритма. Оно што је потребно да дата компоненте обезбеди је да корисник има привид униформног протока времена. Уколико се користи

временом вођена симулација онда се посматра скуп сукцесивних временских интервала чије време обраде треба да буде равномерно како би остварила могућност да корисник има привидно равномерно протицање времена. На овај начин се постиже да се дигитални асинхрони сигнали уведу у симулацију.

Уколико симулатор користи догађајима вођена симулација онда је поступак извршавања симулације непромењен у односу на оригинални алгоритам. Ово је могуће остварити зато што се код ове групе симулатора евалуација обавља само у случајевима да је дошло до неких промена на улазу без обзира како се та примена догодила. Овај вид симулације може да има проблема са визуализацијом код корисника зато што овај тип симулатора нема униформан проток времена и тешко је предвидети на који се следећи временски тренутак прелази. Начин да се ово избегне би захтева постојање компоненте која би периодично генерисала догађаје како би корисник имао равномеран проток времена.

3.1.3 Пројектовање симулационог дела симулатора

Симулациони слој симулатора представља део симулатора који води рачуна о формирању и распоређивању свих компонента по доступним рачунарима и протоколима за њихову синхронизацију. Алгоритми распоређивања и синхронизације се развијају независно од области у којој се могу примењивати, али њихове перформансе доста зависе од конкретне области. Приликом пројектовања симулационог слоја симулатора треба дати одговор на питања где симулатор треба да обавља симулацију. Одговором на ово питање дефинише се домен проблем. Домен проблема се обноси на то који алгоритми распоређивања и синхронизације треба да буду реализовани унутар симулатор. На основу анализе симулатора уочено је да симулаторе потребно развијати тако да остали слојеви симулатора не буду свесни постојања дистрибуираног извршавања. Поступком раздвајања симулатора на слојеве постиже се да све компоненте могу међусобно да комуницирају као да се налазе повезане на једном рачунару. Овакав распоред слојева је од великог значаја за пројектовање симулатора са дистрибуираном обрадом јер омогућава паралелно извршавање уз поделу логичког модела на већи број под модела. Приликом поделе модела треба водити рачуна да укупно време потребно да се симулација обави може значајно да варира од распореда компонента и комуникационог времена које слојеви троше на међусобну

синхронизацију.

Основни задатак симулационог слоја симулатора је обезбеди равномерно распоређивање компонента по доступним рачунарима и синхронизацију извршних слојева симулатора. Да би се покренула симулација неопходно је распоредити обраду и компоненте по доступним рачунарима, иницијализовати све слојеве симулатора на тим рачунарима, обезбедити синхронизацију између компонената и слојева као и синхронизацију са корисничким интерфејсом. У овој секцији ће бити представљени могући поступци за расподелу компонената по рачунарима, расподелу података, као и поступцима оптимизације извршења тако дистрибуираног модела.

3.1.3.1 Расподела компонената

Поред одабира алгоритма који ће се користити приликом паралелне обраде у дистрибуираном окружењу потребно је извршити и поделу компонената на делове како би се извршавали у паралели. Алгоритам за оптималну расподелу компонената на доступне ресурсе како би се оптимизовало време извршавања захтева превелику рачунарску сложеност која може да превазиђе само време потребно да се симулација обави. Ови алгоритми имају NP сложеност што значи да време потребно за његово извршавање није полиномијално и да доста зависи од броја компонената које је потребно обрадити. Коришћење оваквих алгоритама може да доведе до проблема јер у дистрибуираном окружењу постоји велики број међузависних параметара о којима треба водити рачуна. Да би се овај проблем ублажио развијено је више алгоритама који на приближан начин решавају проблем расподеле компонената.

- Случајна деоба као полазну основу узима случајан распоред рачунара на који ће бити смештена појединa компонента. Што се времена за расподелу компонената по рачунарима тиче овај алгоритам има линеарну сложеност, и спада у групу најједноставнијих алгоритама. Овај алгоритам се доста примењује јер има веома једноставну имплементацију, а даје оптималне перформансе у случају малог броја компоненти које имају велико и приближно исто време обраде.
- Принцип кружног распоређивања (round robin) је облик равномерног распоређивања компоненти по рачунарима. Код овог принципа компонента се распоређује на први слободни рачунар и тако укруг док се не распореде све

компоненте. Овај принцип се једноставно примењује и у просеку има боље перформансе од случајне расподеле. Принцип даје оптималне перформансе у случају компонента са великим бројем међусобних веза које имају приближно исто време обраде. Недостатак овог принципа је у томе што претпоставља да време обраде сваке компоненте траје приближно исто, и то да је подједнако вероватно да сваке две компоненте међусобно комуницирају.

- Подела на регионе је приступ који тежи да се компоненте које су у блиској логичкој вези и које формирају сложеније компоненте распоређују на исти рачунар ради даље обраде. Овај поступа захтева итеративну примену тако што се након формирања региона састављених од компонената најнижег нивоа прелазе на груписање сложених компонената у веће регионе према броју јединица које их формирају. Овај начин треба да обезбеди подједнаку упосленост свих рачунара у случају да је време обраде компонената најнижег соја приближно једнако. Ово је први алгоритам који се заснива на разматрањима особина графа повезаних компонената. Ни овај алгоритам како ни претходни не узима у разматрање време потребно за обраду појединих компонената, како ни вероватноћу да поједине компоненте комуницирају јер су то динамички критеријуми.
- Принцип равномерног времена обраде је поступак распоређивања компонената код које сваки рачунар добија компоненте тако да је време обраде компонената на рачунарима подједнако. Ово је први приступ који узима у разматрање време обраде појединих компонената узето на основу пробног мерења. Овај приступ даје боље резултате него претходни, а најбоље резултате у ситуацијама када су све компоненте подједнако упослене. Овај алгоритам не узима у разматрање везе између компонената као ни вероватноћу да поједине компоненте међусобно комуницирају.
- Минимизација комуникационих трошкова као принцип полази од претпоставке да је време потребно да се обави комуникација између рачунара, односно компонената на неком рачунару, пресудан параметар за дистрибуирано извршавање. Ово решење има за циљ да обави балансирање између времена извршавања и времена потребно да се обави комуникација процеса унутар симулатора. Ово решење комбинује статичке параметре времена обраде компонената и времена потребног да се комуникација између чворова обави. Овај

приступ има боље резултате у односу на претходне у случајевима када постоји асиметрична комуникација између чворова на коју се троши доста времена. Овај приступ се може комбиновати и са статички одређеном вероватноћом размене порука између компонената. Мана овог приступа је што не узима динамичке параметре у разматрање, али даје набоље резултате од свих остарилих алгоритама расподеле.

3.1.3.2 Расподела и синхронизација података

У раду у дистрибуираном окружењу поред компонената код којих се комплетне информације налазе на једном рачунару постоје и компоненте код којих се подаци налазе размештени на већем броју рачунара. Сваки дистрибуирани систем има барем једну дистрибуирану компоненту која представља читав симулирани систем, а могуће да поседује и већи број у зависности од расподеле компонента по рачунарима. Да би се обезбедио рад једне такве компоненте потребно је обавити поступак синхронизације података о тој компоненти која се чува на већем броју рачунара.

Код симулатора архитектуре и организације рачунара који су разматрани у овом раду постоји потреба за интерактивном комуникацијом са корисником. Ова комуникација се обавља преко једног рачунара где се кориснику исцртавају резултати симулације, и где корисник интерактивно уноси нове параметре симулације. У зависности где се налазе подаци, и да ли се на централном серверу обрада обавља у једној или више токова контроле симулаторе можемо поделити на две класе: прва код које постоји дељена меморија и друга код које свака компонента има своју меморију. Прва класе симулатора захтева конкурентну обраду на већем броју нити а комуникација се обавља помоћу дељених података док се код друге класе комплетна комуникација обавља разменом порука.

У зависности од одабраног облика дељења података разликује се и поступак синхронизације података између рачунара који раде обраду. Подаци се између рачунара који раде обраду и које је потребно синхронизовати могу поделити на више начина. Начини зависе од тога да ли се синхронизација података обавља на крају задатог циклуса или ју је потребно непрекидно успостављати.

- Први приступ поступку синхронизације података се заснива на коришћењу

централног рачунара. Овај рачунар може да буде централни сервер који обавља извесну обраду, али и рачунар на коме се налази презентациони слој симулатора. Код овог приступа комплетан скуп података се налази на централном серверу док радне станице које обрађују податке добијају скуп који је потребно да обраде. Овај скуп не представља комплетну информацију о стању симулатора већ један њен подскуп. Када радна станица заврше обраду добијеног скупа података онда све добијене резултате враћају централном рачунару. Овај начин расподеле података је погодан за временом вођене симулације јер омогућава велики степен паралелизације делова који у неком интервалу не интерагују.

- Други приступ поступку синхронизације података се заснива на коришћењу централног рачунара и већег броја радних станица у активном режиму рада. Приступ се заснива на побољшањима првог приступа у случају да се радне станице јављају централном серверу са захтевима за новим подацима за обраду. На ова начин се радне станице из пасивног положаја пребацују у активнији положај. Овај приступ даје добре резултате у ситуацијама када се на радним станицама обавља обрада која траје приближно исто и када није потребно транспортовати велику количину података између радних станица и централног рачунара. Приликом одабира поступка треба водити рачуна о томе колико износи комуникационо време да постигли оптимални резултати, али и одрадило балансирање протока, јер мрежа може да буде ограничавајући фактор у комуникацији. Овај начин расподеле података као и претходни је погодан за временом вођене симулације. Свака радна станица треба да добије целокупан скуп стања и како би могла итеративно да обавља петљу израчунавање следећег стања за сваки подскуп компонената и обављање синхронизације свих радних станица са централним сервером на крају циклуса.

- Трећи приступ поступку синхронизације података је да се комплетан симулирани систем држи на већем броју рачунара како би први слободан рачунар могао да преузме обраду у неком тренутку. Пошто се код овог приступа комплетна логичка шема симулираног система дистрибуира на више радних станица потребно је користити додатне алгоритме за синхронизацију приступа овим дељеним подацима. Овај приступ оптимално троши време процесора када је у питању обрада, али захтева додатно време за синхронизацију копија података.

Поступак синхронизације може да захтева размену велике количине података између свих радних станица што може да доведе и до губитка на перформансама у поређењу са другим приступима. И овај начин расподеле података је погодан за временом вођене симулације, мада се може примењивати и код догађајима вођених симулација које имају мали степен интеракције између компонената.
- Четврти приступ поступку синхронизације података је подела симулираног система на већи број дисјунктних делова који се онда дистрибуирају на већи број радних станица како би се обавила обрада. Приликом поделе симулираног система потребно је применити неки од алгоритама описаних у претходним секцијама. Код овог приступа једна радна станица све време симулације обавља обраду истог скупа компонената. Овај приступ се смањује комуникационе трошкове у дистрибуираном систему јер смањује количину размењених порука измешу рачунара који интерагују. Овај приступ даје се користити у случајевима када је симулирани систем комплексан и састоји се од великог броја компонената које је временски захтевно обрађивати на једном рачунару. Овај начин расподеле података је погодан за догађајима вођене симулације јер је у основи обраде догађаја дељење на дисјунктне делове и размена порука о догађајима у систему.

3.1.3.3 Оптимизација извршавања

Представљени алгоритми распоређивања података и синхронизације описују рад симулатора код којих се расподела компонената одрађује статички приликом иницијализације симулатора. У случајевима када се не може проценити време обраде овај приступ има извесних недостатака јер не одражава динамичко понашање симулатора током извршавања симулације. Уколико код симулатора постоје велика одступања у времену обраде између појединих делова који се динамички мења потребно је омогућити динамички прерасподелу компонента међу доступним радним станицама. Поступак динамичког распоређивања компонената може да утиче на укупно време обраде симулације а сам резултат симулације мора да остане исти при било ком алгоритму и начину симулације.

Поступак одређивања рачунара на којима се симулација може обавити се заснива на томе да се постигне што већа упосленост доступних радних станица. Поступак се заснива на процени колико ће времена бити потребно да се изврши извесна секвенца на основу података добијених у претходно посматраном

временском интервалу. Један од поступака процене се заснива на изједначавању времена извршавање појединих процеса на појединим радним станицама сразмерно које компоненте колико времена троши на обраду. Према овом принципу се процеси који се најспорије у том тренутку извршавају пребацују на најбржу радну станицу. Други поступак процене се заснива се на прерасподели компонената тако да бој примљених и послатих порука другим радним станица буде приближно једнак. Према овом принципу на исту радну станицу се пребацују компоненте који имају највећу међусобну комуникацију. Поступак има за циљ да динамички врши балансирање извршавање целокупне симулације. Поступак прерасподеле се примењује итеративно све док се симулација не заврши.

Поступак оптимизације извршавања балансирањем расподеле компонената по радним станицама треба да омогући да се симулација што брже извршава. Овај поступак уводи додатне режијске трошкове тако да треба одредити ситуације у којима се описани поступак може примењивати. Поступак балансирања даје најбоље резултате код временски вођеним симулацијама код којих је временски корак довољно велики да се сва кашњења комбинационих модла могу сматрати окончаним и где постоји мали број чворова који обрађују асинхроне улазе. На другој страни резултати балансирања могу да доведу до губитака на перформансама у симулаторима са догађајима вођеним симулацијама у системима који обављају симулацију система који ради у синхроном режиму рада.

3.1.3.4 Расподела обраде

Симулатори који се користе у области архитектуре и организације рачунара обављају симулације система који садрже веће број повезаних компоненти. Компоненте које се користе могу да буду реализоване на више различитих нивоа апстракције и могу се груписати у известан број логичких целина. Поред саме расподеле компонената потребно је водити рачуна о томе да се време обраде тих компонената равномерно расподели по рачунарима. Ово може да буде проблем јер време обраде зависи од вероватноће којом неке поруке пристижу некој компоненти за обраду. У претходним секцијама је представљен опис формирања логичког и извршног слоја симулатора компонената док је у овој секцији посебна пажња посвећена интеграцији симулатора и синхронизације на

симулационом слоју симулатора.

Поступак интеграције треба да обезбеди повезивање слојева једног или више симулатора као и дистрибуцију података унутар дистрибуираног окружења између симулатора који користе исти алгоритам симулације у извршном слоју. Компоненте које се користе у симулаторима најчешће су дате или на функционалном нивоу описа понашање система или на нивоу трансфера између регистара. Због тога је поступак интеграције представљен на примеру интеграције два симулатора, на симулационом нивоу, једног дефинисаног на функционалном нивоу (JPC) и једног дефинисаног на нивоу трансфера између регистара. Симулатор дат на функционалном нивоу описује сложену логичку компоненту која представља симулатора целог система (full system simulator) која користи временски вођену симулацију са кораком који одговара једној инструкцији. Симулатор дат на нивоу трансфера између регистара представља конфигурабилни симулатор развијен на нивоу трансфера између регистара који је обавља временом вођену симулацију на нивоу једног такта. Оба симулатора су развијена користећи исти програмски језик. У наставку ове секције бити приказани кораци које треба обавити приликом пројектовања одговарајућих слојева симулатора како би се обезбедило паралелно извршавање и комуникација користећи симулациони слој.

Предложени поступак се састоји из 5 корака који садрже екстракцију омотача, дефинисање интерфејса, раздвајање протокола, дефинисање независног извршавања и синхронизација зависног извршавања.

Корак екстракција омотача дефинише потребне податке неопходне за функционисање појединих делова система. Овај корак је потребно независно урадити за обе стране које се интегришу. Резултат овог корака је листа потребних података која се у случају интеграције са симулатором архитектуре рачунара (JPC) састоји из скупа података који се дохватају из адресног простора. Овај скуп је локализован на нивоу класе за приступ меморији FlatMemory. Из угла конфигурабилне кеш меморије скуп података се састоји из адресе и редоследа по којима се приступа, затим информације о типу операције и временском тренутку када су тражени подаци неопходни. Овај скуп података је локализован на нивоу приступних класа које имај различит формат од оних које постоје код JPC симулатора.

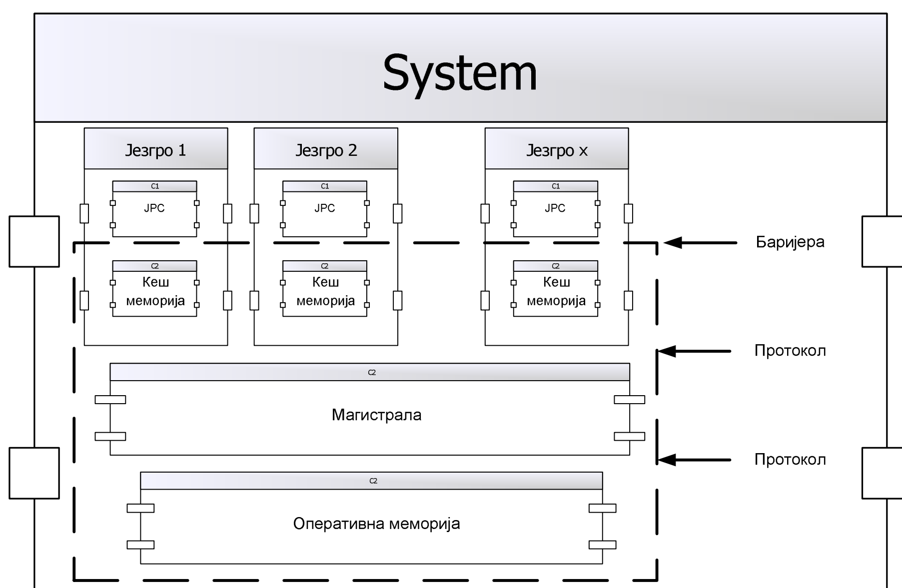
Корак дефинисање интерфејса је потребно урадити и за JPC симулатор и за симулатор кеш меморије на основу података добијених у претходном кораку. Интерфејси представљају скупове метода помоћу којих ова два симулатора размењују податке. На страни JPC симулатора те методе су уписи и читања на нивоу адресибилних јединица. На страни кеш меморије интерфејс се састоји из метода за приступ оперативној меморији и синхронизације са осталим уређајима у систему. Интерфејси су тако развијени да се уместо JPC симулатора може користити и други симулатор који опсује друге архитектуре рачунара.

Корак у коме се врши раздвајање протокола треба да дефинише редослед интеракција и тренутке позивања метода дефинисаних у претходном кораку. При интеграцији JPC симулатора потребно је дефинисати колико података може да се прочита у јединици времена и када ће ти подаци бити доступни. Као најважнији протокол који је подржан је интеграција протокола на магистрали с обзиром да магистрала представља дељени ресурс који користи већи број компонената система. Као посебна компонента која обавља овај корак интеграције уведена је компонента арбитратор која омогућава синхронизацију и арбитрацију између појединих компонената система. Компонента арбитратор је развијена користећи фиксни редослед давања предности компонената према њиховој идентификацији. На овај начин је омогућено да се симулатор користи не само у системима који поседују једно језгро већ и у системима састављених од више језгара.

Корак независног извршавања има за циљ да дефинише независно извршавање појединих делова симулатора. У једној нити симулатора се обавља симулација компонената чији је потпис дат на нивоу трансфера између регистара, док је за сваку нову инстанцу JPC симулатора потребно додати нову нит. На овај начин је омогућено ефикасно искоришћење ресурса рачунарског система на коме се симулација обавља. Увођење посебне нити за свако појединачно језгро обезбедило је потребан ниво изолације, чиме је избегнута потреба да се JPC симулатор модификује у већој мери. Овај исти поступак раздвајања се може применити и на интеграцију са другим симулаторима, јер не захтева даљу модификацију конфигурабилног симулатора кеш меморије.

Нити креиране у претходном кораку су посматране као независни делови, али оне нису међусобно независне и потребно их је посебно синхронизовати.

Синхронизација између ових нити је обављена увођењем баријере између симулатора кеш меморије и инстанци JPC симулатора. Синхронизација нити у Јави је обављена увођењем посебних бафера између сваке нити. Повратна информација коју конфигурабилни симулатор враћа JPC симулатору се такође баферише. Као један од проблема који овде могу да се јаве се је и ситуација када је потребно одступити од секвенцијалног извршења и поновити извршење појединих инструкција. Ово је посебно битно код система који треба да моделују понашање посебних типова меморије која хардверски подржава трансакције.



Слика 15: Поступак интеграције

На слици 15 је дата шема поступка интеграције. Резултати корака екстракције и дефинисања интерфејса означени су како поједини блокови оивичени пуном линијом. То су блокови JPC, кеш меморија, магистрала и оперативна меморија. Резултат корака раздвајања протокола јесте скуп протокола који окружују магистралу. Резултат корака симулације у више нити приказани су као блокови оивичени испрекиданом линијом. И резултат корака синхронизације јесу баријеру коју постоји између JPC блокова и блокова кеш меморије.

Поред радних станица које раде обраду симулатори имају и централни рачунар преко кога се обавља интерактивно презентовање резултата симулације. Централни рачунар преко кога се обавља комуникација може да има улогу у поступку синхронизације јер корисник на својој радној површини не сме да добије резултате који су добијени спекулативним извршавањем. Овакав облик

синхронизације ствара синхронизацију на баријери, при чему је баријера једна од интеракција са корисником. Приликом одабира алгоритма извршавања треба водити рачуна и о презентационом слоју симулатора јер и он може да захтева доста трансфера и размене порука у систему, пошто презентациони слој служи за интеракцију корисника са симулатором преко кога корисник може да задаје команде које утичу на ток симулације.

3.1.4 Пројектовање физичког дела симулатора

Слој физике симулатора представља везу логичких компонената симулатора у области архитектуре и организације рачунара и физике компонената. Приликом пројектовања слоја физике симулатора треба дати одговор на питања да ли је понашање симулатора у складу са реалним физичким законима. Одговором на ово питање дефинише се домен проблем. Домен проблема се обноси на то које физичке компоненте треба да чине симулатор, односно помоћу којих физичких компонената је могуће описати логичке компоненте симулатора. Овај слој представља још један ниво у хијерархији апстракција компонената и веза које се користе приликом симулација.

Посматрани симулатори у овом раду су се у малом обиму бавили слојем физике компонената. Ови симулатори са веома великим нивоом детаља описују симулиране системе. Овај ниво детаља по свом обиму обично превазилази домете курсева намењених области архитектуре и организације рачуната. Ниво детаља који се користе код ових симулатора такав да је време потребно за обраду ових симулације неупоредиво дуже у односу на симулаторе који обраду раде на логичком нивоу. Слој физике симулатора води рачуна о физичким карактеристикама ресурса на који би се требало да се преслика логички део симулатора. Секције које следе се баве моделовањем физичких ресурса, компонената и веза, моделовањем агрегатних параметара зависних од већег броја компонената (времена обраде и дисипације).

3.1.4.1 Компоненте слоја физике

У овој секцији се разматрају компоненте слоја физике које се користе за детаљан опис логичких компонената које се појављују у области Архитектура и организација рачунара. Опис компонената је најчешће дат у облику скупа линеарних једначина или чешће диференцијалних једначина. Да би се понашање

компонената описало потребно је увести скуп релација којима се повезују основне величине које се физичке величине које се посматрају. Стање система и прелаз под утицајем улазних побуда се врло често одређује решавањем диференцијалних једначина. Решавање једначина се врло често обавља итеративном нумеричким решавањем уз увођење потребног нивоа апроксимације. У неким случајевима није могуће спровести овај поступак са великим нивоом детаља због велике временске сложеност потребне приликом анализе сложених система који се користе у области архитектуре и организације рачунара. Компоненте слоја физике могу припадати неком од следећих нивоа апстракције: Ниво идеалних прекидача, Ниво прекидача, и Ниво кола. Ови нивои као и компоненте расположиве код ових типова симулатора су приказани у Табели 9.

Табела 9: Нивои апстракције компонената слоја физике

Нивои компонената	Расположиве компоненте
Ниво идеалних прекидача	Идеални транзистори
Ниво прекидача	Транзистори, отпорници, кондензатори
Ниво кола	Отпорници, кондензатори, напон, струја

Компоненте које се корите на нивоу идеалних прекидача као основну компоненту користе идеалне транзисторе. Приликом рада са идеалним транзисторима саме вредности напона и струје нису од интереса приликом рачунања, већ је само битно присуство одређеног напона односно струје на појединим улазима прекидача. По овој својој карактеристици компоненте овог нивоа веома личе на логичких компоненти и представљају најапстрактније компоненте овог ниво. Време потребно за обраду компонената овог нивоа се не разликује пуно у односу на време обраде компонената слоја логичких компонената. Ове компоненте као један од својих параметара имају време одзива, односно параметар који специфицира колико након промене неког од улазних параметара долази до промене на излазу.

Као усложњење компонената састављених од идеалних прекидача јављају се компоненте које описују рад реалних прекидача. Компоненте овог нивоа које описују понашање дигиталних система обухватају транзисторе, отпорнике и

кондензаторе. Понашање ових компонената утиче на целокупно понашање дигиталне компоненте на тај начин што се одређује тренутак промени излазних сигнала. Код овог типа компонената се користе отпорници и кондензатори, али се они разматрају статички да би се одредило време кашњења/пропагације сигнала кроз дигиталну компоненту. Статичко разматрање подразумева да се понашање компоненте одређује само једном без улажења у поступак решавање диференцијалних једначина током читавог трајања симулације. На овај начин се добијају реалнији излази система, као и кашњење. Ови параметри се прослеђују свим слојевима у хијерархији изнад како би се могли користити приликом симулације.

Као најсложенији ниво јављају се компоненте задате на нивоу електричних кола. Компоненте овог нивоа поред транзистора, отпорника и кондензатора, време посматрају континуално и заснивају се на непрестаном решавању диференцијалних једначина како би се одредили излази и понашање система. Излази и понашање система састављених од ових компонената су описани континуалним величинама напона и струје. Приликом поступка симулације компоненте овог нивоа захтевају најдуже време обраде зато што обављају најсложенија израчунавања. Као резултат ових израчунавања добија се реалан увид у то да ли је очекивано понашање дигиталних система адекватно описано. Поред увида добијају се и параметри који се могу користити као синтетични на следећим нивоима. Ово параметри обухватају време пропагације сигнала, времена потребна за постављање сигнала, дисипацију система у функцији улаза, могућност повезивања излаза једне компоненте на веће број улаза других компонената без губитка информација.

3.1.4.2 Везе слоја физике

У овој секцији се разматрају везе између компонената слоја физике симулатора. Као и везе код логичких компонената, везе слоја физике треба да обезбеде размену информација и података између компонента. Подаци које ове компоненте транспортују се разликују од слоја до слоја, али се разликују и по тренуцима када је потребно компоненте испоручити, као и гранулатности посматраног времена. Као ни логичке везе слоја физике служе искључиво за транспорт и не обављају никакву обраду. Код ових компонента може да постоји

већи степен повезаности тако да везе могу да постоје између великог броја компонената. На основу поделе на нивое сложености компонената, на исти начин се деле и везе између компонената слоја физике. Нивои веза, као и расположиве везе унутар нивоа су приказани у Табели 10.

Табела 10: Нивои апстракције веза слоја физике

Нивои веза	Расположиве везе
Ниво идеалних прекидача	Размена дигиталних сигнала
Ниво прекидача	Дигитални сигнали и кашњење
Ниво кола	Напон, струја континуалне вредности

Везе које повезују компоненте нивоа идеалних прекидача треба да обезбеде комуникацију између идеалних транзистора. Ове вредности најчешће спадају у скуп {0, 1}, {0, 1, X}, као и {0, 1, X, Z}, иако могу да буду и реалне вредности. Везе овог нивоа се формирају између компоненти и полазе од излаза једне до улаза друге компоненте. Оно што је потребно обезбедити код овог слоја компонената је поред вредност имати информацију о томе када је нека вредност постављена. Овај тип компонената и веза се користи за прорачуне дигиталних компонената код којих постоји кашњење, временска разлика, између појаве улазне побуде и креирања одговора на дату побуду.

Преласком на сваки нижи ниво апстракције симулатора повећава се количина података које је потребно да компоненте међусобно размењују, али и подаци постају све сложенији. На нивоу реалних прекидача вредност које компоненте размењују могу да буду у истом скупу као и код слоја идеалних прекидача, али са том разликом што се овде кашњење може много прецизније одредити. Приликом описа система састављеног од компонената овог нивоа везе представљају помоћ приликом израчунавања кашњења сигнала решавање једначина за неки скуп улазних параметара.

На ниво логичких кола разматрају се напон и струја у одређеном временском тренутку. Величине напона и струје се посматрају као реалне вредност да би што прецизније омогућиле решавање једначина које описују понашање система и релацију између поменутих величина. Ова врста симулатора

захтева најинтензивнију размену података по обрађеном временском интервалу, али даје најпрецизније резултате како би омогућило прецизну симулацију на осталим слојевима.

3.1.4.3 Агрегатни резултати

Поступак симулације дигиталних система поред израчунавања непосредних величина које прате симулацију треба да се бави израчунавање агрегатних резултата који могу да потичу од већег броја компонената. Да би се омогућио овај поступак потребно је обавити креирање посебних компонената/ресурса и извршити пресликавање одређеног скупа компоненти на ту компоненту. Овај поступак омогућава трансформацију графа веза између компонената у граф расположивих ресурса. Овај поступак најчешће треба да омогући описивање просторних зависности већег броја компонената, било логичког слоја било слоја физике, које утичу на један агрегатни резултат. Агрегатни резултати се могу поделити на статичке и динамичке у зависности да ли их је потребно рачунати само једном, у време припреме за извршавање симулације, или их евалуирати током цело поступка симулације.

Типичан пример статичких агрегатних резултата симулације који се израчунавају и припремају само једном је просторна расподела компонената једног слоја. Ова расподела очекује познавање описа компонената које се симулирају, али и познавање графа пресликавања и коришћења ресурса. Из овога се може закључити да слој физике треба да има вишеструке критеријуме који треба да омогуће моделовање дигиталних система и њихову симулацију са потребним нивоом детаља. Описи и компоненте физичког нивоа се поред понашања физике компонената односе и на просторни распоред компонената које одговарају компонентама на логичком нивоу. Описа просторног распореда компонената представља статички параметар, јер након пројектовања симулираног система не долази до померања описиваних компонената.

Поред статичког агрегатног резултата постоје и динамички резултати који се срачунавају током целог израчунавања симулације. Типичан пример овог типа резултата је дисипација. Дисипација је најчешће последица пуњења паразитних капацитивности услед промене напона, односно промене улазних сигнала. Пример дисипације је доста комплексан јер да би се добио овај агрегатни резултат

постоји потреба да формирањем просторног распореда који такође може да буде агрегатни резултат. Да би се одредила дисипација потребно је током симулације патити број промена појединих улаза, и на основу карактеристика посматране компоненте се израчунава дисипација, али на основу просторног распореда, та дисипација се распоређује по одговарајућем чипу.

3.1.5 Пројектовање презентационог дела симулатора

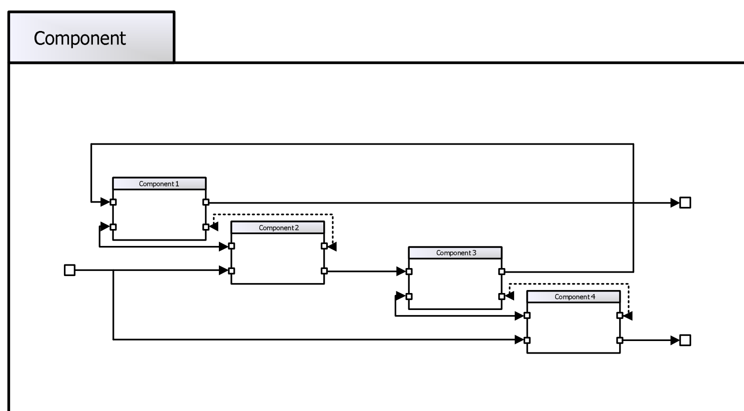
Презентациони слој симулатора представља везу онога што се симулира и корисника симулатора. Приликом пројектовања презентационог слоја симулатора треба дати одговор на питања како треба за изгледа резултат симулације. Одговором на ово питање дефинише се домен проблем. Домен проблема се обноси на то која је циљна група којој је симулатор намењен и шта та циљна група треба да има могућност да види и на који начин интерактивно да контролише поступак симулације. Одговори на ово питање се могу наћи у анализи симулатора који се користе у области архитектуре и организације рачунара. Приликом анализе су уочене две групе симулатора које за користе различите поступке за презентацију резултата симулације. Једна група омогућава интерактивну и визуелну симулацију уз континуално праћење резултата симулације, док други приступ тек након обављене комплетне симулације даје известан скуп статистичких извештаја о обављеној симулацији.

Приликом анализе изворног кода симулатора у овој области уочено је да су код већине симулатора презентациони и логични нивои два одвојена нивоа који треба независно да се развијају а који имају известан скуп додирних тачака. Оба нивоа имају циљ да опишу компоненте које се користе у обој области само користећи различита изражајна средства. Логички слој описује шта компонента треба да ради, а презентациони како корисницима може графичким путем да се прикаже поступак рада компоненте на логичком нивоу, како да повезују креиране компоненте, како да интерагују са компонентама, и како да прате ток симулације. У наставку ове главе су описани поступци представљања графичког интерфејса компонената, поступци интеракције са корисником током израде модела као и приликом уноса параметара, као и визуелизација тока симулације.

3.1.5.1 Визуелизација компонената

Основна сврха презентационог слоја симулатора је да омогући визуелно

праћење тока симулације. Приликом рада симулатора треба обезбедити приказивање резултата симулације користећи један или више панела. Панел је основни носећи елемент који омогућава позиционирање и исцртавање компонената. Панел омогућава груписање графичких елемената који означавају известан скуп компонената и веза. Панел поред презентационих компоненти и веза може садржати и други графички елементи који треба да олакшају поступак визуелног праћења симулације. Панел треба да чува информације о графичким елементима који се налазе на њему, али и о логичким компонената чије се графичке репрезентације налазе на панелу. Овај податак је неопходан јер је потребно омогућити кретање условљености између панела и праћења повезаности са свим повезаним панелима. Оно што треба обезбедити је уколико постоји веза између компонената на логичком нивоу и нивоу изнад у хијерархији онда таква иста веза трба да се оствари и између графичких компонената које панел представља. Уколико панел садржи фиксан суп графичких елемената и код кога се вредности придружених атрибута компонента директно интерпретирају онда се такви панели тешко могу интерпретирати. На слици 16 је дат пример панела са придруженим компонентама.



Слика 16: Шематски приказ панела са придруженим компонентама

Приликом анализе симулатора уочено је да постоји вели број начина за представљање истих логичких компонената. Логичке компоненте имају велики броји различитих графичких интерфејса који зависи од циљне групе којој је симулатор намењен. Приликом израде графичког интерфејса појединих компонената треба водити рачуна поред тога шта компонента представља и о месту унутар графичке површи на коју се компонента поставља. Треба водити

рачуна о међусобној повезаности компоненти и њиховом визуелном распореду у симулатору. Креирана графичка репрезентација логичке компоненте треба да одражава особине компоненте по питању броја и типа веза са другим компонентама, као и са могућношћу представљања параметара и стања компоненте. У овом раду је представљен концепт приступних тачака преко којих се обавља комуникација једне компоненте логичког слоја са другим компонентама, као и концепт веза које служе за пренос информација између компонената логичког слоја. Исти концепт се може применити и приликом поступка представљања графичког интерфејса компоненте.

Сваки графички интерфејс компонента се састоји од извесног броја графичких елемената које је могуће исцртати, чије је структура позната а који на визуелан начин треба да опишу понашање логичке компоненте. Изглед и распоред графичких елемената је независтан од карактеристика логичких елемената, али број и тип појединих врста графичких елемената зависи од броја приступних тачака логичке компоненте. Повезивањем приступних тачака на презентационог нивоу омогућава да се успостави повезивање и приступних тачака на логичком нивоу, чиме се креира листа повезаности компонената. Поред повезивања различитих компонената графичке компоненте обезбеђују да и могућност повезивање појединих поља логичке компоненте са презентационим пољем графичке компоненте. Takође могуће је извршити доделу различитих графичких елемената различитим вредностима појединих атрибут.

Графички интерфејс везе треба да има двојаку функцију. Једна је да омогући визуелно повезивање компонената и фазу пројектовања симулатора, а други да омогући визуелно праћење резултата током поступка симулације. Да би симулатор могао да обавља визуелно повезивање елемената потребно је постојање графичке компоненте која на визуелан начин обезбеђује повезивање логичких елемената са презентационим еквивалентима датих елемената унутар презентације. Овде се уочава битна разлика између концепта везе на логичком и графичком нивоу. На логичком нивоу веза представља апстрактан појам који означава да већи број компонената међусобно могу да комуницирају разменом порука, док у графичком смислу веза представља компоненту коју је потребно исцртавати. Поступак повезивања графичке и логичке компоненте није бијекција

(1-1) јер једна логичка веза може да има више различитих инстанци графичких компонената. Ово се најупечатљивије може видети у случајевима када је могуће представити везу са физички не повезаним симболом, али који је повезан на логичком нивоу.

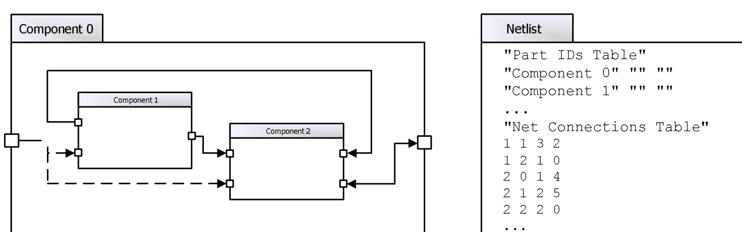
3.1.5.2 Интеракција са корисником

Први облик интеракције корисника са симулационим окружењем огледа се у фази пројектовања симулираних компоненти графичким путем. Приликом графичког пројектовања компоненти корисник позиционира графичке елементе на презентационом слоју. Овај поступак је од великог значаја јер помаже крајњем кориснику да лакше визуелно пати рад симулатора. Симулатори који се овде разматрају се превасходно користе у наставне сврхе и треба корисницима да помогну да разумеју како систем функционише. Основна два поступка интеракције су поступак позиционирања компонената и поступак повезивања компонената.

Приликом позиционирања графичких елемената постоје два приступа решавању апсолутно и релативно позиционирање. Код првог приступа обавља се апсолутно позиционирање графичких елемената код кога се свакој компоненти даје фиксна позиција унутар графичке површи на којој се приказује. Основна предност првог приступа је томе што је поступак пројектовања не захтева постојање сложених структура података у којима би се чувале информације о већем броју параметара већ свака графичка компонената чува информације независно од осталих. Овај приступ омогућава већи степен контроле над визуелним приказом. Основни недостаци овог приступа леже у томе што је приликом промене неког од параметара захтева за променом потребно проследити свим графичким компонентама јер у супротном се може догодити да се приликом промене резолуције догоди да известан део радне површи остао неискоришћен. Следећи недостатак овог приступа је велика тешкоћа приликом премештати групу графичких елемената јер информације о томе није могуће проследити већ је потребно појединачно померање графичких елемената. Код другог приступа решавању овог проблема графички елементи се релативно позиционирају у односу на почетак групе компонента. Овај начин размештања компонената омогућава решења проблема померања групе или резолуције који су се јавили код

првог приступа. Овакав приступ омогућава рад са групом компонената без промене њене унутрашње структуре. Основни недостатак лежи у томе што су информације о положају распрострите по већем броју компонената и што је доста тешко утврдити повезаност између компонената.

Поступак визуелног повезивања компонената треба да омогући пројектовање сложених компонената састављених од већ креираних. Основни проблем код визуелног се састоји у откривању да ли је графичка компонента која симболизује везу довољно близу приступној тачки неке друге графичке компоненти да би се сматрало да су они спојени. Поступак повезивања компонената на графичком нивоу повлачи и спајање одговарајућих компонената на логичком нивоу. Овај поступак треба да буде детерминистички спроведен да би се избегле ситуације код којих није јасно да ли је дошло до успостављања везе или не. Приликом повезивања компонената треба ићи на потпуно поклапање графичког симбола везе и дела елемента који представља порт компоненте. Поступак повезивања треба да буде са дискретним кораком како би се омогућило равномерно померање унутар презентације. Поред овог корака потребно је поставити и редундантан податак како би корисник знао да је обављено повезивање. На слици 17 је дат пример који представља исправно протумаченог повезивања логичких елемената на основу позиције презентационих елемената. Да би се кориснику омогућило да прати повезаност компонената односно да уочи у којим случајевима није дошло до повезивања јер је корак са којим је рађено повезивање велики потребно је да постоји прозор за приказ листе повезаних компонената. Поред овога потребно је да корисник може да графичким путем истакне које су компоненте повезане међусобно тако што одабере одговарајућу везу и тражи све повезане компоненте односно њихове приступне тачке.



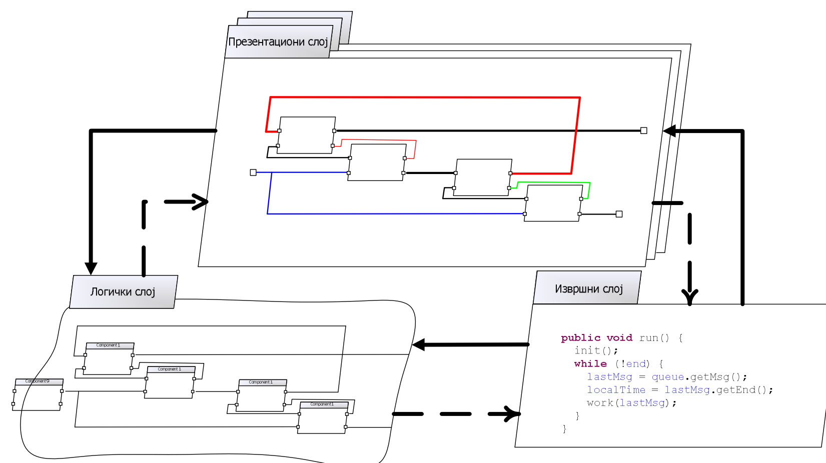
Слика 17: Пример визуелног повезивања елемената и генерисања листе повезаних компонената

3.1.5.3 Праћење тока симулације

Приликом анализе постојећих симулатора у области Архитектуре и организације рачунара уочено је више различитих начина на које корисници прате ток симулације и утичу на начин извршавања. Оно што симулатори треба да обезбеде је да кориснику у току рада посматра поједине атрибуте логичких компонената које се симулирају. Оно што треба нагласити је да се током симулације не посматра сама компонента него њени атрибути и репрезентације тих атрибута потребно је реализовати већи број начина за праћење тих атрибута како би симулирани систем сагледао са више различитих нивоа. У зависности од типа атрибута и његовог утицаја на остале компоненте потребно је изабрати погодан шаблон за праћење вредности атрибута. Постоји већи број начина за праћење вредности атрибута код којих се већина заснива на праћењу тренутне вредности док други захтева праћење комплетне историје извршавања. Један приступ подразумева праћење тренутне вредности појединих атрибута на презентационом моделу компонената. Други тип подразумева праћење временских облика појединих сигнала. Трећи тип подразумева праћење тренутних вредности у табеларном облику, уз могућност промене вредности. Четврти тип подразумева праћење текстуалног лога који прати извршавање симулације.

Први приступ праћењу тока симулације омогућава да се ток симулације прати користећи повезаност између атрибута и појединих графичких елемената. Овај приступ се најчешће примењује код симулатора који се може применити у случајевима када се жели посматра намењени за образовне сврхе, код којих се посматра понашање читавог система. Ово посматрање омогућава да се у произвољном временском тренутку види вредност свих сигнала, без информација о претходном стању. Овај поступак омогућава бољу прегледности тренутног стања компоненте, као и лакше откривање карактеристичних ситуација. Овај поступак има и својих недостатака јер од корисника захтева да иде кроз симулацију корак по корак што може да захтева доста времена за визуелну инспекцију. Графичке компоненте које се користе у овим системима су прилагођене томе да треба да интерпретирају вредности појединих атрибута. Поступак интерпретације се најчешће заснива на промени неке од карактеристика као што су боја или дебљина линије, или вредност на некој кабели. Корисник

поред пасивног праћења тока симулације има могућност и да интерактивно утиче на ток симулације променом вредности појединих атрибута и сигнала. На слици 18 је дат пример симулатора код кога је омогућено праћење тока извршења симулације користећи графичке елементе компонената.

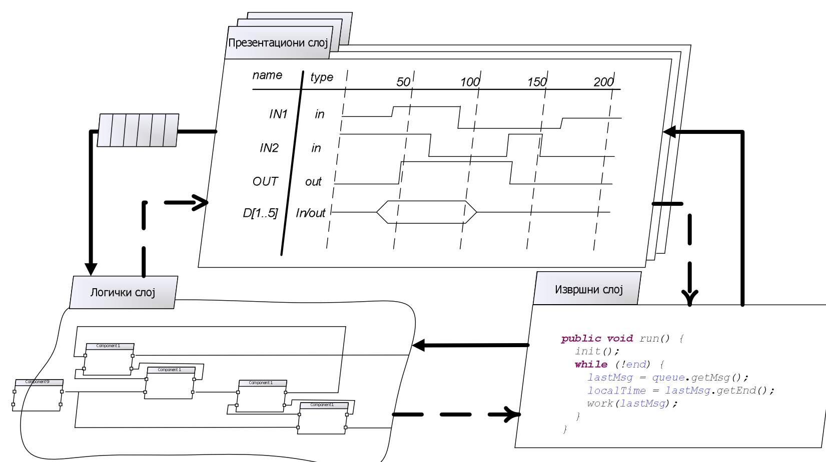


Слика 18: Шематски приказ архитектуре за праћење стања симулације користећи графичке репрезентације логичких компонената

Приказ резултата се заснива на коришћењу MVC пројектног узорка. Код овог узорка можемо да издвојимо три улоге које је потребно имплементирати у три различита слоја симулатора. Модел је представљен слојем логичких компонената које чувају информацију о логици унутар симулације. Презентациони слој представља слој који посматра резултате симулације, док извршни слој представља контролни део узорка. Интеракција започиње тако што корисник пре корисничког интерфејса задаје да се обави корак симулације. Ова информација синхроно пристиже извршном слоју симулатора који обавља симулацију интеракцијом са слојем логике. Када се ова интеракција заврши извршни слој обавештава презентациони слој да може да прочита нове вредности сачуване унутар слој логике и да их прикаже. Поред овог сценарија постоји и други у коме саме компоненте када се деси промена обавештава презентациони део како би се промена исцртала. Разлика између ова два приступа лежи у томе што приликом једне интеракције са корисником, односно покретања симулације за задати временски интервал може да дође до великог броја догађаја које је потребно исцртати и то може да вишеструку успори симулацију.

Други приступ праћењу тока симулације захтева праћење вредности неког

сигнала у дужем временском периоду користећи временски график. Овај облик омогућава кориснику праћење временских облика сигнала који садрже вредност појединих атрибута током комплетне историје промена вредности. Временски облици сигнала се обично састоји из x осе која представља протекло време, односно временску осу и y осе која репрезентује вредност сигнала у појединим тренуцима. Коришћење временских облика сигнала представља један од најзаступљенији приступа праћењу тока симулације. Овај приступ се користи и за интерактивно праћење тока симулације, али и за пакетску обаду код које се анализи обавља тек након завршетка комплетне симулације. Овај облик приказа има неколико варијанти које се користе за приказ сложених сигнала када је потребно урадити интерпретацију вредности које атрибути имају. Ова интерпретација се састоји из поделе опсега вредности, или промене текста који се исписује или промене боје која одговара појединим сигналима. Најчешћи начин за унос и промену атрибута и сигнала који кориснику дозвољавају интеракцију са симулатором се састоје у уносу нових вредности које неки од атрибута или сигнала може да има. На слици 19 је дат пример симулатора код кога је омогућено праћење тока извршења симулације користећи временске облике сигнала.

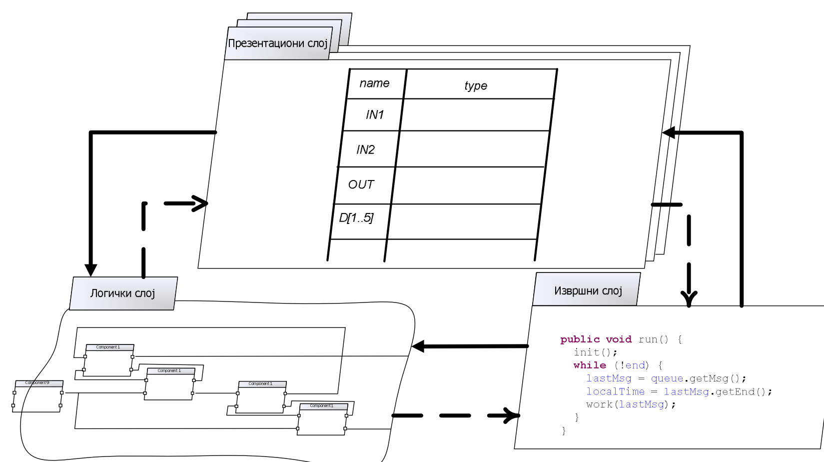


Слика 19: Шематски приказ архитектуре за праћење стања симулације користећи временске облике сигнала

Приказ резултата се и у овом случају заснива на коришћењу MVC пројектног узорка са истим распоредом слојева као и у случају када је визуелно потребно пратити само текуће стање симулације. Разлика се огледа у томе што је у овом случају поред текућег стања потребно пратити и комплетну историју

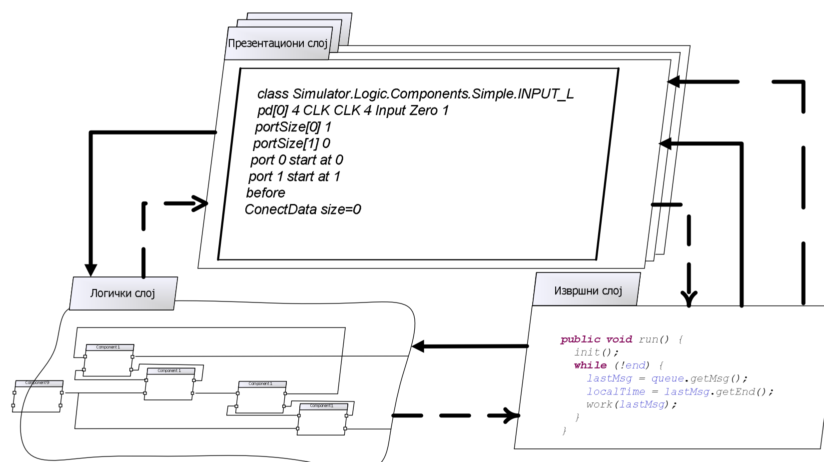
дешавања унутар симулације за одабране сигнале. Да би се ово постигло потребно је чувати историју које се приказује. Да би се ово постигло потребо је увести бафер за чување информације о стању. Овај бафер се разликује од бафера за чување историје код оптимистичког приступа извршавању симулација. Разлика је у томе што код оптимистичког приступа бафер служи да би исправи грешке које су последице спекулативног извршавања код кога је могуће у неком тренутку симулације имати информацију која није исправнас током извршавања симулације које је последица коришћеног алгоритма и разлике у локалном времену извршног слоја. Код презентационог слоја није дозвољено постојање неисправних ситуације већ је увек потребно кориснику представљати тачне резултате симулације а не спекулативног извршавања. Због потребе за коришћењем само валидних резултата у овом случају није могуће користити варијанту код које логички слој обавештава презентациони слој да је дошло до промене већ је потребно да то уради извршни слој у тренутку када постоје валидни резултати.

Трећи вид праћења тока симулације прегледом стања сигнала и атрибута у табеларном облику. Овај вид праћења стања система је донекле сличан првом приступу јер посматра стање у једном конкретном тренутку и табеларно бележи стања и промене појединих атрибута. Код овог приступа сваком атрибуту је додељено поље или скуп поља које посматра, као и назив посматраног атрибута. Овај приступ је доста сличан приступ који постоји у поступку извршавања програма корак по корак програма у одговарајућем режиму рада намењеном за тражење грешака. Приступ омогућава и кориснику да интерактивно може да промени вредност појединих атрибута и на тај начин утиче на ток симулације. На слици 20 је дат пример симулатора код кога је омогућено праћење тока извршења симулације користећи табеларни приказ атрибута и сигнала. Архитектура која се користи у овом случају треба да буде иста као и она у случају када је визуелно потребно пратити само текуће стање симулације.



Слика 20: Шематски приказ архитектуре за праћење стања симулације користећи табеларни приказ стања система у појединим тренуцима

Четврти приступ праћењу тока симулације је коришћење текстуалног исписа који представља лог одрађених акција у симулатору, као и промењених вредности сигнала и атрибута. Приликом исписа вредности атрибута и сигнала у лог треба водити рачуна о начину интерпретацију вредности појединих атрибута у складу за спецификацијом. Поступак интерпретације зависи од типа атрибута као и од функције којој је атрибут намењен. Овај приступ се може користити у случајевима када се прати рад симулатора који захтева информације о тренутном стању више од једног ресурса који се не могу представити у облику временских облика сигнала. На слици 21 је дат пример симулатора код кога је омогућено праћење тока извршења симулације користећи текстуални приказ атрибута и сигнала. Архитектура која се користи у овом случају може да буде иста као и она у случају када је визуелно потребно пратити само текуће стање симулације. Поред представљене варијанте могуће је користити и варијанту код које је извршни слој та ко генерише поруке које је потребно исписати.



Слика 21: Шематски приказ архитектуре за праћење стања симулације користећи текстуални приказ стања система у појединим тренуцима

Поред представљених начина за праћење тока симулације неки симулатори дозвољавају да се сам презентациони слој компоненте мења у зависности од вредности појединих атрибута. Овакав тип представљања резултата симулације захтева посебан поступак за приказ јер је постојање интерпретера који у зависности од типа компоненте припада генерише одговараће графичке интерфејсе. Начин на који ће вредности појединих атрибута бити интерпретирана зависи од тога ком презентационом делу је намењена. Презентациона компонента која посматра одређени атрибут мора да има информације о тупу атрибута који описује. Ових интерпретатора може да буде више и оно могу да буду независни од логичких и од презентационих компонената.

3.2 Аналитички модел времена извршавања симулације

У овој секцији су представљени аналитички модели који описују извршавање симулације на симулаторима дискретних догађаја развијених сходно описаном поступку. Представљени модели у изради симулатора треба да поседује аналитични облик очекиваних перформанси рада симулатора уколико се користе у конкурентним или дистрибуираном окружењу. Аналитички модели су прилагођени за оптерећења, карактеристике и начин употребе симулатора који се срећу приликом коришћења у настави у области Архитектуре и организације рачунара сходно поступцима и карактеристикама издвојеним у главама 2.1.2 „Преглед постојећих симулатора из области Архитектуре и организације рачунара“ и 2.1.3 „Преглед карактеристика симулатора“.

Уместо експлицитног моделовања свих обрада приликом њиховог конкурентног извршавања аналитички модел у овом раду карактерише репрезентативно понашање извршавања обраде на једном рачунару и утицаја које остали рачунари имају на то извршавање. Утицај осталих рачунара је обухваћен одговарајућим прорачуном параметара и понашања модела са једним рачунаром. Овај приступ претпоставља да извршавања на свим рачунарима имају сличне понашање (similar probabilistic behavior) ([79]-[88]).

Аналитички модел изложен у овом раду проучава перформансе система независно од времена између појединих приступа компонентама унутар извршавања симулације. За разлику од [89] предложени модел посматра напредовање симулације као континуални процес, а не у дискретним тренуцима приступа меморијским локацијама. Посматрају се само ситуације код којих је један процесор задужен за обраду само једног логичког процеса. Модел је окарактерисан следећим претпоставкама и параметрима:

1. Вероватноћа да је дошло до рестарта не зависи од места на коме се симулација налази, нити од тога да ли се симулација претходно већ рестартовала или не.

2. Расподела места где догађаји приступају подацима и компонентама је униформна унутар симулације.

3. Оптерећење по процесорима је равномерно распоређено тако да не долази до празног хода усред неадекватне расподеле компонената.

Модел је одређен помоћу следећих улазних параметра:

- Број симулираних компоненти M.
- Број логичких веза између симулираних компоненти V.
- Просечан број излаза који воде са једног порта компоненте F (fan out).
- Просечно логичко време извршавања симулације L.
- Просечно логичко време једног догађаја t_l.
- Просечно физичко време обраде једног догађаја t_p.
- Вероватноћа да се по иницијалном догађају генерише нови догађај P_f.
- Вероватноћа да је компонента активна у интервалу обраде догађаја P_e.
- Број процесора N на којима се симулација обавља.
- Просечно физичко време за транспорт догађаја између два рачунара t_t.

- Вероватноћа да порука остаје на истом процесору P_b, подразумевано 1/N.
- Вероватноћа формирања контекста P_c.
- Просечно физичко време потребно за обраду контекста компоненте t_b.
- Просечно физичко време за приказ компоненте/везе t_d.
- Вероватноћа да је компоненту/везу потребно приказати P_d.
- Просечно физичко време за обраду физике догађаја t_f.

3.2.1 Моделовање времена обраде логичког слоја

Алгоритми који омогућавају извршавање симулација у сваком свом кораку консултују компоненте логичког слоја симулатора. Компоненте логичког слоја представљају пасивне компоненте које немају свој ток контроле, али у великој мери утичу на сам ток симулације. Утицај на ток симулације се огледа на време потребно да се обради свака појединачна компонента. Времена обраде сваке компоненте логичког слоја су независне промењливе са истом расподелом. Физичко време потребно за обраде једне логичке компоненте се може моделовати Експоненцијалном расподелом за параметром t_p.

$$t \sim Exp(1/t_p)$$

Логичко време потребно за обраде догађаја једне логичке компоненте се може моделовати Експоненцијалном расподелом за параметром t_l.

$$t \sim Exp(1/t_l)$$

3.2.2 Моделовање времена обраде извршног слоја

У овој секцији је представљен математички модели времена потребно да се обави симулација на извршном нивоу симулатора. Време потребно да се обави симулација на извршном нивоу директно зависи од физичког времена које је потребно за обраду сваке компоненте. Ово време зависи од више параметара који су расподељену унутар свих слојева симулатора. Аналитички модел је представљен за временом и догађајима вођене симулације које се извршавају са једним током контроле у секвенцијалној симулацији као и са више токова контроле у паралелним симулацијама.

3.2.2.1 Временом вођена секвенцијална симулација

Алгоритам временом вођене симулације у сваком посматраном временском тренутку ради обраду свих компонената логичког слоја. Алгоритам

при израчунавању стања сваке компоненте одређује стање у које ће та компонента прећи, та нова стање се уписује у други скуп променљивих, и тек се на крају итерације чувају унутар самих компоненти. Просечно време потребно да се обави симулација се добија на основу броја итерација које је потребно обавити n, броја компонената које је потребно обрадити M и физичког времена потребног да се једна компонента обради tp.

$$T_t = n \cdot M \cdot t_p$$

Број итерација се добија на основу просечног логичког времена извршавања симулације L и просечног времена трајања логичког догађаја tl.

$$n = \frac{L}{t_l}$$

Пошто се у овом прорачунима занемарује време потребно за чување нових вредности у привремене променљиве математичко очекивање временског интервала потребног да се одраде сви догађаји се добија као:

$$E_t(T) = \frac{L \cdot M}{t_l} \cdot t_p$$

3.2.2.2 Догађајима вођена секвенцијална симулација

Алгоритам догађајима вођене симулације се огледа у обради догађаја убачених у листу која је сортирана према времену када је потребно извршити обраду тих догађаја. Обрада догађаја може да проузрокује креирање нових догађаја које је у неком тренутку потребно обрадити. Ове догађаје је потребно убацити у листу сортираних догађаја које је потребно обрадити у неком тренутку у будућности. Приликом одређивања времена потребно да се обави догађајима вођена симулација, потребно је проценти број догађаја које је потребно обрадити у посматраном временском интервалу. Број догађаја који треба да одради у датом временском интервалу се може моделовати Поасоновом случајном применљивом:

$$P_e(k) = e^{-\bar{k}} \cdot \frac{(\bar{k})^k}{k!}$$

Просечан број догађаја које је потребно обрадити се добија на основу просечног логичког времена извршавања симулације L, просечног времена трајања логичког догађаја tl, броја симулираних компонената M, вероватноће да је дату компоненту потребно обрађивати Pe у посматраном интервалу времена

барем једном, као и вероватноће да компонента генерише нови догађај на основу обрађеног догађаја P_f.

$$\bar{k} = M \cdot P_e \cdot \frac{1}{1-P_f} \cdot \frac{L}{t_l}$$

Пошто се у овом прорачунима занемарује време потребно за опслуживање и сортирање листе догађаја математичко очекивање временског интервала потребног да се одраде сви догађаји се добија као:

$$E_p(T) = \sum_{k=0}^{\infty} k \cdot P_e(k) \cdot t_p$$

$$E_p(T) = \frac{M \cdot P_e}{1-P_f} \cdot \frac{L}{t_l} \cdot t_p$$

3.2.2.3 Временом вођена паралелна симулација

Алгоритам временом вођене паралелне симулације у сваком посматраном временском тренутку ради обраду свих компонената логичког слоја расподељених на доступне радне станице. Паралелна временом вођена симулација се од секвенцијалне временом вођене симулације разликује по томе што понаша како да постоје већи број секвенцијалних симулација које је потребно синхронизовати у одговарајућим временским интервалима. Број паралелних симулација зависи од времена потребног да свака од симулација изврши обраду свих компонената које су додељене датој радној станици.

На основу разматрања у секцији „Временом вођена секвенцијална симулација" уз претпоставку да на свакој радној станици постоји исти број компонената добија се да је просечно време потребно за обраду на једној радној станици у једној итерацији симулационог алгоритма:

$$T_{tp1} = \sum_{i=0}^{\left\lceil \frac{M}{N} \right\rceil} t_{pi}$$

Интервал времена који је потребан да се обави симулација на N рачуна који користе алгоритам временом вођене симулације се мора димензионисати према најспоријој радној станици. Пошто се овде ради о рачунарима који имају исту расподелу оптерећења може се применити алгоритам за одређивање расподеле максималне вредности у случају N случајних променљивих са истом расподелом. Уколико се претпостави да време обраде једне итерације једне

компоненте има експоненцијалну расподелу онда је функција густине вероватноће суме већег броја таквих обрада таква да одговара Гама расподели:

$$t \sim \Gamma\left(\left\lceil \frac{M}{N} \right\rceil, \frac{1}{t_p}\right)$$

$$F_1(t) = \int_0^t \frac{\lambda^n \cdot e^{-\lambda \cdot x} \cdot x^{n-1}}{(n-1)!} \cdot dx$$

Где је:

$$n = \left\lceil \frac{M}{N} \right\rceil$$

$$\lambda = \frac{1}{t_p}$$

онда густине расподеле вероватноће максимума се може израчунати као:

$$F_N(t) = \left(F_1(t)\right)^N$$

односно:

$$f_N(t) = \left(F_N(t)\right)' = N \cdot \left(F_1(t)\right)^{N-1} \cdot \left(F_1(t)\right)'$$

Математичко очекивање временског интервала потребног да се одраде сви догађаји у једној итерацији на N рачунара се добија као:

$$E_{tp1N}(T) = \int_0^{+\infty} t \cdot f_N(t) \cdot dt$$

Број итерација се добија на основу просечног логичког времена извршавања симулације L и просечног времена трајања логичког догађаја tl.

$$m = \frac{L}{t_l}$$

Што за укупно време извршавања n итерација даје:

$$E_{tp}(T) = \frac{L}{t_l} E_{tp1N}(T)$$

3.2.2.4 Догађајима вођена паралелна симулација

У овој секцији је представљен математички модели времена потребно да се обави симулација на извршном нивоу симулатора за догађајима вођене паралелне симулације. Аналитички модел је представљен за конзервативни и оптимистички алгоритам симулације. Код оптимистичког алгоритма симулације биће

представљен аналитички модел за израчунавање вероватноће да дође до рестарта симулације, као и времена потребно да се симулација обави у случају да је контекст чуван инкрементално, периодично или у случају да се симулација враћа на почетак свог извршавања.

Моделовање очекиваног времена извршавања симулације код конзервативног алгоритма симулације

Конзервативни алгоритам симулације не дозвољава спекулативно извршавање симулације и прелази на извршавање догађаја само када је сигуран да се неће појавити неки догађај који се односи на неки претходни временски тренутак. Овај приступ захтева одређивање тог минималног интервала колико симулација на појединим рачунарима сме догађаја да обради пре него што се приступи синхронизацији за свим рачунарима и одређивања новог симулационог интервала. Број догађаја који сваки од рачуната треба да одради у датом временском интервалу се може моделовати Поасоновом случајном применљивом:

$$P_e(k) = e^{-\bar{k}} \cdot \frac{(\bar{k})^k}{k!}$$

Пошто се овде ради о системима са високом интеракцијом код које корисник одређује минималан интервал синхронизације, који најчешће износи један такт, просечан број догађаја који сваки рачунар треба да одради се добија на основу броја рачунара на којима се симулација извршава N, просечног логичког времена извршавања симулације L, просечног времена трајања логичког догађаја tl, броја симулираних компонената M, вероватноће да је дату компоненту потребно обрађивати Pe, као и вероватноће да компонента генерише нови догађај на основу обрађеног догађаја P_f.

$$\bar{k} = \frac{M \cdot P_e}{N} \cdot \frac{1}{1 - P_f} \cdot \frac{L}{t_l}$$

Интервал времена који је потребан да се обави симулација на N рачуна који користе конзервативни алгоритам симулације се мора димензионисати према најспоријем рачунару. Пошто се овде ради о рачунарима који имају исту расподелу оптерећења може се применити алгоритам за одређивање расподеле максималне вредности у случају N случајних променљивих са истом расподелом.

Нека је P(k) вероватноћа да неки рачунар треба да одреди тачно k догађаја, и нека је F(k) функција густине те вероватноће таква да важи:

$$F(k) = P(x \le k) = \sum_{i=0}^{k} e^{-\bar{k}} \cdot \frac{(\bar{k})^i}{i!}$$

онда густине расподеле вероватноће максимума има вредност:

$$P_N(x_N \le k) = \left(F(k)\right)^N$$

односно:

$$P_N(k) = \left(F(k)\right)^N - \left(F(k-1)\right)^N$$

$$P_N(k) = \left(\sum_{i=0}^{k} e^{-\bar{k}} \cdot \frac{(\bar{k})^i}{i!}\right)^N - \left(\sum_{i=0}^{k-1} e^{-\bar{k}} \cdot \frac{(\bar{k})^i}{i!}\right)^N$$

Математичко очекивање временског интервала потребног да се одраде сви догађаји на N рачунара се добија као:

$$E_{epp}(T) = \sum_{k=0}^{\infty} k \cdot P_N(k) \cdot L$$

$$E_{epp}(T) = \left(\sum_{k=0}^{\infty} k \cdot \left(\left(\sum_{i=0}^{k} e^{-\bar{k}} \cdot \frac{(\bar{k})^i}{i!}\right)^N - \left(\sum_{i=0}^{k-1} e^{-\bar{k}} \cdot \frac{(\bar{k})^i}{i!}\right)^N\right)\right) \cdot L$$

$$E_{epp}(T) = \left(\sum_{k=0}^{\infty} k \cdot \left(\left(\sum_{i=0}^{k} \frac{(\bar{k})^i}{i!}\right)^N - \left(\sum_{i=0}^{k-1} \frac{(\bar{k})^i}{i!}\right)^N\right)\right) \cdot e^{-N \cdot \bar{k}} \cdot L$$

Моделовање вероватноће рестарта код оптимистичког алгоритма симулације

Као један од основних параметара који утичу на просечно време извршавања симулације, E(T), је вероватноћа рестарта R који не представља улазни параметар. Овај параметар је потребно израчунати на основу преосталих улазних параметара. Вероватноћа да се симулација на посматраном рачунару не рестартује у тренутку x након обраде догађаја A када започиње обраду новог догађаја уколико је током обраде догађаја A примила s порука са догађајима од других симулација на другим рачунарима у тренуцима x1, ..., xs је дат следећим изразом:

$$P_{nrs}(x_1, x_2, ..., x_s \mid x) = p_{nr}(x_1) \cdot p_{nr}(x_2) \cdot ... \cdot p_{nr}(x_s)$$

Где је $p_{nr}(x)$ вероватноћа да се симулација неће рестартовати уколико је примила догађај од другог рачунара. Ова вероватноћа је одређена условом да је логичко

време пријема поруке x мање од времена када је порука послата увећаног за време транспорта:

$$x \le y + t_c$$

где t_c нормализовано физичко време транспорта поруке:

$$t_c = \frac{t_t}{t_p} \cdot t_l$$

Математичко очекивање да након обрађеног догађаја у тренутку x не дође до рестарта уколико је примљено s порука је:

$$\mathrm{P_{nrs}(x) = E(P_{nrs}(x_1, x_2, ..., x_s \mid x)) = \int_{-\infty}^{+\infty} ... \int_{-\infty}^{+\infty} g(x_1, x_2, ..., x_s \mid x) \cdot P_{nrs}(x_1, x_2, ..., x_s \mid x) dx_1 dx_2 ... dx_s}$$

где је g(x1, ..., xs | x) функција густине расподеле вероватноће места на коме се догађао пријем порука са догађаја од других рачунара. Пошто се ради о s независних порукама добија се:

$$g(x_1, x_2, ..., x_s \mid x) = g(x_1 \mid x) \cdot g(x_2 \mid x) \cdot ... \cdot g(x_s \mid x)$$

$$g(x_i \mid x) \sim Unif(x - t_l, x)$$

Одавде следи:

$$P_{nrs}(x) = \left(\int_{x-t_{ll}}^{x} \frac{1}{t_l} \cdot p_{nr}(x_1) dx_1 \right) \cdot ... \cdot \left(\int_{x-t_l}^{x} \frac{1}{t_l} \cdot p_{nr}(x_s) dx_s \right) = \left(\int_{x-t_l}^{x} \frac{1}{t_l} \cdot p_{nr}(t) dt \right)^s = \left(\int_{x-t_l}^{x} \frac{1}{t_l} \cdot \int_{0}^{t-t_c} \frac{1}{L} dy dt \right)^s$$

$$P_{nrs}(x) = \left(\frac{1}{L \cdot t_l} \cdot \int_{x-t_l}^{x} (t - t_c) dt \right)^s = \left(\frac{2 \cdot x - t_l - t_c}{2 \cdot L} \right)^s = \left(P(x) \right)^s$$

где је

$$P(x) = \frac{2 \cdot x - t_l - t_c}{2 \cdot L}$$

Вероватноћа да се догоди тачно s пријема порука током обраде посматраног догађаја се може моделовати Поасоновом случајном променљивом.

$$P_s(s) = e^{-\bar{s}} \cdot \frac{(\bar{s})^s}{s!}$$

Параметар за просечан број порука које је посматрани рачунар примио од почетка обраде догађаја А до тренутка x када се завршила обрада датог догађаја и треба прећи на обраду следећег догађаја се добија на основу броја примљених порука и броја догађаја по свакој примљеној поруци. Разлог зашто се узима број порука а не број догађаја је тај што већи број догађаја може да буде синхрон, то

јест да има исто време када их је потребно обрадити. Ово значи да ако неки догађај испуњава услов да не изазове рестарт то значи да ни остали синхрони догађаји из исте поруке који треба да се обраде у истом тренутку логичког времена не изазивају рестарт.

Пошто се ради о стационарној симулацији број примљених догађаја у посматраном интервалу је једнак броју послатих догађаја. Број послатих догађаја се добија на основу просечног броја креираних симултаних догађаја по обрађеном догађају, F, и вероватноће да догађај не остаје на истом рачунару (1-Pb). Треба нагласити да догађаји који остају на истом рачунару не изазивају рестарт зато што њихово време доспећа није мање од текућег логичког времена. Ово даје:

$$N_e = F \cdot (1 - P_b)$$

Број примљених порука у посматраном интервалу је једнак броју послатих порука. Број различитих рачунара који су примили поруку са посматраног рачунара се добија на основу броја послатих догађаја Ne и броја рачунара којима су дати догађаји могли да буду упућени N-1. Број порука се добија користећи генерализовани проблем расподеле рођендана (Generalized Birthday Problem):

$$N_m = (N-1) \cdot \left(1 - \left(1 - \frac{1}{N-1}\right)^{F \cdot (1-P_b)}\right)$$

Комбинујући претходна два израза добија се израз за просечан број порука које је посматрани рачунар примио током обраде посматране поруке:

$$\bar{s} = \frac{N_m}{N_e}$$

$$\bar{s} = \frac{N-1}{F \cdot (1-P_b)} \cdot \left(1 - \left(1 - \frac{1}{N-1}\right)^{F \cdot (1-P_b)}\right)$$

Израз за вероватноћу Pnr(x) да се посматрана симулације неће рестартовати у тренутку x, се добија на основу вероватноће Pnrs(x) да се симулација неће рестартовати уколико од почетка обраде посматраног догађаја до тренутка x било примљено тачно s порука, и вероватноће Ps(s) да је примљено тачно s порука:

$$P_{nr}(x) = \sum_{s=0}^{+\infty} P_s(s) \cdot P_{nrs}(x) = \sum_{s=0}^{+\infty} e^{-\bar{s}} \cdot \frac{(\bar{s})^s}{s!} \cdot (P(x))^s = e^{-\bar{s}} \cdot \sum_{s=0}^{+\infty} \frac{(\bar{s} \cdot P(x))^s}{s!} = e^{-\bar{s}} \cdot e^{\bar{s} \cdot P(x)}$$

Вероватноћа да се симулација рестартује се може одредити на основу вероватноће да се симулације није рестартовала.

$$1 - R(x) = P_{nr}(x)$$

Односно:

$$R(x) = 1 - e^{-\bar{s}\cdot(1-P(x))}$$

Математичко очекивање вероватноће рестарта се добија на основу R(x) и интервала на коме важи њен аналитички израз а то је интервал [tl+tc,L]:

$$R = E(R(x)) = E(1 - e^{-\bar{s}\cdot(1-P(x))})$$

$$R = \frac{L - t_l - t_c}{L} - \frac{1}{\bar{s}} \cdot \left(e^{-\bar{s}\left(\frac{t_l + t_c}{2\cdot L}\right)} - e^{-\bar{s}\left(1 - \frac{t_l + t_c}{2\cdot L}\right)} \right)$$

Моделовање очекиваног времена извршавања симулације уколико се симулација увек враћа на почетак

Посебан случај за претходна два приступа је ситуација када се контекст уопште не чува и када се симулације у свим конфликтним ситуацијама враћа на почетак свог извршавања. Овај приступ може да у неким специјалним случајевима да боље резултате него претходна два уколико је време потребно за чување контекста такво да превазиђе могућу добити насталу чувањем контекста.

Време извршавање једне симулације се као и у претходним случајевима описује на следећи начин:

$$E(T) = \sum_{i=0}^{+\infty} p_i \cdot \hat{T}_i$$

Где се вероватноћа p_i рачуна на идентичан начин док се приликом рачунања $\hat{T}_i$ узима да се симулација увек враћа на почетак свог извршавања, што даје за случај када нема рестарта:

$$T_0 = L$$

док за случај када се симулација рестартовала тачно i пута:

$$T_i = x_1 + x_2 + ... + x_i + L = L + \sum_{j=1}^{i} x_j$$

На основу претпоставке 1 добија се да су случајне променљиве x_i независне са униформном расподелом чиме се добија:

$$E(T_i) = \hat{T}_i = L + \sum_{j=1}^{i} E(x_j)$$

$$x_j \sim Unif(0, L)$$

$$E(x_j) = \int_0^L \frac{x_j}{L} \cdot dx_j = \frac{1}{L} \cdot \int_0^L x_j \cdot dx_j = \frac{L}{2}$$

$$E(T_i) = L + \sum_{j=1}^{i} \frac{L}{2} = L + i \cdot \frac{L}{2} = L \cdot \left(1 + \frac{i}{2}\right)$$

На основу претходних једначина добија се просечно време извршавања симулације приликом коришћења оптимистичног алгоритма симулације и враћања на почетак симулације у слушају конфликта:

$$E(T) = \sum_{i=0}^{+\infty} R^i \cdot (1-R) \cdot L \cdot \left(1 + \frac{i}{2}\right)$$

$$E(T) = \frac{2-R}{2 \cdot (1-R)} \cdot L$$

Моделовање очекиваног времена извршавања симулације код инкременталног чувања контекста

Један од начина на који се може обавити извршавање симулације користећи оптимистични алгоритам је инкрементално чување контекста. Инкрементално чување контекста захтева да се приликом обраде сваког догађаја извесно време утроши на чување контекста дате компоненте. Овакав вид симулације захтева извесно додатно време за обраду сваког појединачног догађаја а базира се на претпоставци да се неће све компоненте користити у задатом временском интервалу тако да неће бити потребе за обимним контекстом симулације.

Време извршавање симулације се може описати на следећи начин:

$$\mathrm{E(T)} = \sum_{\mathrm{i}=0}^{+\infty} \mathrm{p_i} \cdot \hat{\mathrm{T}}_\mathrm{i}$$

Где p_i представља вероватноћу да се симулација рестартовала тачно i пута, а $\hat{T}_i$ време извршавања симулације уколико се она рестартовала тачно i пута. Користећи претпоставку 1, вероватноћа p_i да се симулација рестартовала тачно i пута се може моделовати геометријском расподедом са параметром 1-R. Параметар 1-R представља вероватноћу успеха у секвенци Бернулијевих експеримената и представља вероватноћу да се симулација завршила без других

рестарта:

$$p_0 = (1-R)$$
$$p_1 = R \cdot (1-R)$$
$$\ldots$$
$$p_i = R^i \cdot (1-R)$$

Време извршавања симулације које се није рестартовала износи L.

$$T_0 = L$$

За случај да се симулација рестартовала тачно једном, место на коме се догодио рестарт је произвољан тренутак x1 од почетка до краја симулација, односно у интервалу од 0 до L. Место од кога симулација наставља извршавање након рестарта одговара произвољном временском тренутку y1. На основу претпоставке 3 случајна променљива y1 узима произвољну вредност од почетка симулација до места рестарта са униформном расподелом у интервалу од 0 до x1.

$$T_1 = x_1 + L - y_1 = x_1 - y_1 + L$$

За случај да се симулација рестартовала тачно два пута, место првог рестарта, x1, и место од кога симулација наставља, y1, се одређују на исти начин као и за случај тачно једног рестарта. Међутим место где може да настане следећи рестарт x2 мора бити после тренутка y1 јер с обзиром на оптимизацију рестарта симулација неће поново извршавати део симулације који је пре тренутка y1. Случајна променљива x2 има униформну расподелу на интервалу од y1 до L. Место од кога симулација наставља извршавање након рестарта y2 као и у проходном случају има униформну расподелу у интервалу од 0 до x2.

$$T_2 = x_1 + x_2 - y_1 + L - y_2 = x_1 - y_1 + x_2 - y_2 + L$$

На аналоган начин се добија да је време извршавања симулације које се рестартовала тачно i пута:

$$T_i = x_1 + x_2 + \ldots + x_i + L - y_1 - y_2 - \ldots - y_i$$
$$T_i = L + \sum_{j=1}^{i} (x_j - y_j)$$

Математичко очекивање времена извршавања симулације која се рестартовала тачно i пута се на основу овога може исказати као:

$$E(T_i) = \hat{T}_i = L + \sum_{j=1}^{i} (E(x_j) - E(y_j))$$

Случајне промењиве xi и yi су зависне и имају униформне расподеле чиме се добија:

$$x_1 \sim Unif(0, L)$$
$$y_1 \sim Unif(0, x_1)$$
$$...$$
$$x_i \sim Unif(y_{i-1}, L)$$
$$y_i \sim Unif(0, x_i)$$

Математичко очекивање промењивих xi и yi се може представити као:

$$E(x_i) = \int_0^L \frac{1}{L} \int_0^{x_1} \frac{1}{x_1} ... \int_{y_{i-1}}^{L} \frac{x_i}{L - y_{i-1}} dx_1 dy_1 ... dx_i$$
$$E(y_i) = \int_0^L \frac{1}{L} \int_0^{x_1} \frac{1}{x_1} ... \int_{y_{i-1}}^{L} \frac{1}{L - y_{i-1}} \int_0^{x_i} \frac{y_i}{x_i} dx_1 dy_1 ... dx_i dy_i$$

Решавањем једначине за математичко очекивање по y_i добија се зависност између математичких очекивања x_i и y_i:

$$E(y_i) = \int_0^L \frac{1}{L} \int_0^{x_1} \frac{1}{x_1} ... \int_{y_{i-1}}^{L} \frac{1}{L - y_{i-1}} \left(\int_0^{x_i} \frac{y_i}{x_i} dy_i \right) dx_1 dy_1 ... dx_i$$
$$E(y_i) = \frac{1}{2} \cdot \int_0^L \frac{1}{L} \int_0^{x_1} \frac{1}{x_1} ... \int_{y_{i-1}}^{L} \frac{x_i}{L - y_{i-1}} dx_1 dy_1 ... dx_i$$
$$E(y_i) = \frac{1}{2} \cdot E(x_i)$$

Решавањем једначине за математичко очекивање по x_i добија се зависност између математичких очекивања x_i и y_{i-1}:

$$E(x_i) = \int_0^L \frac{1}{L} \int_0^{x_1} \frac{1}{x_1} ... \int_0^{x_{i-1}} \frac{1}{x_{i-1}} \left(\int_{y_{i-1}}^{L} \frac{x_i}{L - y_{i-1}} dx_i \right) dx_1 dy_1 ... dy_{i-1}$$
$$E(x_i) = \int_0^L \frac{1}{L} \int_0^{x_1} \frac{1}{x_1} ... \int_0^{x_{i-1}} \frac{1}{x_{i-1}} \left(\frac{1}{L - y_{i-1}} \cdot \left(\frac{L^2}{2} - \frac{y^2{}_{i-1}}{2} \right) \right) dx_1 dy_1 ... dy_{i-1}$$
$$E(x_i) = \frac{1}{2} \cdot L \cdot \int_0^L \frac{1}{L} \int_0^{x_1} \frac{1}{x_1} ... \int_0^{x_{i-1}} \frac{1}{x_{i-1}} dx_1 dy_1 ... dy_{i-1} + \frac{1}{2} \cdot \int_0^L \frac{1}{L} \int_0^{x_1} \frac{1}{x_1} ... \int_0^{x_{i-1}} \frac{y_{i-1}}{x_{i-1}} dx_1 dy_1 ... dy_{i-1}$$
$$E(x_i) = \frac{1}{2} \cdot L + \frac{1}{2} \cdot E(y_{i-1})$$

Комбиновањем једначина за математичко очекивање x_i и y_i обија се диференцна једначина:

$$E(x_i) = \frac{1}{2} \cdot L + \frac{1}{4} \cdot E(x_{i-1})$$

$$E(x_{i+1}) - \frac{1}{4} \cdot E(x_i) = \frac{1}{2} \cdot L$$

Да би се поменута диференцна једначина решила потребно је наћи решавање линеарне диференцне једначине првог реда облак:

$$a_{n+1} - f(n) \cdot a_n = g(n)$$

Ова једначина има опште решење које гласи:

$$a_n = \prod_{j=0}^{n-1} f(j) \cdot \left(const + \sum_{k=0}^{n-1} \frac{g(k)}{\prod_{j=1}^{k} f(j)} \right)$$

Константа const се добија за a_0.

Решавањем дате једначине добија математичко очекивање за x_i:

$$\mathrm{E}(\mathrm{x_i}) = \frac{2}{3} \cdot \left(1 - \frac{1}{4^{\mathrm{i}}}\right) \cdot \mathrm{L}$$

На основу претходних резултата добије се:

$$E(T_i) = \left(\frac{8}{9} + \frac{i}{3} + \frac{1}{9 \cdot 4^i} \right) \cdot L$$

На основу претходних једначина добија се просечно време извршавања симулације са оптимизованим рестартом:

$$E(T) = \sum_{i=0}^{+\infty} R^i \cdot (1-R) \cdot \left(\frac{8}{9} + \frac{i}{3} + \frac{1}{9 \cdot 4^i} \right) \cdot L$$

$$E(T) = \frac{(2-R)^2}{(1-R) \cdot (4-R)} \cdot L$$

Приликом посматрања временске комплексности код оптимистичног алгоритма симулације са инкременталним чувањем контекста потребно је израчунати колико износи време потребно за враћања инкрементално сачуваног контекста из меморије и поновна иницијализација компонената. Подаци који су враћају су они који задовољавају услов да им је верзија максимална међу верзијама које су мање од оне верзије на коју се врши рестарт. Враћања верзије из оперативне меморије најједноставније је реализовати проласком кроз листу догађаја и поништавањем њихових ефеката. На овакав начин се добија да се време

потребно са враћање контекста може моделовати разликом између места на коме се догодио рестарт и места на које се симулација враћа, добија се:

$$T_i = x_1 + c \cdot (x_1 - y_1) + x_2 + c \cdot (x_2 - y_2) + ... + x_i + c \cdot (x_i - y_i) + L - y_1 - y_2 - ... - y_i$$

$$T_i = L + \sum_{j=1}^{i} (x_j - y_j) + c \cdot \sum_{j=1}^{i} (x_j - y_j) = T_i^{INC} + c \cdot \sum_{j=1}^{i} (x_j - y_j)$$

што даје

$$E(T) = \frac{(2-R)^2}{(1-R) \cdot (4-R)} \cdot L + c \cdot \frac{R}{(1-R) \cdot (4-R)} \cdot L$$

Где константа c добија вредност у зависности од физичког времена потребног за обраду једног догађаја и времена потребног за рестаурирање контекста једне компоненте:

$$c = \frac{t_b}{t_p}$$

Одатле следи да инкрементално чување контекста у поређењу са симулаторима који се враћају на почетак симулације, при истим вероватноћама рестарта, R, даје позитиван резултат уколико је:

$$\mathrm{c} < \frac{2 - \mathrm{R}}{2}$$

$$\mathrm{c}_{\mathrm{MAX}} = \frac{1}{2}$$

Моделовање очекиваног времена извршавања симулације приликом периодичног чувања контекста

Други приступ чувању контекста код оптимистичног алгоритма симулације захтева периодично чување контекста комплетне симулације на датом рачунару. Овај приступ има лошије перформансе што се места на које се симулација рестарује враћа али може генерално да има боље перформансе јер може да има мање кумулативно време потребно за чување контекста током симулације.

Овај приступ захтева да се у претходни модел унесу извесне модификације. У случају када се обавља симулација са периодичним чувањем контекста вероватноћа рестарта се моделује на исти начин као код система са инкременталним чувањем контекста, међутим разликује се модел који описује

време извршавања симулације у случају да је дошло до рестарта. Време извршавања симулације које се није рестартовала износи L.

$T_0 = L$

За случај да се симулација рестартовала тачно једном, место на коме се догодио рестарт је произвољан тренутак x1 од почетка до краја симулација, односно у интервалу од 0 до L. Овај рестарт је изазван догађајем коме одговарам временски тренутак y1, међутим место од кога симулација наставља извршавање након рестарта одговара тренутку yr1 на коме је последњи пут сачуван контекст а који је мањи или једнак тренутку y1. На основу претпоставке 3 случајна променљива y1 узима произвољну вредност од почетка симулација до места рестарта са униформном расподелом у интервалу од 0 до x1.

$$T_1 = x_1 + L - y_{r1} = x_1 - y_{r1} + L$$

За случај да се симулација рестартовала тачно два пута, место првог рестарта, x1, и место од кога симулација наставља, yr1, се одређују на исти начин као и за случај тачно једног рестарта. Међутим место где може да настане следећи рестарт x2 мора бити после тренутка yr1 јер с обзиром на оптимизацију рестарта симулација неће поново извршавати део симулације који је пре тренутка yr1. Случајна променљива x2 има униформну расподелу на интервалу од yr1 до L. Место од кога симулација наставља извршавање након рестарта yr2 као и у проходном случају има униформну расподелу у интервалу од 0 до x2.

$$T_2 = x_1 + x_2 - y_{r1} + L - y_{r2} = x_1 - y_{r1} + x_2 - y_{r2} + L$$

На аналоган начин се добија да је време извршавања симулације које се рестартовала тачно i пута:

$$T_i = x_1 + x_2 + \ldots + x_i + L - y_{r1} - y_{r2} - \ldots - y_{ri}$$

$$T_i = L + \sum_{j=1}^{i} (x_j - y_{rj})$$

Математичко очекивање времена извршавања симулације која се рестартовала тачно i пута се на основу овога може исказати као:

$$E(T_i) = \hat{T}_i = L + \sum_{j=1}^{i} E(x_j - y_{rj}) = L + \sum_{j=1}^{i} (E(x_j) - E(y_{rj}))$$

Место на коме се догодио пријем поруке која је изазвала рестарт xi има униформну расподелу на интервалу између места са кога је претходно започело

извршавање yri и краја симулације L, док је место на које се потребно вратити под условом да је сачуван комплетан контекст има униформну расподелу на интервалу од 0 до xi.

$$x_{i+1} \sim Unif(y_{ri}, L)$$
$$y_i \sim Unif(0, x_i)$$

Ово даје прву везу између зависних променљивих:

$$E(y_i) = \int_0^L \frac{1}{L} \int_0^{x_1} \frac{1}{x_1} \ldots \int_{y_{ri-1}}^{L} \frac{1}{L - y_{ri-1}} \left(\int_0^{x_i} \frac{y_i}{x_i} dy_i \right) dx_1 dy_1 \ldots dx_i$$

$$E(y_i) = \frac{1}{2} \cdot \int_0^L \frac{1}{L} \int_0^{x_1} \frac{1}{x_1} \ldots \int_{y_{ri-1}}^{L} \frac{x_i}{L - y_{ri-1}} dx_1 dy_1 \ldots dx_i$$

$$E(y_i) = \frac{1}{2} \cdot E(x_i)$$

Друга веза се добија на основу:

$$E(x_i) = \int_0^L \frac{1}{L} \int_0^{x_1} \frac{1}{x_1} \ldots \int_0^{x_{i-1}} \frac{1}{x_{i-1}} \left(\int_{y_{ri-1}}^{L} \frac{x_i}{L - y_{ri-1}} dx_i \right) dx_1 dy_{r1} \ldots dy_{ri-1}$$

$$E(x_i) = \int_0^L \frac{1}{L} \int_0^{x_1} \frac{1}{x_1} \ldots \int_0^{x_{i-1}} \frac{1}{x_{i-1}} \left(\frac{1}{L - y_{ri-1}} \cdot \left(\frac{L^2}{2} - \frac{y^2{}_{ri-1}}{2} \right) \right) dx_1 dy_{r1} \ldots dy_{ri-1}$$

$$E(x_i) = \frac{1}{2} \cdot L \cdot \int_0^L \frac{1}{L} \int_0^{x_1} \frac{1}{x_1} \ldots \int_0^{x_{i-1}} \frac{1}{x_{i-1}} dx_1 dy_{r1} \ldots dy_{ri-1} + \frac{1}{2} \cdot \int_0^L \frac{1}{L} \int_0^{x_1} \frac{1}{x_1} \ldots \int_0^{x_{i-1}} \frac{y_{ri-1}}{x_{i-1}} dx_1 dy_{r1} \ldots dy_{ri-1}$$

$$E(x_i) = \frac{1}{2} \cdot L + \frac{1}{2} \cdot E(y_{ri-1})$$

Трећа веза се добија на основу зависности yi и yr1. Случајна променљиве yi и yr1 су зависне тако да y1 имају униформну расподелу између два најближа сачувана контекста yr1 и yr1+d, чиме се добија:

$$E(y_j - y_{rj}) = \int_{y_{rj}}^{y_{rj}+d} \frac{y_j}{y_{rj} + d - y_{rj}} \cdot dy_j - y_{rj} = \frac{d}{2} = \frac{L}{2 \cdot (1+n)}$$

$$E(y_j - y_{rj}) = E(y_j) - E(y_{rj}) = \frac{L}{2 \cdot (1+n)}$$

Где n представља број контекста које је потребно сачувати у току симулације. Овај број контекста се може добити на основу времена обраде једног догађаја tl, логичког времена трајања симулације L, и вероватноће чувања контекста Pb.

$$n = 1 + \frac{L}{t_l} \cdot P_b$$

На основу ових веза се добија систем диференцних једначина:

$$E(x_{i+1}) = \frac{L}{2} + \frac{E(y_{ri})}{2}$$

$$E(y_i) = \frac{E(x_i)}{2}$$

$$E(y_{ri}) = \frac{E(y_i)}{2} - \frac{L}{2 \cdot (1+n)}$$

Комбиновањем ових диференцних једначина добија се:

$$E(y_{ri+1}) = \frac{E(y_{ri})}{4} + \frac{L}{4} - \frac{L}{2 \cdot (1+n)}$$

$$E(y_{ri+1}) - \frac{E(y_{ri})}{4} = \frac{L}{4} - \frac{L}{2 \cdot (1+n)}$$

Решавањем дате једначине добија математичко очекивање за y_{ri}:

$$E(y_{ri}) = \frac{4}{3} \cdot \left(1 - \frac{1}{4^i}\right) \cdot \left(\frac{L}{4} - \frac{L}{2 \cdot (1+n)}\right)$$

Као и математичко очекивање за x_i:

$$E(x_i) = \frac{L}{2} + \frac{2}{3} \cdot \left(1 - \frac{1}{4^i}\right) \cdot \left(\frac{L}{4} - \frac{L}{2 \cdot (1+n)}\right)$$

На основу добијених израза за xi и yri може се одредити математичко очекивање времена извршавања симулације која се рестартовала тачно i пута:

$$E(T_i) = L + \sum_{j=1}^{i} (E(x_j) - E(y_{rj}))$$

$$E(T_i) = L + \sum_{j=1}^{i} \left(\frac{L}{2} + \frac{2}{3} \cdot \left(1 - \frac{1}{4^i}\right) \cdot \left(\frac{L}{4} - \frac{L}{2 \cdot (1+n)}\right) - \frac{4}{3} \cdot \left(1 - \frac{1}{4^i}\right) \cdot \left(\frac{L}{4} - \frac{L}{2 \cdot (1+n)}\right) \right)$$

$$E(T_i) = \frac{8}{9} \cdot L + \frac{i}{3} \cdot L + \frac{L}{9} \cdot \frac{1}{4^i} + \frac{2}{9 \cdot (1+n)} \cdot L + \frac{i}{3 \cdot (1+n)} \cdot L - \frac{2}{9 \cdot (1+n)} \cdot L \cdot \frac{1}{4^i}$$

На основу претходних једначина добија се просечно време извршавања симулације која користи оптимистични алгоритам симулације и периодично чување контекста:

$$E(T) = \frac{(2-R)^2}{(1-R)\cdot(4-R)}\cdot L + \frac{(2-R)\cdot R}{(1+n)\cdot(1-R)\cdot(4-R)}\cdot L$$

$$E(T) = E(T^{inc}) + \frac{(2-R)\cdot R}{(1+n)\cdot(1-R)\cdot(4-R)}\cdot L$$

Где је E(Tinc) време потребно да се изврши симулација која обавља инкрементално чување контекста.

Приликом посматрања временске комплексности разматра се реализације алгоритма враћања сачуваног контекста из меморије и поновна иницијализација компонената. Подаци који су враћају су они који задовољавају услов да им је верзија максимална међу верзијама које су мање од оне верзије на коју се врши рестарт. Време враћања контекста из меморије може бити константно уколико се вршило чување комплетног контекста свих симулираних компонената. Уколико би се време враћања на одговарајућу верзију моделовало као константно добија се:

$$T_i = x_1 + c + x_2 + c + ... + x_i + c + L - y_1 - y_2 - ... - y_i$$

$$T_i = L + \sum_{j=1}^{i}(x_j - y_j) + \sum_{j=1}^{i} c = T_i^{INC} + c\cdot i$$

што даје

$$E(T) = E(T^{NR}) + c\cdot\frac{R}{(1-R)}$$

Где је $E(T^{NR})$ време потребно да се изврши симулација која обавља периодично чување контекста константа али код које је време чување једнако 0, и где c добија вредност у зависности од броја компонената на датом рачунару и времена потребног за рестаурирање контекста једне компоненте:

$$c = \frac{M}{N}\cdot\frac{t_b}{t_p}\cdot t_l$$

Одатле следи да периодично чување контекста у поређењу са симулаторима који се враћају на почетак симулације, при истим вероватноћама рестарта, R, даје позитиван резултат уколико је:

$$\mathrm{c} < \frac{2-\mathrm{R}}{2\cdot(4-\mathrm{R})}\cdot \mathrm{L}$$

$$\mathrm{c}_{\mathrm{MAX}} = \frac{1}{6}\cdot \mathrm{L}$$

3.2.3 Моделовање времена обраде презентационог слоја

Презентациони слој симулатора представља везу онога што се симулира и корисника симулатора. Представљена слојевита архитектура симулатора приликом графичког представљања резултата симулације треба да обави два корака. Први корак се састоји у томе да прочита контекст компоненте коју је потребно исцртати и другог корака у коме је потребно исцртати прочитани контест. Презентациони слој податке о вредностима које треба да представи добија од логичког слоја симулатора. Ове вредности се односе на стање логичких компонената и веза између компонената. На логичком нивоу веза представља апстрактан појам који означава да већи број компонената међусобно могу да комуницирају разменом порука, док у графичком смислу веза представља компоненту коју је потребно исцртавати.

Свака графичка компонента се састоји од извесног броја графичких елемената које је могуће исцртати, чије је структура позната а који на визуелан начин треба да опишу понашање логичке компоненте или везе. Укупно време потребно за исцртавање компоненти на презентационом слоју добија се као сума времена потребних за исцртавање презентационих компонената и презентационих веза.

$$T_{pr} = \sum_{i=1}^{M} t_i + \sum_{j=1}^{V} t_j$$

$$E(T_{pr}) = E\left(\sum_{i=1}^{N} t_{Ki} + \sum_{j=1}^{V} t_{Vj} \right)$$

Времена исцртавања су независне променљиве са истом расподелом. Време потребно за исцртавање једне презентационе компоненте, која одговара логичкој компонентини t_k или логичкој вези t_k, се може моделовати Експоненцијалном расподелом за параметром t_d.

$$t_K \sim Exp(1/t_d)$$

$$t_V \sim Exp(1/t_d)$$

На основу овога се добија:

$$E(T_{pr}) = \sum_{i=1}^{M} E(t_{Ki}) + \sum_{j=1}^{V} E(t_{Vj}) = \sum_{i=1}^{M+V} E(t)$$

$$E(T_{pr}) = (M + V) \cdot t_d$$

Овако добијено време се односи на ситуацију када је потребно исцртати све компоненте и везе које се налазе у симулатору. У случају да није потребно исцртати све компоненте, већ само одговараћу панелсе са компонента процентуално се умањује време. На основу претходне једначине и пероватноће исцртавања добија се коначан израз:

$$E(T_{pr}) = P_d \cdot (M + V) \cdot t_d$$

3.2.4 Моделовање времена обраде симулационог слоја

Овај слој симулатора постоји само код симулатора који имају већи број токова контроле који се извршавају у дистрибуираном окружењу код којих постоји потреба за комуникацијом измешу већег броја компонената логичког слоја који се налазе на тим дистрибуираном рачунарима. Компоненте симулационог слоја представљају активне компоненте које интерагују са извршним слојем симулатора као и са другим компонентама симулационог слоја на другим рачунарима. Време потребно за транспорт података између две логичке компоненте које се налазе на различитим рачунарима се могу сматрати да су независне променљиве са Експоненцијалном расподелом за параметром t_t.

$$t \sim Exp(1/t_t)$$

3.2.5 Моделовање времена обраде слоја физике

На исти начин на који компоненте логичког слоја утичу на ток симулације утичу и компоненте слоја физике компонената. Извршни алгоритми који обављају симулацију у сваком свом кораку симулације поред консултовања компонената логичког слоја консултују и компоненте слоја физике симулатора. Компоненте слоја физике представљају пасивне компоненте које немају свој ток контроле, али код симулатора које имају овај слој могу у великој мери утичу на ток симулације. Утицај се огледа на време потребно да се обради свака компонента. Времена обраде компонената слоја физике су независне променљиве са истом расподелом. Време потребно за обраде једне компоненте се може моделовати Експоненцијалном расподелом за параметром t_f.

$$t \sim Exp(1/t_f)$$

4 SLEEP СИМУЛАТОР

У овој глави је представљен симулатор дискретних догађаја опште намене SLEEP развијен у програмском језику Јава према описаној методологији. Симулатор је развијен као симулатор опште намене, након чега је прилагођен за коришћење у области Архитектуре и организације рачунара. Прилагођење се заснивало на развоју појединих компонената и модула који омогућавају симулацију дигиталних компонената које се користе у настави архитектуре и организације рачунара. Симулатор је представљен са два аспекта, један је аспект имплементације чији су детаљи представљени у секцији 4.1 „Реализација симулатора SLEEP“ док је друг аспект коришћење симулатора чији су детаљи представљени у секцији 4.2 „Начин коришћења SLEEP симулатора“.

Добра пракса у области развоја софтверских архитектура захтева модуларну структуру која има кључну улогу у случају развоја великих софтверских система [90]. Модули који су имплементирани унутар SLEEP симулатора су организовани у пет различитих слојева: извршни, симулациони, логички, презентациони и слој физике компонената. Извршни слој спроводи симулацију тако што израчунава следеће стање компонената логичког слоја симулатора, и биће представљен у секцији 4.1.2 „Симулациони алгоритми“. Симулациони слој симулатора омогућава расподелу догађаја и омогућава рад у дистрибуираном окружењу и биће представљен у секцији 4.1.3 „Рад у дистрибуираном окружењу“. Логички слој симулатора се односи на опис понашања симулираног система, и састоји се из функционалних описа понашања свих симулираних компонената, као и њихове повезаности, и биће представљен у секцији 4.1.1 „Логички слој симулатора“. Слој физике компонената има сличну структуру као и логички слој само се обавља опис компонената у смислу физичког понашања компонената користећи резултате логичког слоја симулатора, имплементација овог слоја биће представљена у секцији 4.1.5 „Слој физике компонената“. Презентациони слој симулатора омогућава кориснику графички приказ стања симулираних компоненти, као и интеракцију са симулатором и његови детаљи су представљени у секцији 4.1.4 „Презентациони слој

симулатора“. Поред ових општих слојева симулатор треба да омогући опис компонената користећи више различитих техника. Један начин је графичким путем унос компонената, други директно пишући код појединих компонената, и трећи користећи стандардни VHDL за опис компонената. Детаљи имплементације овог слоја су представљени у секцијама 4.1.1.3 „Писање кода користећи VHDL“

Са аспекта овог рада принципи конкурентног и дистрибуираног програмирања коришћени и имплементирани у SLEEP симулатору су од највећег значаја. У овој глави су представљени детаљи везани за реализацију симулатора које омогућавају рад у режиму са једним током контроле симулације (single thread), приликом рада у конкурентном окружењу са дељеном меморијом (multi thread) и у дистрибуираном окружењу (multi thread distributed). Детаљи везани за реализацију свих алгоритама који се користе у извршном слоју симулатора су представљени у секцији 4.1.2 „Симулациони алгоритми“.

4.1 Реализација симулатора SLEEP

У овој секцији су представљени имплементациони детаљи SLEEP симулатора. Комплексност предложеног SLEEP симулатора захтевала је стриктну хијерархијску организацију свих пакета унутар решења, како предложених интерфејса и комуникационих објеката, тако и организације поља, метода и алгоритама да би се остварила жељена функционалност. Пошто је симулатор развијен користећи објектно оријентисани програмски језик Јава део имплементације је представљен користећи језик за опис модела UML док су најзанимљивији делови кода представљени у свом изворном облику.

Приликом развоја SLEEP симулатора прво је на основу анализе постојећих решења, потребе за извођењем наставе из архитектуре и организације рачунара, и претходног искуства у развоју симулатора развијена методологија и поступак утврђивања шта све симулаторе треба да ради, а тек онда се прешло на одређивање како симулатор треба да ради. Приликом одређивања начина функционисања идентификовани су кључни дефови имплементације за сваки слој који су обухватала следећа три основна дела: скуп модула које треба да интерагују, скуп објеката који се у тој интеракцији користити и алгоритми који ће се примењивати.

На почетку реализације SLEEP симулатора прво су дефинисани интерфејси који омогућавају развој свих слојева предложеног симулатора. Скуп објеката који задовољавају тражене карактеристике омогућавао је вишеструка решења за поједине захтеве.

У овој секцији су сходно претходно описаном поступку прво утврђени детаљи организације функционалности симулатор, а онда начини реализације предложене организације сваког слоја симулатора. На почетку ове секције су дати детаљи о сваком слоју симулатора док су у секцијама које следе представљени имплементациони детаљи.

Логички слој симулатора се односи на компоненте и њихове међусобне везе на нивоу понашање, без уласка у физику компоненти. Компонента обавља одређену функцију конзумирањем улазних сигнала, трансформишући и каснећи их, и производе излазне сигнале. Компоненте логичког слоја су једини активне делови у симулираном систему, док остали слојеви симулатора само помажу тој обради. Свака компонента има одређени број приступних тачака, портова различитих типова (улаз, излаз и улаз-излаз) преко којих комуницира са осталим компонентама користећи остале слојеве симулатора. Компоненте су повезане везама између њихових портова који се користе за пренос порука/догађаја о току симулације.

Извршење слој симулатора се бави компонентама логичког слоја и порукама које они размењују, чинећи извршну инстанцу симулатора. Главне функције извршног слоја су обрада догађаја, израчунавање новог стања симулиране компоненти, праћење протеклог логичног времена, као и одређивање временског тренутка када догађаји треба да буду обрађена. У циљу обавља своје функције извршни слој инстанце симулатора треба да буде синхронизован са извршавањем унутар других слојева симулатора. Сва комуникација са другим слојевима или другим инстанцама извршног слоја које раде обраду догађаја се обавља преко симулационог слоја.

Дизајн симулатора треба да обезбеде ниво апстракције који крије комплексност дистрибуираних извршења. Увођење симулационог слоја ствара се утисак да се све компоненте извршавају у истој инстанци симулатора. Постојање симулационог слоја омогућава транспарентну поделу компоненти унутар

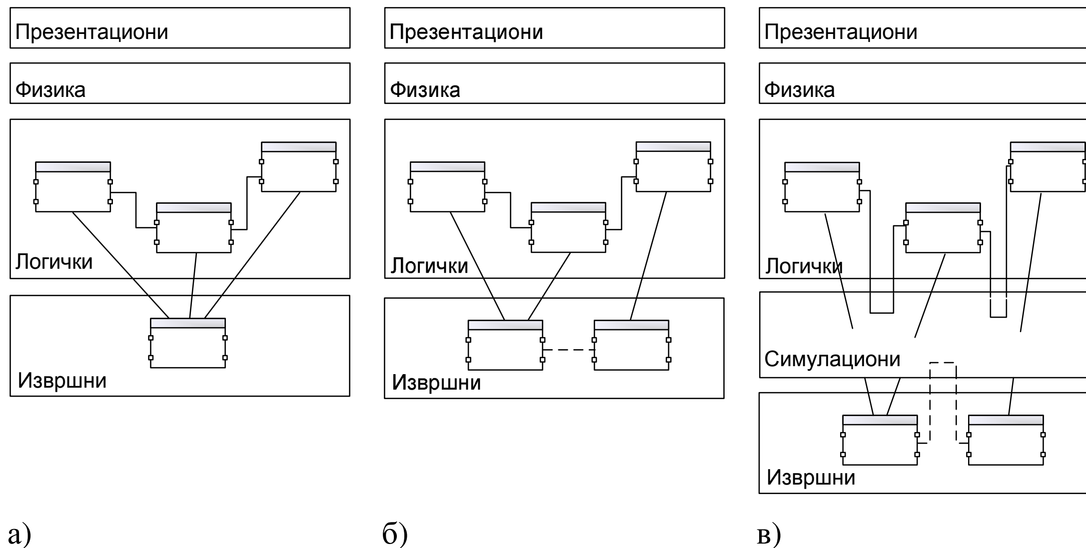
логичког слоја. Овим је обезбеђено да компоненте комуницирају међусобно, чак и ако се извршавају на различитим инстанцама симулатора.

Презентациони слој симулатора омогућава поглед компоненти логичког слоја. Поглед садржи више области рада за представљање хијерархије симулираних компоненти и вредности сигнала које размењују. Компоненте из логичког слоја могу да имају више графичких репрезентације у презентацију слој у зависности сврхе симулацију. Током симулације, презентациони слој треба да буде синхронизована са логичким слојем. Имајући у виду различите алгоритме извршавања, презентациони слој никад не би требало да даје вредности симулираних компоненти које се спекулативно израчунавају. Приликом пројектовања презентационог слоја посебну пажњу треба посветити минимизацији синхронизације коју овај слој има јер она може да буде ограничавајући фактор приликом извршавања саме симулације.

Слој физике симулираних компоненти се бави описом физичких карактеристика симулираних компоненти логичког слоја и њиховим просторним распоредом. Физика компонента се може моделовати као скуп ресурса, њиховим својствима и њиховом интеракцијом. Компоненте логичког слоја могу да имају више представа у физичком слоју у зависности од сврхе симулације. Током симулације, компоненте логичког нивоа обавештавају своје репрезентације у слоју физике око промене стања. Слој физике, у зависности од дате интеракције сваку промену стања претвара у одговарајућу промену физичких карактеристика компоненте. Што се осталих слојева симулатора тиче слој физике се понаша на идентичан начин као и логички слој и сва обрада је јединствена.

Комбиновањем различитих алгоритама и техника унутар SLEEP симулатора су имплементирани различити алгоритми у извршном слоју тако да постоје имплементације које се извршавају унутар једног тока контроле (single thread versions) и оне које захтевају већи бој паралелних нити (multithread versions). Ове верзије се извршавају унутар једне виртуелне машине, комуникацију и синхронизацију обављају користећи дељене променљиве и не захтевају измене симулационог слоја симулатора. Дистрибуирана верзија симулатора уводи симулације слој који омогућава транспарентну комуникацију логике и компоненти слој извршења који се може груписати и извршава одвојено

на различитим радним станицама. На слици 22 су приказане интеракције између слојева симулатора у ове три варијанте.



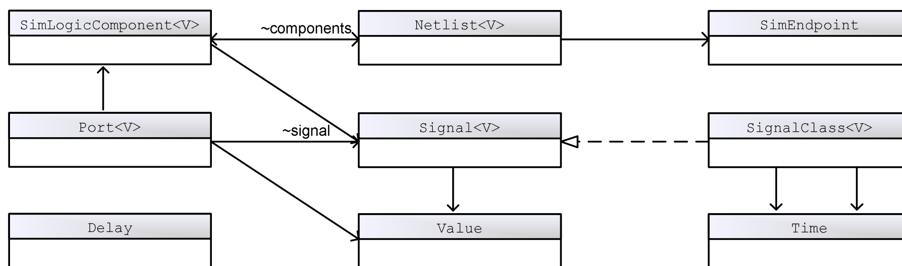
Слика 22: Комуникација и синхронизација између слојева SLEEP симулатора. Пуна линија представља комуникацију, а испрекидана линија представља синхронизацију. (а) Варијанта са једним током контроле; (б) Конкурентна варијанта са више нити; (в) Дистрибуирана варијанта са више нити.

Представљени слојеви симулатора чине грубу слику организације читавог система, и имају намеру да корисника упуте на могуће начине имплементације симулатора архитектуре и организације рачунара који је способан за рад у конкурентном и дистрибуираном окружењу. Дата имплементација је заснована на предложеној методологији, али нема намеру да намећу и строго дефинишу будуће начине имплементације сличних решења.

4.1.1 Логички слој симулатора

Реализација логичког слоја симулатора треба да обезбеди механизме за дефинисање компонената и веза између логичких компонената. Приликом дефинисања компонената ова слој даје начин за дефинисања начина на који компоненте примају сигнале преко веза, како користе те сигнале и интерне променљиве како би их обрадили и генерисали излазне сигнале. Да би компоненте могла да комуницира са осталим компонентама и размењивала сигнале компоненте дефинишу и листу приступних тачака, портова, преко којих комуницира са осталим компонентама. Хијерархија класа која дефинише ове

карактеристике је дата на слици 23.



Слика 23: Хијерархија класа логичког слоја симулатора

Као основни градивни блок у пакету rs.ac.bg.etf.zaki.sleep.logic.* јавља се интерфејс SimComponentLogic<V> чије имплементације треба да обезбеде конкретну имплементацију понашања компонената. Основне методе дефинисане интерфејсом SimComponentLogic могу се поделити у неколико група: обрада догађаја, метода за приступ параметрима компоненте и методе за чување постављање параметара симулације. Методе за обраду догађаја чине интерфејс према извршном слоју симулатора. Ове методе су са становишта симулације најзначајнијих део симулатора, јер представљају описе понаша симулираних компонената. Ове методе, у зависности од типа компоненте о којој се ради, треба да на портове поставе нове вредности сигнала који су специфицирани догађајем, и да на основу тих промењених вредности улаза израчунају ново стање у које компонента треба да пређе. То ново стање компоненте описује се промењљивама које могу да постоје унутар логичких компонената. На основу тих промењљивих и улазних компонента израчуната нове вредности тих промењљивих, али и генерисање догађаје које треба обрадити. Ти догађаји могу да иницирају промену излазних сигнал одређених портова, али и да утичи на понашање других компоненти. Методе за приступ параметрима компоненти омогућавају приступ карактеристикама компоненте које дају број и типове портова, интерне промењљиве и сигнале, број интерних компонената, као и њихову шему повезивања. Овде су сачувани најзначајнији параметри који специфицирају интерну структуру.

Основна класа која описује структуру неке компоненте је класа Netlist<V> која чува комплетну листу интерних компонената и њихових међусобних веза. Ова класа има могућност приступа свакој унутрашњој компоненти, њеним

променљивама и методама, али и могућност трансформације хијерархијске структуре која постоји у време пројектовања у денормализовану шему у којој су све компоненте истог хијерархијског нивоа која је потребна у време извршавања симулације. Поступак денормализације се обавља како би се све компоненте ставиле у један исти слој видљив извршном нивоу симулатора јер извршни слој води рачуна само о протеклом времену и ни на који начин не улази у хијерархијску структуру симулираних компоненти. Класа Netlist<V> је задужена и за динамичко креирање симулације на основу текстуалног записа, тако што користећи текстуално има компоненте креира инстанце класе користећи особину рефлексије. Креирање се односи на креирање компоненти и веза између њих. Да би симулација могла да се извршава на неком рачунару потребно је обезбедити да се на том рачунару нађу све дефиниције класа на које се те компоненте референцирају. Уколико то није испуњено компоненте се неће моћи креирати на одређеном рачунару тако да ће он бити недоступан симулацији. О овом на основу порука које прослеђује описана класа се обавештава симулациони слој симулатора, односно тачка на коме се симулација обавља. Поред потребе за трансформацијом листе компонената ова класа је такође неопходна да би логичка компонента која креира неку порука знала коме је та порука упућена. Начин на који је то сада реализовано је позивањем методе public List<Event<V>> transform(List<Event<V>> events) која листу улазних догађаја код којих није специфицирано одредиште већ само извориште трансформише у листу догађаја код које су познати сви параметри генерисаних догађаја. Овде је потребно нагласити да листе улазних догађаја и листе излазних догађаја не морају да буду исте дужине. Ово се догађа у случајевима када је излазни степен неке компоненте већи од један. Уколико је излазни степен већи од један онда се за свако одредиште креира нова порука. Ово је неопходно урадити јер дати догађаји не морају да буду упућени истом рачунару на коме се симулација обавља. Оваква реализација чини могуће формирање симулационог слоја симулатора на јединствен н начин. Ова комуницира између компонента логичког нивоа заснована је на постојању крајњих тачака комуникације. Крајње тачке комуникације описане су користећи два пара атрибута, први чини идентификатор изворишне компоненте и њен порт, а други идентификатор одредишне компоненте и њен порт. У случају да

компонента поставља неки интерни сигнал или свој излазни порт као изворишни порт се поставља вредност -1. Ова класа у још једном случају чини интерфејс према остатку система, а то је у случају када је потребно реализовати оптимистичку симулацију која захтева могућност повратка на неко претходно стање. Повратак на претходно стање захтева континуално чување контекста компонената система. Контекст се може чувати након сваке интеракција, ли и ређе у зависности од појаве вероватноће рестарта.

Класа које треба да буду задужене за везе које постоје између компонената имплементирају интерфејс Signal<V> који специфицира понашање сигнала у неком временском интервалу. Ова класа проверава да ли постоји промена вредности која може да иницира нови догађај који је потребно обрађивати. Оно што овај интерфејс специфицира се односи на понашање током трајања промене вредности које неки сигнал има у зависности од тога ко ту вредност поставља и које вредности имају преостале компоненте које приступају и постављају дати сигнал. Сигнал је омотач који чува вредност и листу догађаја који су довели до промене те вредности. Компоненте презентационог слоја користе вредност које чува овај објекат како би приказале тренутно понашање. Сигнал одговара везама између компоненти, али је и саставни део приступне тачке неке компоненте.

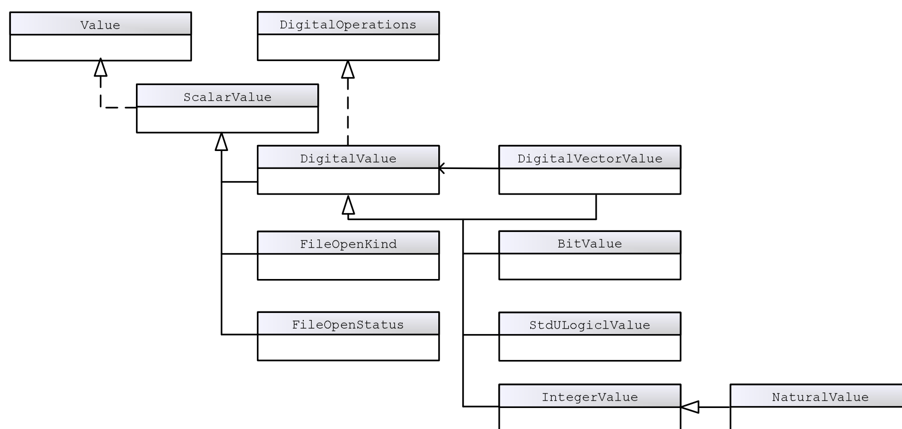
Комуникација унутар симулатора је заснована на догађајима које обрађује извршни слој симулатора тако што их у одговарајућем тренутку поставља као нову вредност сигнала повезаног на приступну тачку, порт компоненте. Ово значи да је потребно креирати објекте приступни тачака, али је и за сваку приступну тачку потребно креирати унутрашњи објекат одговарајућег типа сигнала. Пошто је SLEEP симулатор развоје као симулатор дискретних догађаја опште намене током развоја је предвиђена могућност рада са догађајима и сигналима произвољног типа, а самим тим и портовима који могу да примају те догађаје. Приликом креирања портова, инстанци класе Port<V>, потребно је специфицирати начин за нумеричку идентификацију тог порта на датој компоненти, унутрашње и спољашње име под којим се може приступити датом порту, ширину датог порта, тип порта који би требало вредности да узима вредност из скупа {IN, OUT, INOUT}, поље сигнала са којим је повезан дати порт, уколико је повезан, као и подразумевану вредност коју дати порт има у случају

неактивности осталих унутрашњих компонената.

4.1.1.1 Рад са дигиталним компонентама

У раду са дигиталним компонентама пошло се од претпоставке проширивости скупа вредности са којима компоненте могу да раде. Ови скупови вредности су моделовани тао да могу да се прилагоде било ком набројивом типу дискретних вредност који спада у интервал од 0 до MAXINT, што износи приближно две милијарде различитих дискретних вредности. Приликом представљања вредности набројивим простим типовима података мора се водити рачуна о компатибилности наведених типова података, као и алгоритмима за трансформацију наведених типова података једних у друге уз проверу да ли су наведене трансформације дозвољене. Ово је посебно битно када се ради о повезивању излази једне и улази друге компоненте које интерагују а чији скупови дискретних вредности припадају различитим скуповима, у том случају треба дефинисати правила интерпретације и конверзије података ових скупова. Овај скуп је најчешће један од следећих: {0, 1}, {0, 1, X}, {0, 1, X, Z} где: вредност 0 представља логичку нулу, вредност 1 представља логичку јединцу, Z стање високе импендансе, X представља недефинисано стање. Пошто има више различитих могућности за формирање и тумачење оваквих скупова направљена је комплетна хијерархија класа која одговара овим основним скуповима дискретних вредности.

Хијерархија класа које омогућавају рад са дискретним вредностима приказана је на слици 24. Ова хијерархија је урађена по угледу на хијерархију и скупове вредности које се могу наћи у програмском језику за опис хардвера VHDL.



Слика 24: Хијерархија класа дигиталних вредности

Као основа за да са оваквим вредностима јавља се интерфејс Value који је дефинисан у пакету rs.ac.bg.etf.zaki.sleep.logic.*, док су остале класе које имплементирају овај интерфејс смештене у пакету rs.ac.bg.etf.zaki.sleep.logic.digital.*.

Основне методе дефинисане интерфејсом Value могу се поделити у неколико група: постављање и дохватање дискретне вредности, трансформација текстуалног записа у вредност и обрнуто, као и група помоћних метода. Методе за дохватање дискретне вредности набројивог типа те вредности враћају користећи неки од простих типова података. Узето је да је све дискретне вредности могу да враћају вредност која је целобројног типа. Пошто у свим скуповима дигиталних вредности постоје две дискретне вредности, 0 и 1, формирана је и метода која треба задату вредност да трансформише у једну од ове две вредности сходно правилима тумачења која важе за одговарајући тип. Као посебан додатка претпостављено је да овај интервал може да се прошири тако да вредности могу да се конвертују и у интервал који има више вредност који могу да се представе променљивама које су типа long. Коришћење вредности типа int и long се може двојако тумачити. Прво тумачење је да се представи широк скуп дискретних вредности, други начин тумачења може да буде да се уради паковање одговарајућег типа, како би се уштедело на простору. Конкретна имплементације овог интерфејса треба да искористе ово као могућност. Следећа група метода обухвата методе које треба примљену вредност у текстуалном облику да трансформишу у одговарајућу вредност. Ово је метода која је доста често потребна, нарочито у поступку иницијализације и снимања стања објеката.

Последња група метода олакшава рад са објектима овог типа. Најчешће коришћена метода је метода која треба да одради копирање постојећег објекта и формирања нове инстанце како би се операције над инстанцама класа овог типа обављале на једнообразан начин.

Основна класа за рад са дискретним скупом вредности скаларног типа је класа ScalarValue која имплементира интерфејс Value. Ова класе не поседује никакве информације о дискретним вредностима које поједине класе могу да имају већ представља помоћну класу. Ова класа је формирана у складу са скупом метода које су дефинисане унутар VHDL програмској језика за рад са скаларним вредностима и представља омотач за конкретне вредности. Помоћне методе која пружа ова класа се користе те одређивање интервала које вредности могу да узму, броју различитих елемената унутар тог интервала, растућем или опадајућем поретку вредности унутар датог интервала, одређивање најмање и највеће вредности, као и за излиставање свих или одређених вредности из скупа дискретних вредности. Вредности се у овај скуп пакују у поретку који не мора да буде у складу са поретком који важи за саме дискретне вредности унутар скупа.

Као прва имплементација интерфејса Value која ради са дискретним вредностима јавља се класа DigitalValue која проширује класу скаларних вредности ScalarValue. Вредности у овом скупу нису ограничене и свака класа које проширују ову у већини случајева само проширује скуп основних вредности које може да прими. Пошто се овде ради о дигиталним вредностима две основне дигиталне вредности у укључене у вој скуп још приликом дефиниције ове класе. Ради се о вредностима 0 и 1. Ова класа има разрађене методе за конверзију у текста у интерну целобројну вредност сагласно скупу правила које важе за ову класу. Тај скуп вредности, и њихових текстуалних еквивалената је дефинисан статичким пољима ове класе. Уколико би постојала жеља за њихово другачије текстуално представљање приликом операције исписа потребно је поставити одговарајући дефиницију интерфејса Formater који ради превођење предефинисаног формата записа у тражени формат записа текста.

Класа BitValue проширује класу дискретних дигиталних вредностима DigitalValue на тај начин што дефинише правила интерпретирања вредности из интервала 0 и 1. Што се информација која ова класа носи она не би морала да се

корити већ би се увек могла користи над класа. Избегавање коришћења ове класе би са друге стране повећао могућност откривања грешке која може да се јави у коду уколико се користе класе које на различите начине тумаче основне дискретне вредности.

Класа StdULogiclValue проширује класу дискретних дигиталних вредностима DigitalValue. Ова класа је формирана на основу вредности типа std_ulogic пакета ieee.std_logic_1164.all. Класа дефинише вредности: 0 – логичка нула; 1 – логичка јединица; Z – стање високе импедансе; X – непозната вредност; W – слаб сигнал, не може се одредити да ли је 0 или 1; L – слаб сигнал који би вероватно требало да буде логичка нула; H – слаб сигнал који би вероватно требало да буде логичка јединица; - – било шта, није од интереса; U – не иницијализована вредност која тек треба да буде постављена. Поред проширења ове класе дефинисане су и методе за интерпретацију свих вредности и њихове конверзије у основни тип дигиталних вредности.

Основна класа за рад са дискретним скупом вредности векторског типа је класа VectorValue која имплементира интерфејс Value. Класа представља омотач која чува низ дискретних вредности. Сама класа не чува никакве информације о дискретним вредностима које поједине класе могу да имају већ представља помоћну класу. Ова класа је формирана у складу са скупом метода које су дефинисане унутар VHDL програмској језика за рад са векторским вредностима и представља омотач за конкретне вредности. Помоћне методе која пружа ова класа се користе за одређивање позицијама где вредности могу да се нађу, позиционирање унутар тог интервала, приступ одговарајућем елементу, алгоритме за конверзију низа вредности у скаларни тип, конверзија у целобројну вредност типа int или long, поретку (растућем или опадајућем) елемената унутар датог интервала, одређивање најмање и највеће вредности, као и за излиставање сви или одређених вредности из скупа дискретних вредности.

Класа IntegerValue проширује класу дискретних дигиталних вредностима DigitalValue из интервала од Integer.MIN_VALUE до Integer.MAX_VALUE. Класа представља омотач око целобројних вредности дужине 32 бита. Класа се користи или како замена за целобројне вредности или као класа која служи за паковање векторских вредности како би се убрзао рад са њима. Постоје дефинисане

операције које омогућавају да се ова класа користи у оба контекста. Ова класа је даље проширена како би се уместо рада само са целобројим вредностима успоставио рад са природним бројевима. То је учињено класом NaturalValue која покрива скуп од 0 до Integer.MAX_VALUE.

Класа RealValue проширује класу дискретних дигиталних вредностима DigitalValue из интервала од Float.MIN_VALUE до Float.MAX_VALUE у складу са VHDL препорукама. Класа представља омотач око разломљене вредности у покретном зарезу представљене на дужини од 32 бита. Унутар ове класе дефинисане су методе за конверзију у остале просте типове података, као и скуп метода које дефинишу аритметичке и логичке операције надо овим типовима података.

Приликом рада са дигиталним вредностима потребно је дефинисати скуп метода које ће бити доступне за те операције. Методе које свака класа која треба да имплементира уколико жели да обавља операције са дигиталним вредностима су дате у интерфејсу DigitalOperations пакета rs.ac.bg.etf.zaki.sleep.logic.digital. Све ове методе су дате тако да задовоље спецификацију скупа метода дефинисаних програмским језиком за описом хардвера VHDL. За рад са дигиталним вредностима, скаларним и векторским, је специфициран скуп метода који је дат Табелом 11.

Табела 11: Скуп операција над дигиталним вредностима

Тип операција	Скуп метода
Логичке операције	and, or, not, nand, nor, xor, xnor
Аритметичке операције	add, sub, concatenate, plus, minus, abs
Померачке операције	ssl, srl, sla, sra, rol, ror
Операције множења	mul, div, mod, rem
Релационе операције	equal, different, less, lessOrEqual, greater, greaterOrEqual

Имплементација интерфејса DigitalOperations која ради са дискретним скаларним вредностима је класа DigitalValue. Ова класа прво учитава целобројне вредност које представљају операнде над којима треба обавити одговарајућу операцију. Када се обаве операције ради се поново конверзија целобројне вредности у одговарајући тип који се враћа као резултат. Резултат је нова

инстанца како би се омогућиле операције над аритметичким изразима. Класа не поседује информације о самим дискретним вредностима над којима треба да ради већ то раде конкретне инстанце класа. Уколико је један од операнада низовског типа онда се врши конверзија у низовски тип па се тек онда обавља операција.

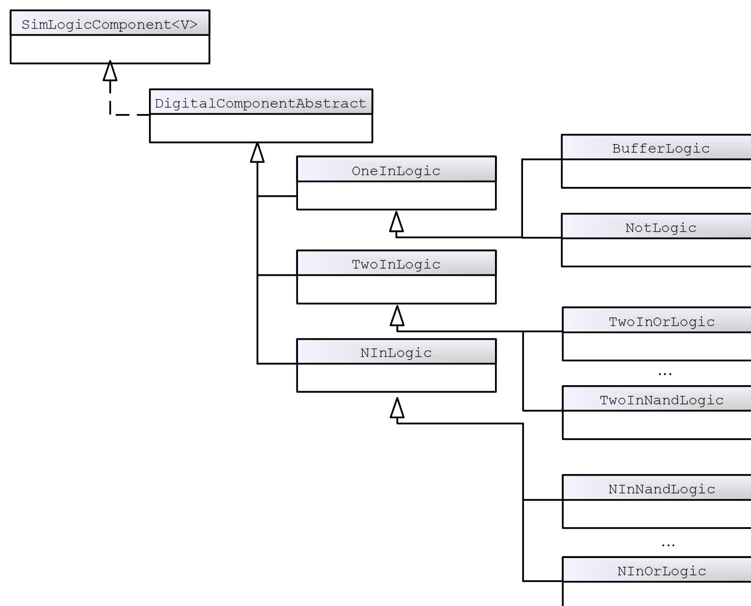
Имплементација интерфејса DigitalOperations која ради са дискретним вредностима векторског типа је класа VectorValue. Ова класа прво учитава бит по бит дигиталне вредности које представљају операнде над којима треба обавити одговарајућу операцију. Операција се обавља над појединачним битима без формирања целобројне вредности где би се тај међурезултат сачувао. Када се обаве операције ради се смештање добијеног резултата у инстанцу одговарајуће класе векторског типа која се враћа као резултат. Резултат је нова инстанца како би се омогућиле операције над аритметичким изразима. Класа не поседује информације о самим дискретним вредностима над којима треба да ради већ то раде конкретне инстанце класа. Уколико је један он операнда скаларног типа онда се врши конверзија у низовски тип па се тек онда обавља операција.

У поступку конверзије и рада са наведеним операцијама користи се помоћна класа која обавља операције конверзије. То је класа Conversion која има одговарајући скуп статичких метода које воде рачуна о улазним и излазним зиповима података и њиховој међусобној условљености. Приликом формирања нових типова податак било би потребно проширити ову класу одговарајућим статичким методама или динамички проследити инстанцу која треба да ради конверзију која би била сачувана унутар овог статичког објекта у његовој мапи.

4.1.1.2 Опис компонената користећи језик Јава

Основни начин за дефинисање градивних блокова симулатора је програмски језик Јава. Симулатор омогућава дефинисање компонената користећи ова програмски језик, како и суп предефинисаних интерфејса, класа и метода. На слици 25 је дат део хијерархије класе које се користе за опис компонената користећи Јава програмски језик. Компоненте описане користећи програмски језик Јава чине основне комбинационе и секвенцијалне мреже које омогућавају даљи рад и постепено подизање сложеност компонената које се симулирају. У раду са Јава компонентама корисник треба да користи класе и интерфејси, као и библиотеке компонената које су дате у пакетима

rs.ac.bg.etf.zaki.sleep.logic.digital.* као и rs.ac.bg.etf.zaki.sleep.logic.digital.java.*.



Слика 25: Хијерархија класа предефинисане библиотеке дигиталних компонената писаних користећи програмски језик Јава

Приликом пројектовања компонената водило се захтевима да се омогући опис компонената које могу да се искористе и за опис компонената користећи VHDL језик за опис хардвера, али и основне принципе објектно оријентисаног програмирања. На слици 26 је дат пример компоненте дефинисане користећи Јава програмски језик. У наставку је објашњен поступак дефинисања логичких компонената на примеру двоулазног И кола користећи програмски језик Јава које одговарају комбинационим и секвенцијалним мрежама.

Да би нека Јава класа могла да се користи унутар симулатора као логичка компонента која ради са дигиталним вредностима мора да имплементира одређен број интерфејса. Ово је потребно да се уради како би компоненте могле да се користе и комбинују са осталим дигиталним компонентама. Уколико се не користе дате компоненте и интерфејси, опет је могуће креирати нове инстанце компонената, али се оне не могу сматрати дигиталним компонента у смислу овог симулатора.

```
package rs.ac.bg.etf.zaki.sleep.logic.digital.java.simple;

import rs.ac.bg.etf.zaki.sleep.logic.*;
import rs.ac.bg.etf.zaki.sleep.logic.digital.*;
import rs.ac.bg.etf.zaki.sleep.util.*;

public class TwoInAndLogic extends DigitalComponentAbstract {

    Port<DigitalValue> a;
    Port<DigitalValue> b;
    Port<DigitalValue> c;

    public TwoInAndLogic() {
        int _cnt = 0;
        a = new Port<DigitalValue>(_cnt, "a", "a", 1, Port.IN,
            new SignalClass<DigitalValue>(), new StdULogiclValue(0));
        _portIds.put(a, _cnt++);
        _ports.add(a);
        b = new Port<DigitalValue>(_cnt, "b", "b", 1, Port.IN,
            new SignalClass<DigitalValue>(), new StdULogiclValue(0));
        _portIds.put(b, _cnt++);
        _ports.add(b);
        c = new Port<DigitalValue>(_cnt, "c", "c", 1, Port.OUT,
            new SignalClass<DigitalValue>(), new StdULogiclValue(0));
        _portIds.put(c, _cnt++);
        _ports.add(c);
    }

    public void twoInAndLogic_behav() {
        DigitalValue result = a.getVal().and(b.getVal());

        if (result.getIntValue() != c.getVal().getIntValue()) {
            _event = new Event<DigitalValue>(_lTime, _lTime
                + getDeley(_portIds.get(c), _portIds.get(c)), _id, -1, _id,
                            _portIds.get(c), result);
                _result.add(_event);
        }
    }

    public void combinationalLogic() {
        twoInAndLogic_behav();
    }
}
```

Слика 26: Пример двоулазног И кола дефинисаног користећи језик Јава

Да би се користиле дефиниције дигиталних компонената потребно је на почетак класе ставити да она користи следеће библиотеке:

```
import rs.ac.bg.etf.zaki.sleep.logic.*;
import rs.ac.bg.etf.zaki.sleep.util.*;
```

Пошто двоулазно И коло које се пројектује представља дигиталну компоненту потребно је укључити и дефиницију библиотеке за рад са дигиталним вредностима. У овом случају ради се о SLEEP библиотеци:

```
import rs.ac.bg.etf.zaki.sleep.logic.digital.*;
```

Спецификација дигиталних компонената се остварује тако што се декларише да креирана класа наслеђује апстрактну класу за рад са дигиталним сигналима и вредностима DigitalComponentAbstract. Ово је реализовано би се избегло да сва компоненте има комплетну дефиницију свих помоћних метода и поља креирана је апстрактна класа дигиталних компонената. Тако да назив класе двоулазних И кола гласи:

```
public class TwoInAndLogic extends DigitalComponentAbstract
```

Пошто се овде ради о двоулазном компоненти то значи да компонента има два улазна порта и један излазни порт. Улазни портови су a и b сваки може да буде дигитална вредност, док је излазни порт c такође типа дигиталних компонената std_logic. Након овога је потребно сваки од ова три порта декларисати као објекат одговарајуће типа:

```
Port<DigitalValue> a;
Port<DigitalValue> b;
Port<DigitalValue> c;
```

На овај начин је извршена декларација портова, а њихова дефиниција се обрађује унутар конструктора класе TwoInAndLogic. Ово значи да је сваки порт потребно креирати унутрашњу објекат порта одговарајућег типа. SLEEP симулатор има могућност рада са догађајима произвољног типа, а самим тим и портовима који могу да примају те догађаје. Конструктор класе портова има следећи потпис:

```
public Port(int id, String inName, String outName, int
range, int type, Signal<V> signal, V defaultValue)
```

где поље id представља нумерички идентификатор тог порта на датој компоненти; поље inName представља име под којим се дати порт компоненте види изнутра, а поље outName име под којим се дати порт види споља; уколико се ради о вишеструком порту који одговара магистрали онда се поставља поље range на одговарајућу вредност, уколико ништа није специфицирано узима се вредност 1; портовима је могуће специфицирати тип коме припадају који узима вредност из скупа {IN, OUT, BUFFERED, INOUT, SIGNAL}; Поље signal представља објекат преко кога се обавља комуникација са остатком симулатора. Ово поље се чита сваки пут када се жели приступити вредности; Поље defaultValue каже коју вредност треба да има порт пре почетка рада. Иницијализација сваког од портова се обавља на следећи начин:

```
a = new Port<DigitalValue>(_cnt, "a", "a", 1, Port.IN,
```

```
        new SignalClass<DigitalValue>(), new
StdULogic1Value(0));
```

сходно правилима за инцијализацију портова. Када се заврши иницијализација портова онда се сваки порт ставља у одговарајуће интерне листе портова.

```
_portIds.put(a, _cnt++);
_portIds.add(a);
```

Објекат _portIds чува редни број порта на компоненти, док објекат _portIds чува који портови све постоје на компоненти. Није пожељно све ово чувати унутар истог објекта јер ширина портова може да буде различита од 1.

Понашање двоулазног И кола је дефинисано унутар методе:

```
public void twoInAndLogic_behav()
```

Операција коју ова дигитална компонента логичког нивоа обавља описана је следећим изразом:

```
a.getVal().and(b.getVal())
```

Када је израчуната нова вредност, ту вредност је потребно доделити сигналу придруженом излазном порту након специфицираног логичког кашњења. Постављање нове вредности неког сигнала се постиже креирањем догађаја који ће ту вредност поставити на жељено месту у жељеном тренутку. Креирање догађаја одговара инстанци класе Event, а пошто се овде ради само о дигиталним вредностима то одговара генеричком позиву Event<DigitalValue>. Конструктор класе догађаја има следећи потпис:

```
public Event(long lTimeCreated, long lTime, long srcID,
        int srcPort, long dstID, int dstPort, V data)
```

где поље lTimeCreated представља логичко време када је креиран догађај који је потребно обрадити. Временски интервали су дати користећи целобројну вредност дужине 64 бита. Уколико интервал није довољно прецизан постоји и класа Time која проширује класу Delay; поље lTime представља логички тренутак када је потребно извршити одговарајући догађај; поља srcID и srcPort идентификују извориште догађаја, тј. компоненту која је креирала догађај и са ког порта дате компоненте је тај догађај упућен; поља dstID и dstPort идентификују одредиште догађаја, тј. компоненту којој је догађај намењен, односно на који порт те компоненте је потребно поставити нову вредност дату датим догађајем; Поље data представља објекат који преноси вредност коју је потребно поставити на одредишни порт одредишне компоненте у специфициран. Уколико се ради о транспорту догађаја које је потребно проследити излазном порту као изворишни

порт поставља се вредност -1. На основу овога изложеног инцијализација догађаја се обавља користећи следећи код:

```
_event = new Event<DigitalValue>(_lTime,
         _lTime + getDeley(_portIds.get(c),
_portIds.get(c)),
         _id, -1, _id, _portIds.get(c), result);
```

Када се креира догађај потребно је тај догађај убацити у бафер како би био обрађен у специфицираном временском тренутку:

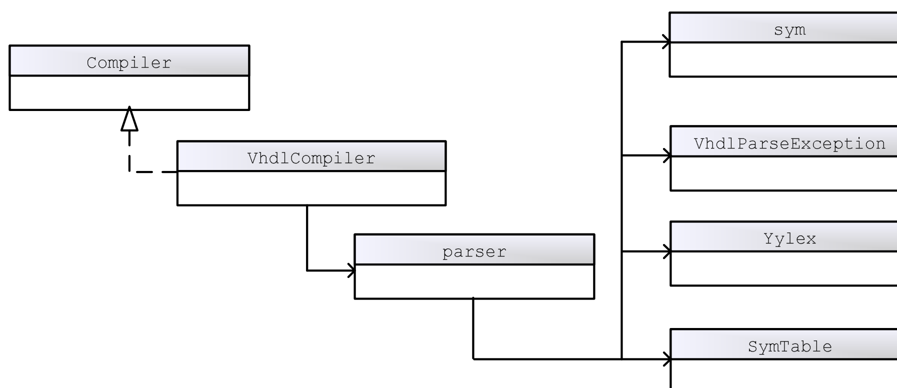
```
_result.add(_event);
```

Извршавање ове методе се остварује тако што је у методу која описује понашање комбинационих мрежа убацимо позив методе за ову компоненту.

```
public void combinationalLogic() {
        twoInAndLogic_behav();
}
```

4.1.1.3 Опис компонената користећи VHDL

Да би се омогућило коришћење што већег броја готових компонената у SLEEP симулатор је уграђена могућност дефинисања компонената користећи VHDL језик за опис хардвера. Компоненте дефинисане користећи VHDL језик се интерно трансформишу у компоненте писане у програмском језику Јава и интегришу у текући пројекат. У поступку трансформације кода писаног користећи језик VHDL искоришћене су класе дате у пакету rs.ac.bg.etf.zaki.sleep.logic.digital.vhdl, као и стандардне библиотеке за превођење и парсирање кода JFlex и CUP. На слици 27 је дата комплетна хијерархија класе које се користе за превођење кода писаног у језику VHDL у код писан у Јави.



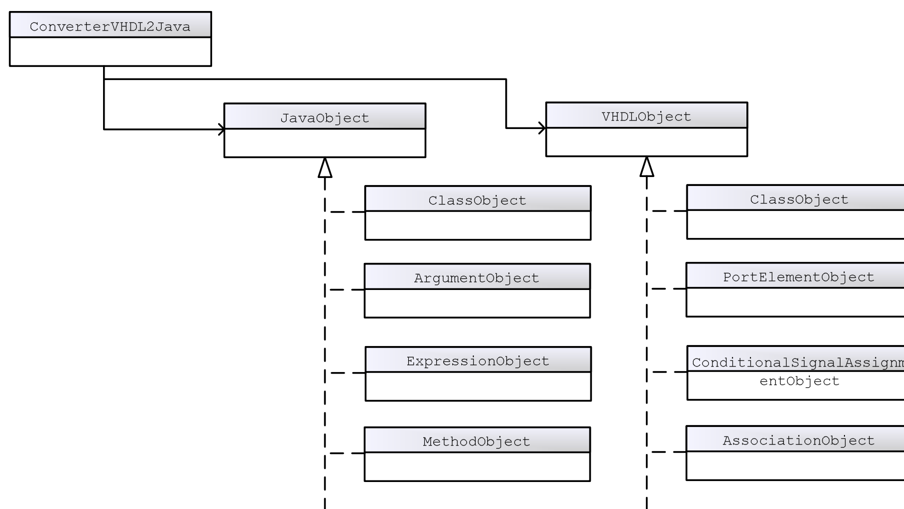
Слика 27: Хијерархија класа са превођење VHDL кода

Основна класа која обавља синтаксну и семантичку анализу је класа parser.

Ова класа је креирана користећи алат CUP на основу задате спецификације програмског језика VHDL и правила задатих за његову трансформацију у одговарајући Јава код задат vhdl.cup датотеком. Алат CUP генерише Јава код LALR(1) парсера на основу спецификације у виду безконтекстне атрибутивно-транслационе граматике. У овој класи се врши целокупна синтаксно-семантичка анализа, као и генерисање извршног Јава програма и табеле симбола. Тако да се класа parser може сматрати основном класом преводиоца. Због прилагођавања кода за раду у SLEEP симулатору, методе ове класе се позивају из класе SleepCompiler.

Методе и поља у овој класи се могу на две групе. Прву групу метода и поља чине основне методе и поља које је формирао CUP генератор преводилаца. Ове основне методе и поља се користе да би омогућили препознавање и трансформацију улазне VHDL датотеке у излазну Јава датотеку. Другу групу метода и поља чине помоћне методе и поља специфично направљена да би омогућиле једноставнију и бржу обраду улазне датотеке и формирање излазне. Спецификација метода и поља ове друге групе је такође обављена користећи улазну CUP датотеку где је било назначено да се ради о додатним методама и пољима које је потребно уградити у код.

Основна метода ове класе је метода parse. Ову методу је генерисано CUP програма и у њој се на основу препознатих синтаксних структура врше дефинисане акције. Акције могу бити унос променљиве у табелу симбола позивањем методе putToSymTab, препознавање језичке конструкције позивање метода putToExecFile, препознавање грешке позивањем методе за додавање елемента у листу errorList, формирање излазног Јава фајла или пријава грешака. Симболи препознати приликом превођења се чувају у објекту simbols класе Tab, док симболи за које се још увек не зна вредност се чувају у објекту класе HashMap<Integer, String>. Резултат превођења се чува у хијерархији објекта која је приказана на слици 28.



Слика 28: Хијерархија класа са чување међукода и конверзију кога из VHDL у Јава програмски језик

Класа VhdlComiler представља класу преводиоца за VHDL која преводи код у Јава програмски језик погодан за извршавање унутар SLEEP симулатора. Пошто је предвиђено постојање већег броја симулатора унутар развијеног система стављено је да сваки преводилац треба да имплементира интерфејс Compiler. Основна метода класа које имплементирају интерфејс Compiler је метода compile којом се врши превођење VHDL кода у Јава код за SLEEP симулатор. Као параметри методи се прослеђују улазни фајл и директоријум у којем ће бити смештени излазне датотеке, ови аргументи се добијају из остатка симулатора. Излазне датотеке су: резултујући Јава код преведеног VHDL кода, табела симбола, као и табела пресликавања изворног VHDL кода у резултујући Јава код. Име резултујућих Јава датотека се аутоматски формирају на основу имена ентитета који су преведени, док се имена датотеке са табелом симбола и табеле пресликавања формирају на основу улазних VHDL датотека. Основни објекти ове класе су lekser који је инстанца класе Yylex и compiler који је инстанца класе Parser. Она представљају лексички и синтаксно-семантички анализатор, респективно. Сам лексер се не користи из ове класе, већ само служи да се проследи парсеру информација од којег ће лексичког анализатора добијати лексеме. Метода compile креира објекат лексичке анализе и иницијализује га улазном датотеком коју је потребно обрадити, и позива методу parse класе parser, чиме се стартује превођење. Уколико је превођење успешно завршено,

преводилац ће регуларно изаћи из методе за превођење, док ће у случају да се појавила нека грешка емитовати изузетак који даје ближе информације о томе зашто се појавила грешка и на ком се месту догодила.

Унутар класе parser користе се услуге класе Obj за смештање препознатих делова Јава кода. Препознати код се чува у пољима ове класе која одговарају структури коју имају Јава класе. Та поља чувају информације о имену пакета коме класа припада (packageName), листи библиотека које треба учитати (importNames), имену класе (className), имену класе коју наведена класа наслеђује (extendsNames), листу имена интерфејса које дата класа имплементира (implementsNames), листу поља са одговарајућим типовима (fields), листу конструктора са њиховим потписима и имплементацијама (constructors), листу метода са њиховим потписима и имплементацијама (methods), као и листу коришћених унутрашњих класа (classes). Поред ових поља класа parser садржи и поља која воде рачуна о грешкама које су се јавили у поступку превођења. У случају да није дошло до грешака при превођењу, резултујући Јава програм ће бити сачуван у излазној датотеци. У случају да се приликом превођења открије грешка чувају се основни подаци о откривеној грешци: тип грешке, линији изворног VHDL програма у којем се јавила грешка и код које лексеме је препозната.

Класа sym представља класу података којом се дефинишу семантичке вредности препознатих токена у току лексичке анализе. Ову класу је генерисао програма CUP обрадом улазног фајла са дефинисаном спецификацијом VHDL језика за SLEEP симулатор (vhdl.cup). Ова класа заједно са класом parser, представља синтаксно-семантички анализатор. И док се у класи parser обавља сва делатност потребна за анализу и генерисање извршног програма, у овој класи се генерално дефинишу објекти које лексер прослеђује парсеру.

Као лексички анализатор за препознавање елемената писаних користећи језик VHDL користи се класа Yylex. Она класа није писана у програмском језику Јава већ је добијена користећи програм JFlex обрадом датотеке VHDL.lex. Основна функција креираног лексичког анализатора је да препозна унапред дефинисане лексичке јединице лексеме и токене. Ова класа се користи као први корак приликом трансформације кода писаног користећи VHDL у Јаву тако што

јој се прослеђује код писан у језику VHDL након чега се препознате лексеме прослеђују синтаксно-семантичком анализатору на даљу обраду. Да би се обезбедила пуна компатабилност са језиком VHDL дефинисани су сви потребни типови лексичких јединица које препознаје овај језик. Како би се омогућило да симулатор за сваку лексичку јединицу поседује и додатне информације било је потребно проследити и додатни скуп података који обухвата информације типу лексеме и тренутној линији изворног програма. Да би се омогућило прослеђују лексичке јединице коришћена је метода yylex() коју је на основу спецификације генерисао JFlex, која је додатно прилагођена за потребе симулатора.

Поред класе Obj за чување резултата парсирања унутар класе parser се користе услуге и већег броја наменских класа које су изведене из класе Obj. Ради се о класама које су специјализоване за чување појединих делова препознатог кода. Информације о препознатим класама се чувају у објекту типа ObjClass која садржи скупове одговарајућих поља класа изведених из класе Obj ради лакше трансформације у циљни Јава код. Информације о препознатим пакетима, али о мапирању тих VHDL пакета на Јава пакете чувају се унутар класа ObjPakage. Информације о препознатим пакетима који се користе унутар пројекта, као и о мапирању тих VHDL пакета на Јава пакете налазе се унутар инстанци класа ObjImport. Класе ObjPakage и ObjImport чине целину пошто на исти начин одређују начин пресликавања VHDL пакета на Јава пакете. Класа у којој се чувају информације о доступним типовима података и начину пресликавања имена VHDL компоненти у имена Јава класа и објеката је дата класом ObjFields. Информације о препознатим методама, али о мапирању метода различитих VHDL компонената на различите методе различитих Јава објеката чувају се користећи инстанце класа ObjNethod. Када се ради о препознавању конструктора класе или његовог динамичког формирања на основу поља и метода унутар класе као и о начинима мапирању из VHDL језика користи се класа ObjConstructor. Класе ObjMethod и ObjConstructor чине целину пошто на исти начин одређују начин пресликавања VHDL извршног кода на Јава код.

Класа VhdlParseException представља класу изузетака која се користи приликом превођења кода писаног користећи VHDL у код писан у Јави. Ова класа представља декларисани изузетак настао проширивањем основне класе изузетака

java.lang.Exception додавањем текстуалног поља comment које омогућава слање порука делу преводиоца који служи за обраду изузетака.

На сликама 29 и 30 је дат пример VHDL кода и кода добијеног превођењем датог кода у еквивалентни Јава код. Превођење програма писаних користећи програмски језик за опис хардвера у Јава програмски језик се обавља у два корака.

```
use ieee.std_logic_1164.all;
-- definition of a full adder
entity FULLADDER is
	port (a, b, c: in std_logic;
		sum, carry: out std_logic);
end FULLADDER;
architecture fulladder_behav of FULLADDER is
begin
	sum <= (a xor b) xor c ;
	carry <= (a and b) or (c and (a xor b));
end fulladder_behav;
```

Слика 29: Пример компоненте дефинисане користећи VHDL

Трансформација кода се обавља примењујући специфициран скуп правила. У наставку ове секције ће бити дат пример трансформације датог VHDL кода у резултујући Јава код.

Свака Јава класа треба да садржи библиотеке које специфицирају изворни пакет у коме се налазе основне дефиниције о компонентама које се користе. Те дефиниције су потребне како би SLEEP умео да интерпретира код који прима. Да би се ово остварило потребно је ставити на почетак класе следећа два реда:

```
import rs.ac.bg.etf.zaki.sleep.logic.*;
import rs.ac.bg.etf.zaki.sleep.util.*;
```

Уколико класа садржи користи спољашње библиотеке (use кључна реч) онда се ради трансформација те полазне VHDL библиотеке у циљну Јава библиотеку. У примеру потребно је трансформисати коришћену библиотеку:

```
use ieee.std_logic_1164.all;
```

у коришћење одговарајуће Јава библиотеке. Пошто се овде ради о стандардним ieee библиотекама те библиотеке су већ уграђене у SLEEP симулатор па пресликавање наведеног реда даје следећи Јава исказ:

```
import rs.ac.bg.etf.zaki.sleep.logic.digital.*;
```

Када се приликом парсирања наиђе на ентитет креира се Јава класа која носи исто име као дати ентитет. У примеру се јавио следећи ред са дефиницијом ентитета:

```
entity FULLADDER is
```

```
import rs.ac.bg.etf.zaki.sleep.logic.*;
import rs.ac.bg.etf.zaki.sleep.util.*;
import rs.ac.bg.etf.zaki.sleep.logic.digital.*; //use ieee.std_logic_1164.all;
public class FULLADDER extends DigitalComponentAbstract {
    private static final long serialVersionUID = 1L;
    Port<DigitalValue> a;
    Port<DigitalValue> b;
    Port<DigitalValue> c;
    Port<DigitalValue> sum;
    Port<DigitalValue> carry;
    public FULLADDER() {
        int _cnt = 0;
        a = new Port<DigitalValue>(_cnt, "a", "a", 1, Port.IN,
            new SignalClass<DigitalValue>(), new StdULogic1Value(0));
        _portIds.put(a, _cnt++);
        _ports.add(a);
        b = new Port<DigitalValue>(_cnt, "b", "b", 1, Port.IN,
            new SignalClass<DigitalValue>(), new StdULogic1Value(0));
        _portIds.put(b, _cnt++);
        _ports.add(b);
        c = new Port<DigitalValue>(_cnt, "c", "c", 1, Port.IN,
            new SignalClass<DigitalValue>(), new StdULogic1Value(0));
        _portIds.put(c, _cnt++);
        _ports.add(c);
        sum = new Port<DigitalValue>(_cnt, "sum", "sum", 1, Port.OUT,
            new SignalClass<DigitalValue>(), new StdULogic1Value(0));
        _portIds.put(sum, _cnt++);
        _ports.add(sum);
        carry = new Port<DigitalValue>(_cnt, "carry", "carry", 1, Port.OUT,
        new SignalClass<DigitalValue>(), new StdULogic1Value(0));
        _portIds.put(carry, _cnt++);
        _ports.add(carry);
    }
    public void fulladder_behav() {
        _event = new Event<DigitalValue>(_lTime, _lTime
            + getDeley(_portIds.get(sum), _portIds.get(sum)), _id,
            -1, _id, _portIds.get(sum), (
            a.getVal().xor(b.getVal())).xor(c.getVal()));
        _result.add(_event);
        // sum <= (a xor b) xor c ;
        _event = new Event<DigitalValue>(_lTime, _lTime
            + getDeley(_portIds.get(carry), _portIds.get(carry)), _id,
            -1, _id, _portIds.get(carry), (
            a.getVal().and(b.getVal())).or(
                c.getVal().and(a.getVal().xor(b.getVal()))));
            _result.add(_event);
            // carry <= (a and b) or (c and (a xor b));
    }
    public void combinationalLogic() {
        fulladder_behav();
    }
}
```

Слика 30: Пример Јава кода добијеног након превођења VHDL примера

Пресликавање овог реда захтева креирање нове класе која је по својој структури компонента. Да би се избегло да сва компоненте има комплетну дефиницију свих

помоћних метода и поља креирана је апстрактна класа дигиталних компонената DigitalComponentAbstract. Трансформисани ред гласи:

```
public class FULLADDER extends DigitalComponentAbstract
```

Приликом парсирања улазне датотеке наилази се на листу портова које дата компонента отвара за интеракцију са другим компонентама. Те портове треба трансформисати у одговарајуће Јава објекте исте намене. У примеру ред:

```
port (a, b, c: in std_logic; sum, carry: out std_logic);
```

треба трансформисати у одговарајући Јава код. Приликом трансформације наилази се на скуп улазних и излазних портова и њихове типове. Улазни портови су a, b, c сваки типа std_logic, док су излазни портови sum, carry такође типа std_logic. Ово значи да је сваки порт потребно креирати унутрашњи објекат порта одговарајућег типа. Тип std_logic се трансформише у StdULogiclValue Јава класу. Овом објекту је потребно поставити почетну вредност. Приликом трансформације узето је да сви објекти буду на почетку иницијализовани на почетну вредност логичне нуле, али је могуће поставити произвољну вредност. Декларација порта у језику VHDL одговара класи Port<DigitalValue>. Овде је стављено да портови примају искључиво дигиталне вредности иако SLEEP симулатор има могућност рада са догађајима произвољног типа, а самим тим и портовима који могу да примају те догађаје. На основу свих ових параметара се добија одговараћу Јава код.

```
Port<DigitalValue> a;
Port<DigitalValue> b;
Port<DigitalValue> c;
Port<DigitalValue> sum;
Port<DigitalValue> carry;
```

Иницијализација портова одговарајућим вредностима обавља се у конструктору класе public FULLADDER(). Иницијализација сваког од портова се обавља на следећи начин:

```
a = new Port<DigitalValue>(_cnt, "a", "a", 1, Port.IN,
        new SignalClass<DigitalValue>(), new
StdULogiclValue(0));
```

сходно правилима за иницијализацију портова. Када се заврши иницијализација портова онда се сваки порт ставља у одговарајуће интерне листе портова.

```
_portIds.put(a, _cnt++);
_portIds.add(a);
```

Објекат _portIds чува редни број порта на компоненти, док објекат _portIds чува који портови све постоје на компоненти. Није пожељно све ово чувати унутар

истог објекта јер ширина портова може да буде различита од 1.

Понашање компоненте писане користећи VHDL дефинисана је делом кода који означава архитектуру компоненте. У примеру се код:

```
architecture fulladder_behav of FULLADDER is
```

трансформише у позив методе истог имена у Јава на следећи начин:

```
public void fulladder_behav()
```

Акциони код који се налази унутар компоненте се трансформише у Јава код користећи особине дигиталних компоненте. Скуп операција над дигиталним вредностима су компатибилне са операцијама које подржава језик VHDL. У примеру израз

```
(a xor b) xor c ;
```

се трансформише у израз:

```
(a.getVal().xor(b.getVal())).xor(c.getVal())
```

Свака додела вредности сигналима или излазним портовим (**<=**) се реализује као догађај који је потребно обрадити у неком тренутку у будућности у зависности од кашњења кроз логичке компоненте. Уколико ово кашњење није специфицирано у исказу постављања сигнала на одговарајући вредност узима се подразумевана вредност пропагације до специфицираног порта. Уколико се ради о транспорту догађаја које је потребно проследити излазном порту као изворишни порт поставља се вредност -1. На основу овога изложеног иницијализација догађаја се обавља користећи следећи код:

```
_event = new Event<DigitalValue>(_lTime,
         _lTime + getDeley(_portIds.get(sum),
_portIds.get(sum)),
         _id, -1, _id, _portIds.get(sum),
         (a.getVal().xor(b.getVal())).xor(c.getVal()));
```

Када се креира догађај потребно је тај догађај убацити у бафер како би био обрађен у специфицираном временском тренутку:

```
_result.add(_event);
```

Извршавање ове методе се остварује тако што је у методу која описује понашање комбинационих мрежа убацимо позив методе за ову компоненту.

```
public void combinationalLogic() {
        fulladder_behav();
}
```

4.1.2 Извршни слој симулатора

Приликом реализације симулационог нивоа симулатора водило се рачуна о већем броју имплементација симулациони алгоритама. Код SLEEP симулатора су

доступна три симулациони алгоритма првог основног који ради у једној нити, и два који раду користећи више нити а притом примењујући конзервативни и оптимистички облик синхронизације [78]. Сви ови алгоритми се извршавају независно од тога да ли су покренути на једном рачунару или на више њих. Ово је постигнуто користећи посебан симулациони слој који крије место на коме се компоненте налазе.

Извршење сваког алгоритама извршног слоја се састоји од иницијализације и петље у којој се ради обрада догађаја. Приликом инцијализације учитавају се све компоненте логичког слоја и припремају се за извршење. Да би извршавање могло да започне све логичке компоненте имају листу иницијалних догађаја коју креирају приликом иницијализације. Ова листа се користи како би симулација могла да крене са радом. Обрада догађаја се састоји од извршењу специфициране акције над компонентом логичког слоја чија је идентификација саставни део догађаја који се обрађује. Обрада једног догађаја је приказана на слици 31. На почетку се прво дохвата компонента која треба да изврши обраду догађаја, након чега јој се прослеђује догађај који треба да обради. Када заврши обраду догађаја компоненте враћа листу нових догађаја које је потребно оградити у неком тренутку у будућности. Пошто компонента није свесна свог окружења потребно је извршити одређивање идентификатора одредишних компонената. Та трансформација се обавља користећи методу transform која постоји у класи која описује листу повезаних компоненти netlist. Након обављене трансформације догађаји се убацују у бафер у коме се скупљају сви догађаји. Овај бафер је саставни део симулационог слоја симулатора тако да се убацивањем поруке у овај бафер у зависности од одредишта те поруке она или задржава на истом рачунару или се транспортује на други. Све ово је потпуно сакривено од корисника. У случају оптимистичког алгоритма симулације ова метода је на полиморфан начин замењена тако да се појављује још један позив током кога се у листу догађаја које је креирала дата компонента убацује и управо креирана компонента.

```
public void work(Event<V> event) {
      SimComponentLogic<V> comp = netlist.getComponent(event.getDstID());
      List<Event<V>> events = comp.execute(event);
      queue.putEvents(netlist.transform(events));
      pastCreatedEvent.addAll(0, events);
}
```

Слика 31: Обрада догађаја

Основни симулациони алгоритам

Основни симулациони алгоритам има две фазе: фазе иницијализациону у цикличну обраде догађаја, што је приказано на слици 32. Иницијализација се обавља само једим приликом покретања система убацивањем иницијалне листе догађаја у бафер/приоритетни ред queue док се петља извршава све док услов завршетка петље не постане испуњен. Сваке итерације петље почиње налажење догађаја из приоритетног реда queue који има најмање логичко време када треба да буде извршен. Бафер садржи догађаје сортиране по логичком времену у коме треба да се обраде, али и према редоследу који компонента заузима у графу обиласка за случај да постоји већи број догађаја који имају исти временски тренутак за извршавање. Оно што је потребно обезбедити приликом пројектовања дигиталног уређаја је стабилност а то значи да резултат не треба да зависи од редоследа обраде догађаја који су заказани за исти временски тренутак. Овде се ради о полиморфном позиву објекта симулационог слоја који водо рачуна о локацијама на којима се догађаји налазе. Метода дохватања догађаја се у свим случајевима обавља над локалним бафером, а не над дистрибуираним. Након провере и чувања тренутног тренутка логичке симулације прелази се на претходно описану обраду догађаја. Овај алгоритам би могао да се делимично усложњи уколико би се извршни слој симулатора користио и за пренос порука намењених и другим слојевима. У том случају би требао непосредно пред обраду догађаја ставити услов да ли је догађај намењен логичком слоју (**if** (lastMsg.ok())).

```
public void run() {
  init();
  while (!end) {
    lastMsg = queue.getMsg();
    localTime = lastMsg.getEnd();
    work(lastMsg);
  }
}
```

Слика 32: Основни алгоритам симулације

Конзервативни алгоритам конкурентне симулације

Симулатор SLEEP такође подржава извршавање користећи већи број нити и конзервативни алгоритам извршавања. Основе овог алгоритма су приказане на слици 33. И овај симулациони алгоритам има две фазе: иницијализациону и фазу обраде догађаја. Иницијализација се обавља приликом покретања убацивањем

иницијалне листе догађаја бафер догађаја. Овде се ради о полиморфном позиву објекта симулационог слоја који води рачуна о локацијама на којима се компоненте којима су догађају упућени налазе. Метода убацује догађаје у одговарајући локални бафер. Уколико се ради о конкурентном решењу са дељеном меморијом онда се догађај убацује у објекат бафера који је на истом рачунару користећи синхронизовану методу за убацивање. А уколико се ради о дистрибуираној варијанти решења транспортује поруку на одговарађу радну станицу. Сваке итерације петље почиње налажење догађаја из приоритетног реда queue који има најмање логичко време када треба да буде извршен. Уколико је то логичко време веће од временског тренутка на који симулација сме да пређе, тај догађај се враћа у бафер и обавља се синхронизација са преосталим радним станицама. На следећи корак сме да се пређе само уколико све радне станице стигну на исто синхронизационо место и исто логичко време. Овде трба водити рачуна да се компоненте логичког слоја поделе по радним станицама на такав начин да време извршења за сваку групу буде приближно исто. Групе се користе за паралелизацију петљи, са коришћењем синхронизације на баријери на почетку сваке итерације. Алгоритам је заснива на другу генерацију конзервативних алгоритама за синхронизацију [91].

```
public void run() {
  init();
  while (!end) {
    Message m = queue.getMsg();
    if (!isTimeInTheRange(m)) {
      queue.putMsg(m);
      synchronize();
      m = queue.getMsg();
    }
    lastMsg = m;
    localTime = lastMsg.getEnd();
    work(lastMsg);
  }
}
```

Слика 33: Конзервативни алгоритам конкурентне симулације

Оптимистични алгоритам конкурентне симулације

Уколико се посматра време које алгоритми проведу у синхронизацији и вероватноћу да је стварно потребно чекати неки догађај, а не онај који је иницијалну узет из бафера може се закључити да овако строга синхронизација

није неопходна. Зависност између догађаја које компоненте логичког слоја креирају не постоји у свакој итерације, а самим тим синхронизација се може обавити користећи оптимистички алгоритам синхронизације. У зависности од вероватноће да се прихваћена порука разликују SLEEP симулатор може користећи оптимистички алгоритам да побољша перформансе извршавања. Ово је остварено уклањањем непотребне синхронизацију из претходног алгоритма користећи алгоритам приказан на слици 34. Овај нови алгоритам омогућава да нит која обављен израчунавања крене унапред у односу на друге нити под претпоставком да неће добити никакву синхронизацију поруке које се односе на тај период који је прескочен. Ако синхронизациона порука којим случајем стигне симулација мора да се врати на тренутак непосредно пред примљену поруку и да настави даље са синхронизацијом. Коришћење оптимисти приступ захтева неке промене у логичком слоју симулатора у циљу подршке поништавању ефеката пребрзог кретања и поновне обраде истог догађаја. Због постојања анти порука код ове реализације симулационог алгоритма неопходно је постојање реда **if** (lastMsg.ok()) који проверава да ли се ради о оригиналној поруци или о анти поруци којом се поништава дејство неке претходне поруке.

```
public void run() {
  init();
  while (!end) {
    Message m = queue.getMsg();
    if(localTime > m.getEnd()){
      restart(m.getEnd());
      continue;
    }
    lastMsg = m;
    localTime = lastMsg.getEnd();
    if (lastMsg.ok()) {
      work(lastMsg);
      pastMsg.putMsg(lastMsg);
    }
  }
}
```

Слика 34: Оптимистични алгоритам конкурентне симулације

Промене су у складу са Master/Worker парадигмом, али без софистицираних оптимизација [92]. Ово је учињено због чињенице да би имплементација симулатора требало да се користи у оквиру курса на основним студијама.

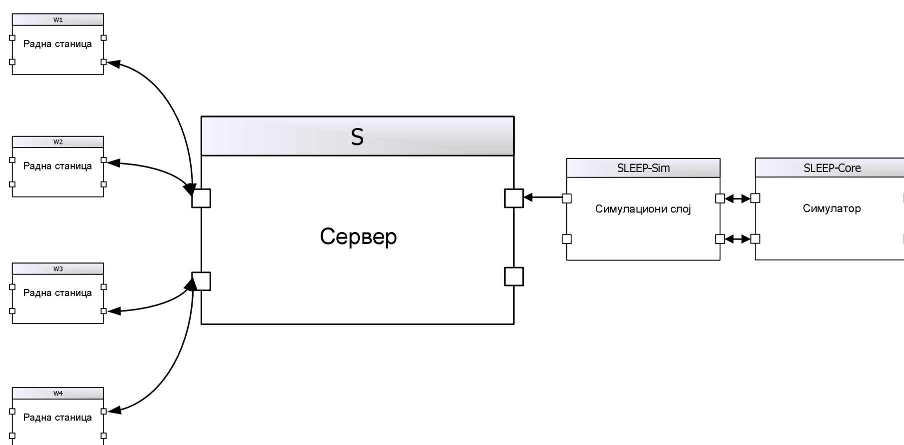
4.1.3 Симулациони слој симулатора

У овој секцији су представљени имплементациони детаљи симулационог слоја SLEEP симулатора. На почетку је дат опис архитектуре предложеног дистрибуираног система као и комуникација између учесника на глобалном нивоу. Након тога је дат опис реализације клијентске, серверске и стране радне станице као и опис интеракције између ових учесника. На крају је дат пример имплементације симулационог слоја користећи мобилне уређаје као радне станице, као и специфичности које су везане за овај тип уређаја.

4.1.3.1 Архитектура дистрибуираног система

Да би се обезбедио рад у дистрибуираном окружењу потребно је развити известан скуп класа и протокола који обезбеђују тај рад. У систему постоји три типа програма чија је архитектура дата на слици 35:

1. Централни сервер који служи за пријем и праћење рада извршавања симулације, чување информација о доступним чворовима у мрежи и могућност поновног стартовања појединих симулација.
2. Радна станица која служи за извршавање симулација које добија од централног сервера као листу логичких компонента и веза између њих. Радна станица, током рада, размењује поруке које су јој упутиле друге радне станице или централни сервер.
3. SLEEP симулатор који задаје листу логичких компонената које је потребно симулирати као, везе између тих логичких компонената, као и потребних параметара везаних за симулацију.



Слика 35: Архитектура дистрибуираног система

Процес започиње тако што кориснички програм, SLEEP симулатор, задаје аргументе логичке шеме које је потребно симулирати. Шема се састоји из листе логичких компонената, веза која описују како су повезани излази једних компонената са улазима дугих компонената, тип симулације коју треба да покрене, као и крајњег, логичког, времена које симулација треба да достигне. Уколико се тај посао први пут шаље онда се комплетан код компонената и веза прослеђује серверу. Након тога SLEEP симулатор контактира централни сервер коме прослеђује, као један посао, тај скуп компонената, листу веза и тип симулације како би их симулирао. Када централни сервер прими посао и његове параметре, он дели тај посао на већи број мањих послова које ће проследити радним станицама. Сервер на основу расположивог броја радних станица дели примљени посао на известан број мањих послове тако да сваки од њих буде приближно исте сложености. Сложеност посла се процењује на основу броја компонената, просечног времена потребног да се изврши симулација појединe компоненте, као и броја веза које један посао има са другим пословима. Када радна станица прими посао, креира нит намењену извршавању тог посла, и у тој нити започиње са његовим извршавањем на основу примљених параметара. Посао се извршава на радној станици тако што покренута нит радне станице прво креира одговарајући тип симулатора, а онда позива методу simulate() инстанце класе за симулацију дискретних догађаја (инстанце класе изведене из апстрактне класе Simulator). Да би се обављало паралелно процесирање потребно је прво иницијализовати ову инстанцу одговарајућим бафером (потребно је имплементирати интерфејс SimBuffer) преко кога се обавља комуникација са осталим рачунарима који обављају симулацију. Поред тога потребно је иницијализовати и симулатор компонентама и визама које ће обављати симулацију позивањем методе, метода setNetlist(Netlist<V> netlist). Компоненте и везе се иницијализују користећи посао који је сервер креирао на основу аргумента које је корисник задао. Симулатор обавља обраду све док логичко време симулације не достигне задату вредност на свим симулаторима који обављају паралелно израчунавање. Компоненте креирају листу порука (догађаја), за себе или за друге симулаторе (рачунаре), позивајући методу List<Event<V>> execute(Event<V> msg). Поруке се транспортују до симулатора користећи класу

која имплементира интерфејс SimBuffer. Радна станица одлучује коме да проследи поруку на основу листе доступних радих станица и листе које каже који порт које логичке компоненте је повезан са којим портом које друге логичке компоненте коју је задао корисник. Када се заврши симулацију радна станица прикупља стања свих логичких компонената позивом методе getState() класе Netlist и прослеђује их централном серверу који та стања касније треба да обједини и да их проследи клијенту.

Да би се обезбедило прикупљање информација о активним радним станицама централни сервер на сваких x секунди проверава да ли је нека радна станица исправна. Уколико сервер утврди да нека радна станица која је до тог тренутка обављала симулацију није више исправна, прекида целокупно извршавање дате симулације и покреће је поново на преосталим радним станицама. Након слања захтева за обраду кориснички програм не раскине везу са централним сервером већ је све време држи отвореном и периодично тражи статус обраде симулације. Централни сервер је реализован тако да може да обезбеди да у паралели прима већи број послова које је потребно обрадити. Кориснички програм, SLEEP симулатор, може од централног сервера да тражи информације о статусу посла, а може да тражи и резултате. Одмах по стартовању радне станице шаљу централном серверу информацију о томе да су стартоване и број послова које могу у паралели да обрађују. Број послова које радне станице могу да приме је аргумент који им се поставља приликом покретања. Конфигурација је тако постављена да комплетна обрада може да се реализује чак и ако постоји само једна радна станица. Уколико постоји само једна станица време извршавања је сигурно веће него у случају да се комплетна обрада обавља само на клијентском рачунару.

Параметри које SLEEP задаје серверу су: листа логичких компонената које је потребно симулирати, листу веза која каже који излазни порт које компоненте је повезан са којим улазним портом које компоненте, као и саме логичке компоненте (class датотеке) које се прослеђују серверу. Ови параметри се налазе унутар неколико датотека које се прослеђују серверу. Када се заврши обрада догађаја SLEEP симулатор добија резултат симулације који онда приказује на презентационом делу симулатора. Иницијализација компонената се постиже

позивом методе public void addComponent(String[] data), док се додавање веза постиже позивом методе public void addConnection(String[][] data).

Централни сервер у лог уписује време када је пристигао сваки посао, број под којим је посао сачуван, име рачунара коме је посао прослеђен, време када је посао завршен и његов тренутни статус. Статус посла може да буде: Ready – приспео на сервер али није никоме прослеђен, Scheduled – тренутно се прослеђује радној станици, Running – извршавање је у току, Done – посао се успешно извршио, Failed – посао није могао да се изврши (емитовао је неки изузетак), Aborted – корисник је одустао од извршавања посла.

Рад у дистрибуираном окружењу је сходно претходно описаној структури подељен један основни пакет који садржи скуп основних класа и интерфејса неопходних за успостављање комуникације између чворова који треба да интерагују и три помоћна пакета у којима се налазе конкретне имплементације протокола за сваку од страна које треба да интерагују. Први пакет чине класе које треба да омогуће комуникацију клијентске апликације, која је у увом случају SLEEP симулатор, са остатком система. Други пакет чине класе које треба да омогуће серверске апликације са клијентском апликацијом и са радним станицама, као и праћење протока времена. Трећи пакет чине класе које треба да омогуће извршавање симулације на радним станицама и комуникацију радних станица са сервером.

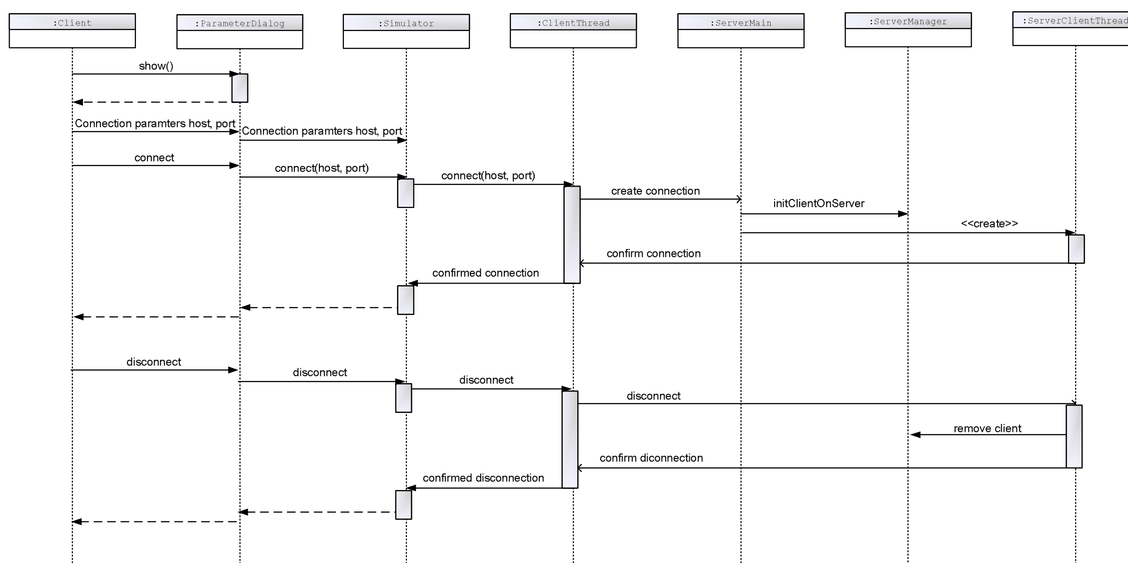
4.1.3.2 Клијентска страна

Први од пакета који омогућавају рад у дистрибуираном окружењу је пакет client, преко кога се задају основни комуникациони параметри клијентске стране апликације. Основна класа овог пакета је класа Clien која представља спој између централног дела симулатора, симулатора у ужем смислу, и дистрибуираног окружења, симулатор у ширем смислу. Ова класа је задужена да преко постављеног порта и адресе серверског рачунара успостави комуникацију са централним сервером. Поред овога класа је задужена и за раскидање комуникације између клијента и сервера, као и за преношење порука које је серверска страна апликације упутила централном делу симулатора. Да би комуникација успешно могла да се обавља потребно је да класа која води рачуна иницијализује листу послова, који су већ послати серверу на извршавање. Ово је

неопходно јер се један исти централни сервер може користити за већи број паралелних симулација које су могли да покрену различити клијенти у различито време. Уколико је иницијализовано приказивање напретка симулације води рачуна и о ажурирању статуса унутар листе покренутих послова. Уколико приликом извршавања дође до неких проблема, или се статус неког од послова промени због недостатка ресурса на којима се извршавање може обавити онда приказује одговарајући дијалог кориснику да он одлучи о даљем току симулације. Уколико корисник у неком тренутку жели да прекине конекцију или одустане од започетог посла са сервером постоји метода која то треба да омогући. Прекид конекције се користи у случајевима када обрада траје превише дуго и корисник је одустао од великог степена интеракције са симулатором и симулација је покренута у пакетској обради. Најважнија функција коју клијентски део дистрибуиране комуникације обавља је да прослеђује посао са задатим датотекама које садрже описе компонента, називом симулатора и логичким временом трајања симулације. У овој методи се врши и делимична провера исправности задатих конфигурационих датотека. Једна од најважнијих функција коју сервер треба да обезбеди, поред слања симулације на дистрибуирану обраду, је и дохватање резултата симулације коју је задао.

У пакету clinet се поред класе која садржи логику шта клијентска апликација треба да ради постоји и известан скуп класа које имплементирирају комуникационе интерфејсе из rs.ac.bg.etf.zaki.sleep.simulation.communication.protocol paketa. Основна класа клијентског протокола је ClientConnection која представља клијентску конекцију са сервером и имплементира потребне методе за слање и примање порука. Поред тога ова класа се извршава у посебној нити, што је обезбеђено наслеђивањем класе Thread и имплементирањем run методе унутар које је реализован пријем порука са сервера. Када се прими порука потребно је на основу протокола у зависности од типа поруке предузети различиту акцију, и зато се као први корак у комуникацију обавља препознавање и тријажа порука према типу. Ова класа у свом протоколу садржи основне методе које проверавају да ли комуникација успешно обављена и да ли се успешно завршила, уколико у било ком тренутку комуникације дође или до раскидања или до прекида комуникације из било ког

разлога обавља се обавештавање корисничког дела апликације емитовање изузетка одговарајућег типа. У комуникационом протоколу који клијент и север обављају као обавезан корак имају размену идентификатора како би се на јединствен начин идентификовала конекција клијента и сервера. Оно што је потребно обезбедити је да уколико из било ког разлога престане рад клијент и сервера овај идентификатор буде јединствен како се не би догодила ситуација да клијент и сервер почну да користе идентификациони број који није исправан. Комплетна описана комуникација између клијентског дела апликације, односно симулатора у ужем смислу, и централног сервера приликом успостављања и раскидања конекције је приказана на слици 36.

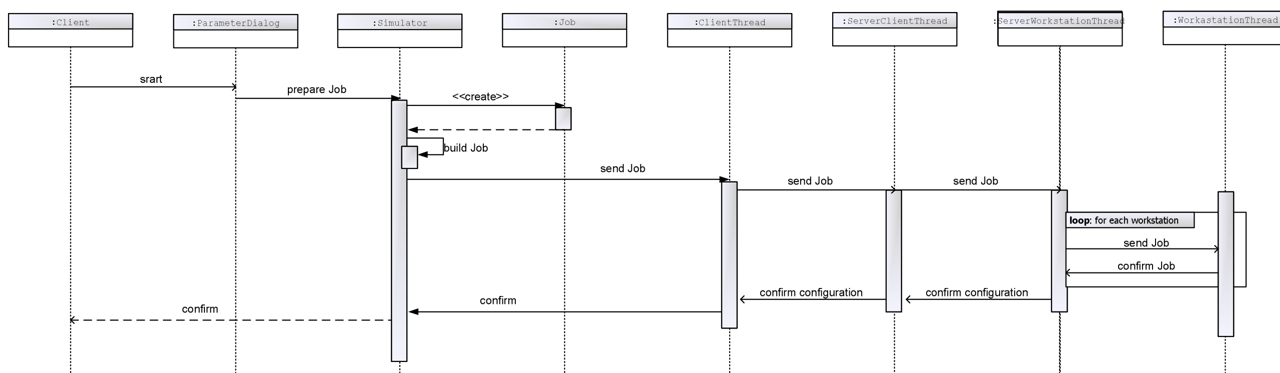


Слика 36: Дијаграм интеракције клијента и сервера приликом започињања комуникације

Приликом комуникације клијентског дела програма и серверска стране обавља се комуникација користећи објектне токове података. Комплетна комуникација може да се обавља или користећи текстуалне токове података или објектне токове података. У случају ове класе комуникација се комплетно обавља користећи објектне токове података, тако да су све класе које се користе у комуникацији обавезне да имплементирају интерфејс Serializable како би се омогућила комуникација користећи ObjectOutputStream и ObjectInputStream.

Следећи корак у комуникацији клијента и сервера је да клијентска

апликација серверу пошаље посао који је потребно обрадити. Посао се састоји из скупа класа и конекција које постоје између тих класа које је потребно успоставити. Класа која преноси информације између клијента и сервера је класа Job. Ова класа садржи основне информације о послу који је потребно обавити на серверској страни, листу омотача порука, низ бајтова са информацијама, листе текстуалних информација о послу, као и статусу посла. Сам посао, у основној варијанти протокола, не садржи информације о томе на који начин би сервер требало да посао раздвоји на већи број радних станица. Овај посебни додатак протоколу садржи и информације о захтеваном размештају компонената по доступним или захтеваним радним станицама у зависности од времена извршавања појединих операција, и временског кашњења између појединих компонената система. Како посебна грана у протоколу стоје методе за обраду ситуација када се захтева статус посла. Статус се може захтевати или аутоматски након истека одговарајућег временског интервала или мануелно интеракцијом са стране централног дела симулатора. Пошто се овде ради о симулаторима са великом интеракцијом постоје и методе које у било ком тренутку од сервера могу да захтева асинхроне операције захтева за прекида симулације. Ови захтеви могу да буду последица ситуације да корисник одустаје од симулације како би се вратио на неки претходни тренутак у извршавању или да би променио неки од аргумената симулације и наставио симулацију са тако промењеним аргументима. Комуникација између клијента и сервера приликом прослеђивања посла серверу и дохватања статуса и резултата посла од сервера је приказана на следећем дијаграму интеракције.



Слика 37: Дијаграм интеракције клијента и сервера приликом слања посла ради започињања симулације

4.1.3.3 Серверска страна

Следећи пакета који омогућавају рад у дистрибуираном окружењу је пакет server, преко кога се задају основни комуникационо параметри серверске стране апликације. Основна класа која треба омогући рад серверске стране апликације је класа Server. Ова класа се извршава у посебном току контроле јер наслеђује класу Thread и имплементира методу run у оквиру које врши расподела послова на радне станице и прихватање резултата обраде. Поред овога класа је задужена и за раскидање комуникације између сервера, клијента и радних станица, као и за преношење порука које је серверска страна апликације упутила централном делу симулатора или радним станицама. Ова класа ради послове који се односе на пријем захтева клијентског дела апликације, пријем резултата обаде радних станица, тражење резултата које клијент обавља, укључивање и искључивање појединих радних станица, вођење евиденције о половима који се тренутно извршавају, о месту где се ти послови извршавају, као и о њиховом статусу. Ова класа представља пословну логику комплетног серверског дела апликације.

У пакету server се поред класе која садржи логику шта клијентска апликација треба да ради постоји и известан скуп класа које имплементирају комуникационе интерфејсе из rs.ac.bg.etf.zaki.sleep.simulation.communication.protocol paketa. Основна класа серверског протокола је ServerConnection која представља серверску везу са клијентском страном и страном радних станица, и имплементира потребне методе за слање и примање порука. Комуникација која ова класа остварују са осталим деловима симулатора се извршава у посебној нити, што је обезбеђено наслеђивањем класе Thread и имплементирањем run методе унутар које је реализован пријем порука са сервера. Када се прими порука потребно је на основу протокола у зависности од типа поруке предузети различиту акцију, и зато се као први корак у комуникацију обавља препознавање и тријажа порука према типу. Ова класа у свом протоколу садржи основне методе које проверавају да ли комуникација успешно обављена и да ли се успешно завршила, уколико у било ком тренутку комуникације дође или до грешке или до прекида комуникације из било ког разлога обавља се обавештавање корисничког дела апликације емитовање изузетка одговарајућег типа. Комплетна обрада захтева се обавља

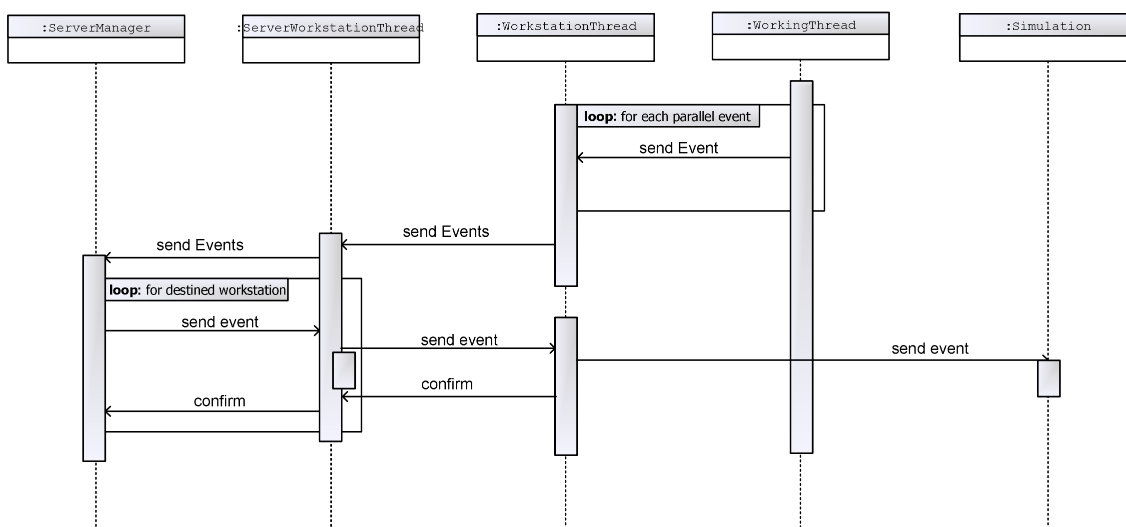
унутар методе processRequest у којој се полиморфно прави инстанца класе која треба да ради обраду захтева који поједине стране упућују.

Прва класа која полиморфно обавља комуникацију између клијента и сервера је класа ClientServer. Ова класа име методу processRequest која позива методе класе Sever за обраду захтеве, за пријем нових послова и да врши координацију између нити која обављају слање и пријем порука и послова измешу сервера и клијената. Ова класа обезбеђује да приликом пријема новог посла буде позвана метода која примљени посао ставља у листу нових послова и ажурира свој статус. Приликом провере статуса посла прво се проверава ова листа послова, уколико посао не постоји у овој листи клијенту се одговара да посао не постоји.

Следећа класа овог пакета која полиморфно обавља комуникацију између радне станице и сервера је класа WorkStationServer. Ова класа име методу processRequest која позива методе одговорне за комуникацију са једном радном станицом. Унутар ове методе се обавља обрада захтева, креирање нових послова и врши координацију између нити која шаље и нити која прима поруке. Пошто је комуникација између раде станице и сервера асинхрона, двосмерна и блокирајућа па се она обавља користећи две нити. Прва нит служи за неблокирајуће слање порука/догађаја радној станици. Ова нит поруке чита из неблокирајућег бафера и прослеђује их радној станици. Друга нит је задужена за пријем порука које се затим обрађују и прослеђују следећим актерима у комуникацији. Ове нити позивом одговарајуће серверске методе симулацију деле на основу броја радних станица које имају слободну бар једну нит и на основу броја компоненти које симулација садржи. Ова подела се обавља на основу број компонената, броја веза и расположивих компонената. Постоји и посебна класа која посао дели на основу времена обраде појединих компонената, ли и на основу расподели коју је клијент иницирао.

Када све радне станице успешно стигну до специфицираног симулационог логичког времена резултати симулација се шаљу серверу. Уколико се симулација извршава на више радних станица, када последња радна станица заврши симулацију свог дела посла, осталим радним станицама на којима се симулација извршавала шаље се синхронизациона порука која их обавештава да серверу проследе резултате симулације. Када се симулација заврши сервер почиње са

пријемом резултата од радних станица и припремом коначног одговора који ће бити прослеђен клијентском делу апликације како би био приказан као резултата. Када сервер прихвати све делове симулације од ангажованих радних станица, ажурира статус посла и припрема се да пошаље резултат. Радне станице не гасе своје нити јер постоји извесна вероватноћа да ће клијент захтевати наставак симулације са истим параметрима. На слици 38 је приказан дијаграм интеракције приликом прослеђивања догађаја који креира једна радна станица, а који је намењен другој радној станици да обави обраду. Пре него што сервер проследи догађај другој радној станици сервер проверава да ли посао у међувремену није прекинут. Уколико посао није прекинут јавља радној станици да прекине дању обраду текуће симулације



Слика 38: Дијаграм интеракције сервера и радне станице приликом прослеђивања догађаја са једне радне станице на другу радно станицу

Поред нити које врше обраду захтева на серверској страни постоји известан број нити које обављају помоћне активности које омогућавају функционисање остатка система. Један од таквих нити служи за одређивање броја доступних радних станица и на њима доступних нити које могу да раде обраду. Ова нит периодично прозива пријављене радне станице и тражи потврду о броју расположивих нити и активних нити, као и информације о томе да ли у претходном временском периоду долазило до прекида везе. Уколико је било неких проблема такве радне станице се избацуј из листе доступних радних станица и њима се не додељују нови послови.

4.1.3.4 Страна радне станице

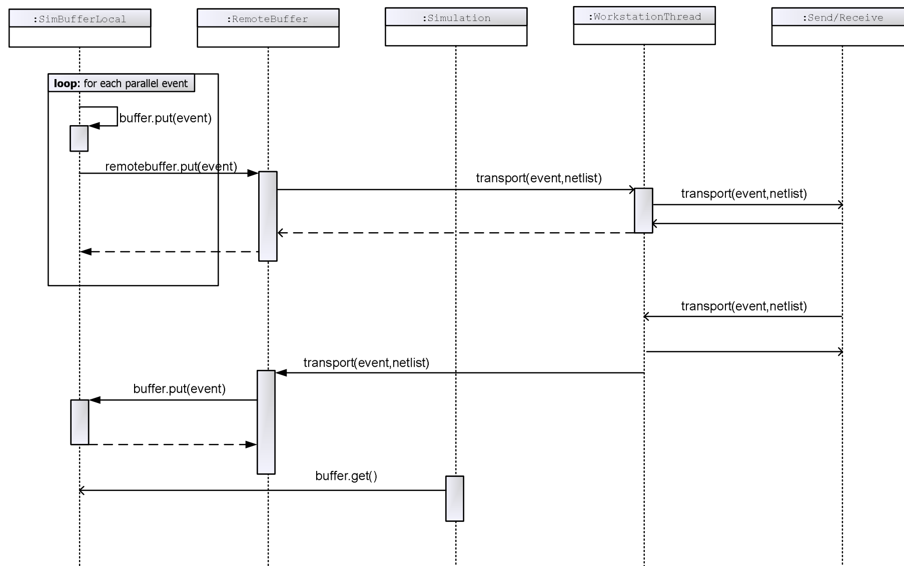
Следећи пакета који омогућавају рад у дистрибуираном окружењу је пакет worker, преко кога се задају основни комуникационо параметри радне станице апликације. Основна класа која треба омогући рад радне станице је класа Worker. Ова класа се извршава у посебном току контроле јер наслеђује класу Thread и имплементира методу run у оквиру које врши обрада послова и припремање резултата за слање серверу. Ова нит сама не обавља обраду послова већ приликом свог креирања формира групу нити које ће обављати обраду. Овој је неопходно како би се на клијентски захтев одговорило што брже. Ова група нити се може динамички повећавати како би се обрадили сви захтеви. Ова нит је задужена и за обраду захтева које је упутио сервер. Ови захтеви се огледају у тражењу статусу посла који се извршавају на датој радној станици, прикупља резултате за одређени посао, отказивање извршавања посла, ажурирање листе активних послова и пратећих датотека и класа.

У пакету worker се поред класе која садржи логику шта радна станица треба да ради постоји и известан скуп класа које имплементирају комуникационе интерфејсе из rs.ac.bg.etf.zaki.sleep.simulation.communication.protocol пакета. Основна класа протокола радне станице је WorkerConnection која остварује конекцију са сервером, и имплементира потребне методе за слање и примање порука. Комуникацију коју ова класа обавља се остварује у посебној нити, што је обезбеђено наслеђивањем класе Thread и имплементирањем run методе унутар које је реализован пријем порука са сервера и других радних станица. На исти начин како и класе које имплементирају протокол у клијентском и серверском пакету обавља препознавање и тријажа порука према типу. На почетку свог рада радна станица позива методу за повезивање са сервером и том приликом размењује јединствен идентификатор који ће се користити приликом комуникације. Радна станица том приликом серверу шаље и величину групе активних нити које могу да раде обраду послова, као и порт преко кога се може остварити испитивање да ли је радна станица исправна као и преко кога се обавља комуникација између радних станица када оне размењују догађаје које производи симулација која се извршава на њеним нитима. Комуникација коју ова класа обавља са другим нитима се остварује користећи неблокирајући бафер.

Једна од основних интеракција коју радна станица обавља је пријем новог посла са сервера и његово покретање на некој од нити која је задужена за обраду. У тренутку када радна станица прими поруку која имплицира да се ради о послу који је потребно урадити радна станица обраду наставља позивом одговарајуће методе која прихвата посао који је потребно обрадити. Посао који је потребно обрадити се састоји из листе компонената, класа које одговарају тим компонентама, пратеће xml датотеке са детаљним описом унутрашње структуре дате компоненте, као и конекције које примљене компоненте имају да другим компонента, постоји и могућност слања локације где се налазе те компоненте како би се остварила размена поука између радних станица при раду симулатора, али се ова комуникација не обавља јер је узето решење где све поруке иду преко централног сервера, а не директно радна станица-радна станица. Посао се додељује некој од доступних нити. Уколико нема доступних нити посао се убацује у бафер како би га узела прва слободна нит. Када нит којој је додељен посао прими посао прво иницијализује компоненте, везе и направи листу спољашњих веза како би се дања комуникација успешно обавила. Након тога се покреће одговарајућа метода симулатора која даље обавља обраду.

Метода simulate обавља симулацију и води рачуна о протеклом логичком времену, и обрађеним догађајима, стањи у коме се налазе свака од компонената, као и о расположивим ресурсима. Интеракција коју нит на радној станици има са другим радним станицама приликом размене догађаја које ствара симулатор је приказана на слици 39. Приликом обраде догађаја догађај се смешта у класу која имплементира интерфејс SimBuffer. У овом случају се ради о имплементацији SimBufferRemote, у којој се на почетку прво проверава да ли је догађај намењен некој компоненти која се налази на истој радној станици, па уколико се налази порука се смешта у локални бафер SimBufferLocal, уколико дестилациона компонента није на истој радној станици порука се транспортује на централни сервер позивом одговарајуће методе. Овај догађај се прослеђује серверу који га прослеђује одговарајућој радној станици. Уколико би се направила директна веза између радних станица једина разлика би била да се промени део имплементације која би проверавала где се налази одредиште догађаја, па се тој радној станици прослеђује догађај. Када сервер упути догађај радној станици радна станица

прима догађаји који на крају завршава у баферу класе SimBufferLocal. Овде треба нагласити да које год да је имплементација протокола комуникације, или који тип симулатора да би симулација могла да функционише потребно је поставити да догађај који се обрађује на крају мора да се нађе у локалном баферу.



Слика 39: Дијаграм интеракције радних станица приликом слања догађаја на обраду

4.1.3.5 Рад на мобилним уређајима

Због своје слојевите архитектуре SLEEP симулатор се учинио погодан и за експериментисање са компонентама и начинима извршавања које нису стриктно везани за архитектуру и организацију рачунара нити на личне рачунаре и сервере. Пошто је симулатор пројектован на такав начин да интерни слојеви симулатора не буду свесни постојања дистрибуираног извршавања и да се увођењем слојева постиже утисак да се све компоненте налазе на једном рачунару. Постојање симулационог нивоа је од великог значаја за пројектовање симулатора са дистрибуираном обрадом јер омогућава логичку поделу модела на већи број под модела. Симулациони ниво управо представља расподелу логичких компонената на различите рачунаре у циљу оптимизације извршавања. Управо оваква организација је овај симулатор учинила погодним за експеримент у коме се покушало пребацивање симулатора за рад у сасвим новом типу дистрибуираних окружења у окружењу састављеном од мобилних уређаја ([93] и [94]).

Ова секција има за циљ да представи искуства стечена приликом реализације симулационог нивоа код симулатора дискретних догађаја користећи мобилне уређаје. Испитивање могућности је обављено креирањем евалуационог прототипа. Коришћење мобилних уређаја за паралелно извршавање симулације је еквивалентно парадигми програмирања користећи Грид технологије (Grid Computing) [95]. Разлог за једним оваквим експериментом се заснова на томе да су мобилни уређаји веома заступљена и широко распрострањена категорија рачунара. Оно што мобилне уређаје чини интересантним када је у питању дистрибуирано извршавање симулације су следеће њихове карактеристике: Број доступних мобилних уређаја је упоредив ако не и већи од броја доступних личних рачунара на којима би симулације могле да се извршавају; Мобилни уређаји подржавају више телекомуникационих протокола (gprs, edge, wi-fi, 3g) помоћу којих је могућа комуникација са уређајем; Процесорска моћ данашњих мобилних уређаја је значајна и има тенденцију све бржег раста; Могућност извршавања делова Јава кода на мобилним уређајима.

Поред својих добрих страна апликације које раде на мобилним уређајима имају и своје недостатке. Ти недостаци се огледају у томе да, за разлику од личних рачунара, међу мобилним уређајима постоји превише различитости како по питању перформанси тако и по доступности расположивих ресурса. Приликом развоја овог модула ишло се са претпоставком да се жели остварити барем делимична компатибилност са постојећим решењима како по питању технологије тако и по питању архитектуре система. Како технологија је одабрана Јава прилагођена за мобилне уређаје, а како архитектуре иста она описана у поглављу 4.1.3 „Рад у дистрибуираном окружењу“.

Овако одабрана технологија даје комплетан скуп функционалности које нуди Јава МЕ, и могућност одабира већег броја уређаја на којима се апликација може извршавати али су те могућности у неким случајевима доста скромније у односу на решења које нуде произвођачи мобилних уређаја и пратећи оперативни системи у случају одабира конкретног мобилног уређаја. Овакав избор платформе је имао једно велико ограничење које се односило на немогућност додавања нове извршне целине у току извршавања апликације на мобилним уређајима без гашења и поновном укључивање апликације на уређају које је корисник морао да

одобри. Конкретно, апликација, мидлет, не може стартовати други мидлет из безбедоносних разлога. Из овог ограничења је проистекао захтев да се уместо динамичког учитавања Јава кода узме могућност да се класе компонената једном направе, статички учитају, а та компонента би онда интерпретирала примљени захтев који би пристизао у виду посебног скрипт језика.

4.1.3.5.1 Преглед скрипт језика

Приликом решавања овог проблема на почетку је урађен преглед доступних решења. Циљ прегледа је био да се одабере или модификује неки од постојећих сктипт језика. Пошто се у тренутку експеримента могло наћи више доступних језика направљена је мала анализа у којој су разматрани неки од најзначајнијих пројеката из ове области [96]-[101].

Језик CellularBASIC који је дериват QBASIC1.1 програмског језика [96]. Поседује преко деведесет кључних речи. Има подршку за рад са покретним зарезом, познате су му тригонометријске функције и генератор случајних бројева. Такође, има подршку и за рад са стринговима. Од посебних предности у односу на остале језике издваја се графички улаз/излаз као и улаз/излаз према датотекама. Ту су и елементарне команде за рад са графиком. Језик поседује и могућности креирања сокета, као и могућност слања СМС поруке из програма. Пласиран је под GPL лиценцом и евентуална надградња саме синтаксе би требало да буде удобна, с обзиром да су доступни lex и yacc датотеке. Недостатци овом језику су изузетно сиромашна финална испорука и то што није предвиђена библиотека како би се језик интегрисао у други пројекат. Такође, језик нема подршку за паралелизам.

Дериват FscriptME језика који је иницијално развијен за Јава CE платформу [97]. Значајно скромнији по карактеристикама од језика CellularBASIC. Међутим, за разлику од њега долази у облику библиотеке и интеграција у постојеће пројекте је веома једноставна. Карактерише га потпуно минималан сет функционалности. За било какву озбиљнију употребу неопходно га је надградити. Долази под GPL лиценцом. Крупан недостатак FscriptME језика је то да његов парсер и лексички анализатор није могуће генерисати услед тога што у пројекат нису укључене изворне датотеке које би то омогућили (lex и yacc).

Hecl је замишљен да буде допуна Јава МЕ. Карактерише га изузетно једноставна синтакса [100]. Оријентисан је ка томе да је могуће уз веома мало напора манипулисати графичким садржајима, контејнерима текста, лабелама итд. Проширивање језика се остварује наслеђивањем одговарајућих класа. Веома је добро подржан рад са датотекама. Зависно од тога на ком Јава МЕ профилу се језик налази, одређене екстензије могу изостати. Hecl долази под Apache 2.0 лиценцом.

Simkin скрип језик је један од пионирских подухвата на овом подручју [101]. Замишљен је да ради уз Јава МЕ или C++. Посебно је развијен Simkin Symbian OC као подршка за скриптинг под Symbian оперативним системом. Вођен је идејом да се крајњем кориснику остави могућност за додатну надградњу одређене апликације користећи скрипт језик. Симкин је под GNU LGPL лиценцом.

Приликом развоја компонената која могу да се извршавају на мобилним уређајима постоји потреба да се предвиди и могућност паралелног рада, као и да језик поседује и подршку за рад са низовима. С обзиром на усвојен концепт система, пожељно је да сам језик познаје паралелизам на неки начин, односно да има подршку за рад са паралелним блоковима.

Да би се омогућило писање скрипти које омогућавају извршавање унутар компоненти на мобилним уређајима одабран је и проширен FscriptME, али тако да скуп функционалност и даље буде минималан. Проширена верзија FScriptME језика названа је у овом случају FScriptME+. Приликом рада са скрипт језиком потребно је дефинисати скуп типова података, контролних структура, оператора и функција које ће бити доступне за те операције. Уколико се ради са оваквим скрипт језиком, ипак није подржано све оно што је реализовано за рад са дигиталним компонентама јер је се рад са мобилним уређајима посматра као експеримент унутар симулатора. Приликом рада са променљивама дозвољено је креирање простих типова података, као и низовских типова, а објектно оријентисани приступ није подржан. У Табели 12 је дат скуп функционалност које пружа представљени скрипт језик.

Табела 12: Карактеристике развијеног скрипт језика

Типови података	string, integer, double
Контрола тока	if/endif
Петље	while/endwhile
Функције	func/endfunc
Аритметичке операције	+, -, *, /
Логичке операције	ǁ, &&, !
Рад са низовима	array
Аутоматска конверзија типа	()
Паралели блок	parallel/endparallel
Улаз/излаз	Две предефинисане променљиве

Пошто је представљени скрипт језик доста једноставан њиме је доста тешко описати сложене дигиталне компоненте, али је могуће писање једноставних компонената, ради провере концепта. Један типичан пример развијеног скрипт језика је приказан на слици 40.

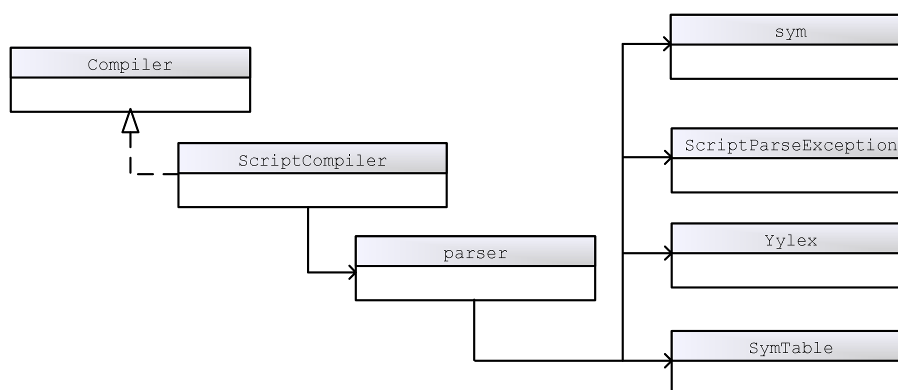
```
func test(int a)
      int c = 0
      if (a > 1)
            c = test(a-1)
      endif
      int s = a + c
      return s
endfunc
      int d = test(20)
return d
```

Слика 40: Пример дела компоненте дефинисане развијени скрипт језик

Представљени скрипт језик је тако реализован да омогући означавање блокова које је могуће извршавати у паралели. Оваква паралелизација је остварена увођењем кључних речи parallel и endparallel. Паралелни блок означава делове које је потребно парсирати у паралели. Ово је омогућено тако што се за сваку линију кода унутар паралелног блока креира по једна инстанца парсера која своје извршење наставља као засебна нит било на истом уређају, уколико је могуће креирати већи број нити, или као нови задатак на неком другом уређају. Да би се омогућило извршавање блокова паралелно на више рачунара, што одговара повезаним догађајима, било је потребно обезбедити серијализацију

инстанце класе парсера који цео свој контекст транспортује кроз мрежу. Синхронизација код овог скрипт језика постоји само на крају паралелног блока на коме посао чека све док се не обједине све нити.

Да би се омогућило извршавање симулације на мобилним уређајима, који тренутно из безбедносносних разлога не подржавају динамичко учитавање класа и компонента, морало се одустати од комплетне динамичке преносивости кода. У односу на приступ представљен у секцијама 4.1.1 „Логички слој симулатора“, 4.1.1.1 „Рад са дигиталним компонентама“ и 4.1.1.3 „Писање кода користећи VHDL“ корисник није развијао нове компоненте користећи Јава или VHDL језик за опис хардвера већ су компоненте биле предефинисане у програмском језику Јава и нису се могле мењати али су имале могућност да извршавају посебан скрип језик. Компоненте дефинисане користећи овај посебан скрипт језик се не трансформишу у компоненте писане у програмском језику Јава већ се као такве интерпретирају на мобилним уређајима и на тај начин избегава динамичко учитавање компонената. У поступку превођења и интерпретације кода писаног користећи овај посебан скрипт језик коришћене су класе дате у пакету murlen.util.fscriptME, као и стандардне библиотеке за превођење и парсирање кода JFlex и CUP. На слици 41 је дата део хијерархија класе које се користе за превођење кода писаног у користећи развијени скрипт.



Слика 41: Хијерархија класа са превођење скрипт језика

Основна класа која обавља синтаксну и семантичку анализу је класа Parser. Ова класа је креирана користећи алат CUP на основу задате спецификацијом креираног скрипт језика и правила задатих за његову интерпретацију задат FScriptMEPlus.cup датотеком. Алат CUP генерише Јава код LALR(1) парсера на основу спецификације у виду безконтекстне атрибутивно-транслационе

граматике. Овај Јава код представља предефинисану компоненту која се налази на мобилном уређају која је задужена за синтаксно-семантичка анализа, као и интерпретацију скрипт језика.

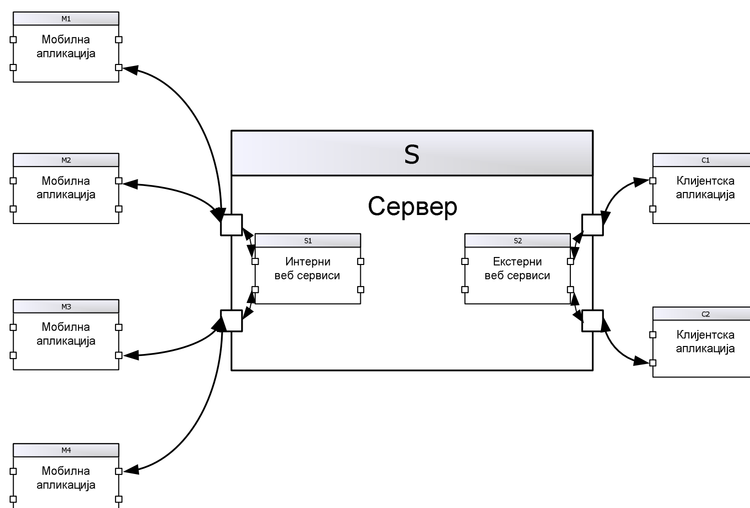
Приликом интерпретације скрипте, мобилна апликација може наићи на паралелни блок. У том случају, она преузима улогу апликације која поставља задатак на извршење и као задатак серверу шаље серијализовану Parser класу са њеним контекстом у том тренутку. Заједно са класом као део задатка се шаљу и редови које треба интерпретирати (одговарајуће линије паралелног блока). На овај начин је реализована да мобилна апликација може прихватити два типа задатака за извршавање: Један тип представља саму садржину скрипте и он се налази у текстуалном формату; Други тип представља серијализовани облик Parser класе и резултат је претходног интерпретирања паралелног блока на неком од мобилних уређаја

Прототип симулационог нивоа симулатора дискретних догађаја који се извршава на мобилним уређајима треба да се бави и да омогући прикупљање, распоређивање и извршавање послова водећи рачуна о комуникационим протоколима. Прототип решења за рад са мобилним уређајима је организован у слојевима тако да функцију прикупљања обавља клијентска апликација, функцију распоређивања обавља серверска апликација покренута на једном серверу, а функцију извршавања мобилна апликација покренута на већем броју мобилних уређаја.

4.1.3.5.2 Архитектура дистрибуираног система

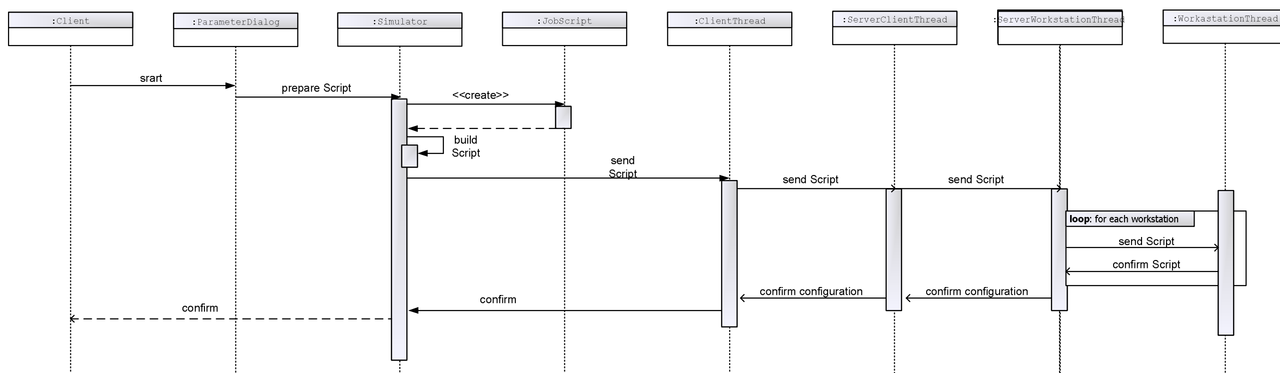
Рад у дистрибуираном окружењу на мобилним уређајима има сличну структуру оној претходно описаној у одељку 4.1.3 „Рад у дистрибуираном окружењу“ и чак задржава комплетни основни пакет који садржи скуп основних класа и интерфејса неопходних за успостављање комуникације између чворова који треба да интерагују и три помоћна пакета у којима се налазе конкретне имплементације протокола за сваку од страна које треба да интерагују и комуницирају. Пошто основни пакет симулатора омогућава потпуно транспарентно решење, потребно је направити извесне модификације у пакетима који обављају комуникацију између сервера и радних станица, док део који се односи на комуникацију између клијената и сервер не треба мењати. На слици 42

су представљена блок шема комуникације између актера у дистрибуираном окружењу заснованом на мобилним уређајима. Клијентска апликација има за циљ да за остатак евалуационог прототипа емулира симулациони ниво симулатора. Клијентска апликација је базирана на Јава СЕ, серверска апликација на Јава ЕЕ, а мобилна апликација на Јава МЕ технологији.



Слика 42: Блок шема система при раду са мобилним уређајима

Први од пакета који омогућавају рад у дистрибуираном окружењу је пакет client, преко кога се задају основни комуникациони параметри клијентске стране апликације. Основна класа овог пакета је класа Clien која представља спој између централног дела симулатора који у овом случају генерише скуп послова које је потребно обрадити. За разлику од уобичајеног рада апликације у овом случају сваки посао је представљен у виду скрипте тако да на сервер није потребно пребацивати класе које је потребно динамички учитати на радним станицама. Ова класа је задужена да преко постављеног порта и адресе серверског рачунара успостави комуникацију са централним сервером и да серверу проследи скрипте које је потребно извршити на радним станицама. Живот скрипте у систему започиње тиме што је клијентска апликација проследи серверу. Поред овога класа је задужена и за раскидање комуникације између клијента и сервера, као и за преношење порука које је серверска страна апликације упутила централном делу симулатора. Клијентска апликација позивајући одговарајуће функције сервер може добити листу регистрованих мобилних уређаја и поставити скрипту за извршење на конкретном уређају.



Слика 43: Дијаграм интеракције клијента и сервера приликом слања скрипте

Следећи пакета који омогућавају рад у дистрибуираном окружењу је пакет server, преко кога се задају основни комуникационо параметри серверске стране апликације. Сервер је задужен да прихвата скрипту и у форми посла је спрема како би је одређена радна станица на мобилном уређају преузео на извршавање. Приликом комуникације између сервера и клијента, на исти начин као и у случају дистрибуираног извршавања, сервер сваком задатку додељује јединствени број и саопштава га клијентској апликацији. На основу тог броја се клијентска апликација се позива на конкретан део симулације и може позивајући одговарајуће функције у сваком тренутку од сервера сазнати докле се стигло са извршењем, и за случај да је извршење завршено може од сервера затражити резултат.

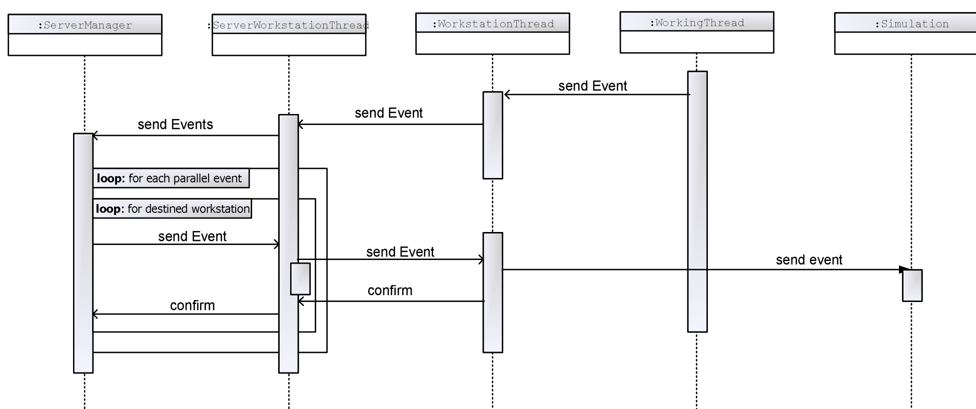
Класа која води рачуна о поменутој комуникацији између клијента и сервера у овом случају је иста као и пре, ClientServer, и њена имплементација се не разликује од имплементације претходне класе истог име само је захтевало додатно проширење комуникационог протокола. На претходној слици је приказана интеракција клијента и сервера приликом слања посла који је потребно извршити, што се у увом случају своди на слање скрипти које потребно извршити. Поред функција које су раније биле доступне клијенту тај скуп функционалности је у овом случају допуњен тако да је клијентској апликацији омогућено да шаље скрипте унутар специфичних компонената за извршавање. Поред овога омогућено је клијентској апликацији да има увид у статус постављеног посла, и за случај да је обрада завршена, да сазна резултат обраде. Функције које немају функцију у самом израчунавању али које се могу искористити за балансирање послова

обухватају методе за дохватање листе радних станица и њихових карактеристика како би сам клијент могао да обави поделу симулацију на већи бој делова. Додатне статистичке функције омогућавају клијенти да сазна колико је трајало извршавање појединих делова симулације на појединим радним станицама.

Следећа класа овог пакета која обавља комуникацију између радне станице и сервера је класа WorkStationServer. Пошто ова класа треба обезбеди комуникацију између уређаја у хетерогеном систему какав је систем састављен од мобилних уређаја искоришћен је апстрактни модел комуникације. Један такав једноставан и широко распрострањен вид комуникације је механизам веб сервиса. Модел комуникације који је потребно реализовати приликом комуникације са мобилним уређајима подсећа на решења која се користе за рад у мрежном грид окружењу а који су заснована управо на интензивном коришћењу веб сервиса. Код овог типа решења постоје и разрађени механизми за пренос података између клијентске и серверске стране тако да у овом случају није неопходно имплементирати специфичне класе као што је било потребно у случају дистрибуираног извршавања. Поред тога овакво решење је веома једноставна каснија надоградња система.

Серверска апликација поседује веб сервис чије су функције намењена мобилној апликацији. Сходно организацији система веб сервис намењен мобилној страни назван интерни веб сервис. Интерни веб сервис стоји на располагању мобилним уређајима како би кроз његове функције могли да се региструју на систем, да преузимају задатке за извршавање, да дају на извршење делове својих задатака другим уређајима, и да на крају објаве резултат обраде. Поред овога класа веб сервиса је задужена за успостављање и раскидање комуникације између сервера и радних станица, као и за преношење послова које је серверска страна централни део апликације упутио радним станицама на мобилним уређајима. Приликом прослеђивања скрипти са сервера на радне станице тим станицама је омогућено да дохватају и шаљу идентификатор посла, само посао односно скрипт који је потребно извршити, као и остале карактеристике посла. Када све радне станице успешно стигну до специфицираног симулационог логичког времена резултати симулација се шаљу серверу позивањем одговарајућих метода. Пошто на мобилним уређајима није било могуће чувати довољно контекста што захтева

оптимистички алгоритам симулације, мобилну уређаји су се користили само у ситуацијама када није било рестарта симулације, а то је било у случају конзервативног приступа алгоритму симулације. Након завршене обраде мобилни уређају прослеђују резултат обраде, али могу да проследе и нове послове, односно скрипте које је потребно негде дистрибуирано извршити. Ова класа веб сервиса је задужена за пословну логику серверског дела апликације. Пошто се овде ради о у принципу спорим уређајима којима је потребно доста времена да приме скрипт на обраду, скрипте се углавном праве тако да могу да се извршавају већ број пута само добијају различите улазне параметре. Ово је обезбеђено тиме да радне станице не прекида своје извршавање и тражи нови посао већ проверава да ли за исти посао постоји нови аргументи јер постоји извесна вероватноћа да ће се захтевати наставак симулације са новим параметрима. На слици 44 је приказан дијаграм интеракције приликом прослеђивања посла који креира једна радна станица, а који је намењен другој радној станици да обави обраду. Пре него што сервер проследи догађај другој радној станици сервер проверава да ли посао у међувремену није отказа.



Слика 44: Дијаграм интеракције сервера и радне станице приликом прослеђивања догађаја са једног мобилног уређаја на други мобилни уређај

Следећи пакета који омогућавају рад у дистрибуираном окружењу на мобилним уређајима је пакет worker, преко кога се задају основни комуникационо параметри мобилне апликације. Основна функција ове класе је да води рачуна о животном циклусу мобилне апликације. Рад мобилне апликације започиње позивом одговарајуће функције веб сервиса на серверу. Овом приликом мобилну

уређај од сервера тражи да том мобилном уређају, односно његовој апликацији додели јединствени идентификациони број. Сервер мобилној апликацији враћа тај број, односно извештај о грешци за случај да су ресурси сервера попуњени. Апликација се у сваком наредном обраћању серверу позива на тај број како би сервер имао информацију са којим мобилним уређајем комуницира. Мобилна апликација на даље комуницира са сервером по принципу прозивања (polling), што значи да се периодично обраћа серверу како би проверила да ли постоји одређени задатак спреман за њу а који сервер проналази у својеврсној торби послова (The bag of tasks). У случају да постоји спреман задатак, он се пребацује на мобилни уређај и започиње његово извршавање.

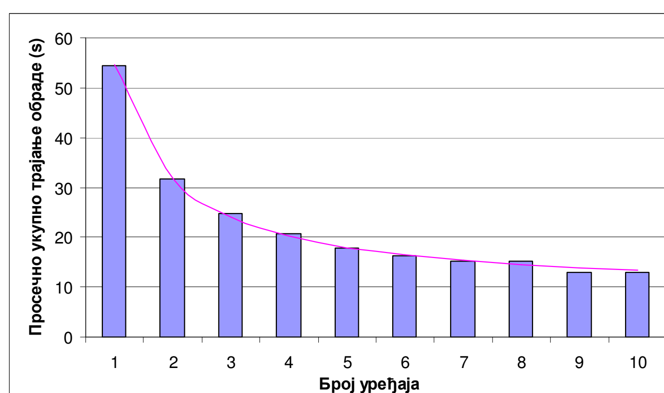
Приликом развоја мобилне апликације посебна пажња је била стављена на делове који се односе на интерпретирање примљених скрипт порука, што је учињено проширивање класе Parser која је развијена у овом пакету. Да би се омогућила размена порука између радних станица, сада на нивоу итерације, било је потребно омогућити серијализацију одређених класа што је била неопходна како би се и сама класа Parser начинила серијализованом. Ово је било неопходно да се уради како би се омогућило паралелно израчунавање појединих послова које сам компонента захтева. На крају је било неопходно извршити мапирање интерног веб сервиса који пружа централни сервер на локалне структуре.

4.1.3.5.3 Резултати мерења

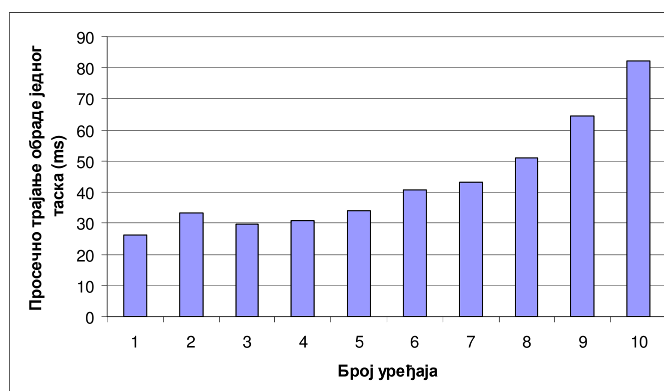
Пошто се ова секција односи на експеримент у раду симулатора у дистрибуираном окружењу на мобилним уређајима овде је дат преглед овог вида рада симулатора. Систем који се извршавао на мобилним уређајима је довољно сложен да би могла да се обави процена понашања система у случају повећања броја мобилних уређаја, процена утицаја времена комуникације на време обраде, као и да се обави процена потрошње батерија на једном уређају.

Пошто је развијени скрипт језик доста једноставан, а време обраде доста велико приликом процене раде решења користећи мобилне уређаје узето је да компоненте симулирају систем који обавља множење матрица. Основна компонента чији се код извршава на мобилном уређају обрађује један део множења матрице и то израчунавање једног елемента резултујуће матрице. Приликом процене брзине рада овако пројектованог система узето је да се варира

број мобилних уређаја а мерено је укупно физичко време извршавања комплетне симулације, логичко време извршавања симулације износило је 1. На сликама 45 и 46 су представљени графици физичког времена извршавања симулације у зависности од броја расположивих мобилних уређаја. Времена обраде на мобилним уређајима могу се моделовати обрнуто пропорцијално са бројем мобилних уређаја, али само у извесној мери зато што доста брзо долази до засићења јер велика количина времена приликом обраде одлази на комуникацију тако да радне станице велики део слог времена троше на комуникацију, а мали на обраду.



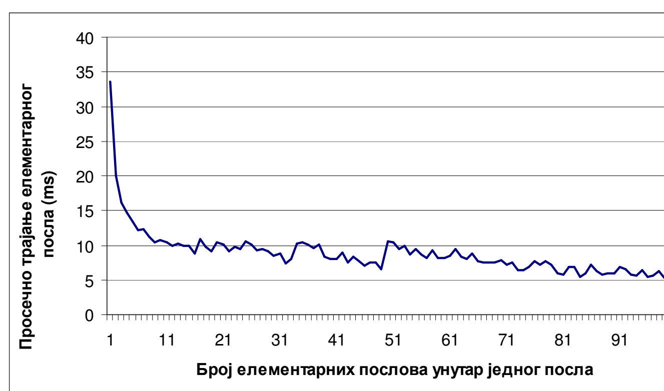
Слика 45: Мерење зависности укупног времена обраде од укупног броја мобилних уређаја на којима се обавља обрада



Слика 46: Просечно време обраде једног догађаја на серверу

Поред заузетости радне станице посматрано је и време обраде које је потребно серверу да прихвати резултате. Може се приметити да просечно време обраде једног задатка расте због веће заузетости сервера већим бројем уређаја и повећања времена комуникације.

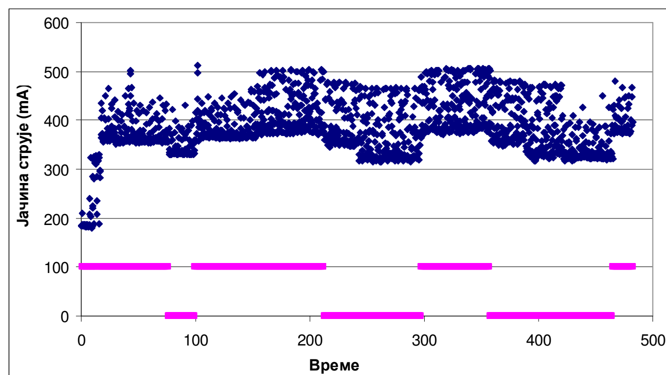
Поред расподеле послова и компонената по мобилним уређајима може се разматрати и повезаност броја компонената по једном мобилном уређају у времена обаде тих свих догађаја на том мобилном уређају. Ово мерење је потребно обавити како би се утврдило утицај времена комуникације и утицај режијских операција на укупно време обраде. Компоненте на мобилним уређајима су извршавале исти посао као у претходном мерењу, само је у овом случају број компонената, односно посао који је обављала једна компонента већи. Резултати овог мерења показују да се груписањем елементарних задатака постиже смањење просечног времена обраде једног елементарног задатка. Приказ резултата мерења је дат на слици 47.



Слика 47: Мерење зависности просечног времена обраде једног задатка од броја елементарних задатака у групи

Када се ради о дистрибуираној обради на мобилним уређајима као један од кључних фактора треба узети потрошњу батерије коју један посао може да изазове. Пошто је у овом случају било потребно урадити мерења на неком конкретном уређају одабрана је Nokia N82, која је у тренутку мерења била распрострањен телефон. Према карактеристикама које је произвођач телефона дао капацитет батерије износи 1050mAh, што овом телефону пружа могућност непрекидног разговора у трајању од око четири часа при просечној потрошњи од 250mA при једном пуњењу батерије. Резултати мерења за посао описан у претходним ставкама су приказани је на слици 48. При обради овог посла просечна потрошња је износила око 390mA. На датом мобилном телефону при измереној потрошњи батерије могуће је остварити око 2 часа и 40 минута континуалне симулације. Овде се може приметити да је време трајања батерије за

1/3 краће у односу на време трајања разговора. Потрошња за обраду је већа из разлога што на потрошњу која се оствари зрачењем самог апарата (GPRS је све време активан) додаје се и интензивно ангажовање процесора због Јава виртуелне машине која обавља симулацију.



Слика 48: Мерење потрошње батерије при континуалној обради јединичних послова

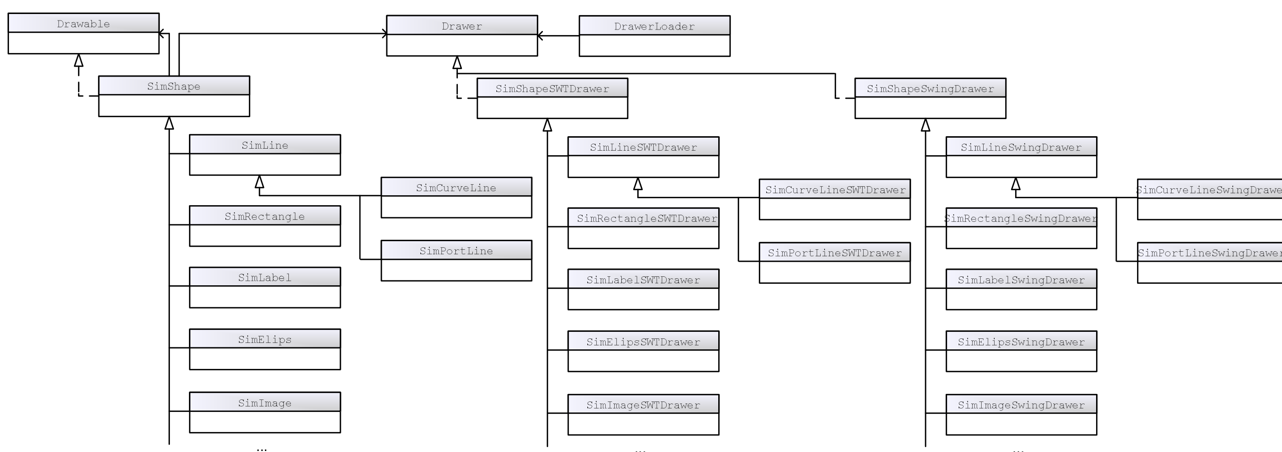
Имајући у виду резултате мерења и чињеницу да се ради о експерименту унутар пројектовања ширег слоја симулатора уочене су ситуације чијом се ефикаснијом реализацијом могу учинити побољшања.

Уочени проблеми би могли да се ублаже уколико би се изменио протокол који користе сервер и радна станица, односно мобилни уређај, тако што би уместо коришћења RPC веб сервиси искористио други протокол XMPP (Extensible Messaging and Presence Protocol). Ово би омогућило да смањи потрошња батерије тако што се веза не би стално одржавала већ би се клијент обавештавао о догађају оног момента када се догађај деси. Овакав вид комуникације би смањио обима саобраћаја према серверу, растерећењем самог сервера и повећањем века батерије на мобилном уређају.

Оно што коришћење мобилних уређаја највише спутава да би се озбиљније користили унутар развијеног симулатора је чињеница да се унутар мобилног уређаја извршава Јава виртуелна машина која интерпретира скрипт језика. Разлог зашто је скрипт језик био је одабран је превазилажење ограничења која намеће Јава ME платформа. Овај недостатак би могао да се ублажи развојем компонента и за сваки тип мобилних уређаја посебно или усавршавањем скрипт језика и његовог преводиоца како би се из постојећих технологија извукле максималне перформансе.

4.1.4 Презентациони слој симулатора

Реализација презентационог слоја симулатора треба да обезбеди механизме за приказивање компонената и веза између логичких компонената. Приликом дефинисања презентационог слоја треба водити рачуна о томе овај слој представља увид који корисник има о стању у коме се нека компонената налази и о везама које са другим компонентама остварује. Потребно је обезбедити да корисници могу да прате сигнале, веза, стања, промене и историју њихових промена. Хијерархија класа која дефинише ове карактеристике је дата на слици 49.



Слика 49: Хијерархија класа презентационог слоја симулатора

Као основни градивни блок у пакету rs.ac.bg.etf.zaki.sleep.presentation.* јавља се интерфејс Drawable чије имплементације треба да обезбеде конкретну имплементацију презентације компонената, веза, слика, лабела и пратећих информација о симулатору. Основне методе дефинисане интерфејсом Drawable могу се поделити у неколико група: методе за исцртавање, метода за померање, методе за повезивање и методе за постављање атрибута визуелне презентације компоненте. Методе за изцртавање компонента представљају везу између онога што се симулира и корисника. Ове методе су од посебног значаја за укупан успех неког симулатора јер што се већине корисника тиче презентациони слој симулатор јесте симулатор. Следећа група метода треба да обезбеде могућност за интеракцију између корисника и визуелне репрезентације једне или више компонента путем померања компонената на одређену локацију. Ове методе у

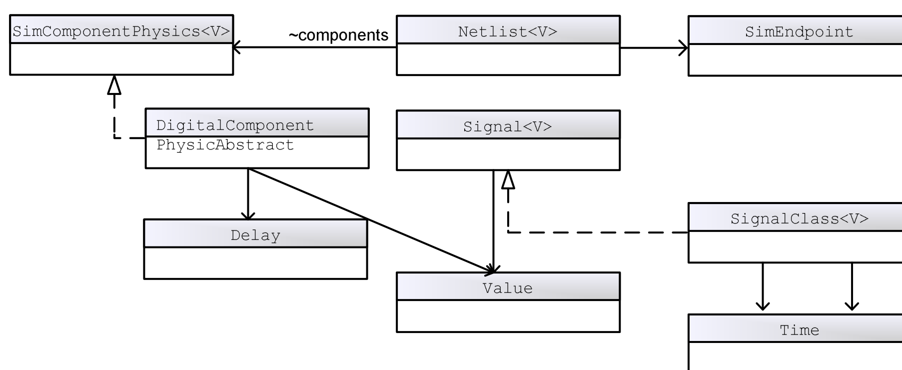
зависности од конкретне имплементације треба да израчунају на коју позицију треба померити компоненту. Ови није једноставна операција транслације јер поред координата свака визуелна презентација садржи и фактор симетрије и угао ротације компоненте. Објекат који имплементира ову класу може да буде композитни објекат који онда рекурзивно треба да позове исте методе. Овде није урађен поступак нормализације и померање сваке компоненте независно јер се ови објекти чине композицију која може да има своја правила и да се понашају као целина на коју нешто треба примењивати. Методе за повезивање специфицирају која визуелна презентација одговара којој компоненти и да ли се интеракција коју је корисник применио односи на неку компоненту. Последњу групу метода чине методе за постављање атрибута визуелне презентације објекта. Атрибути које је могуће постављати односе се на ширину и шрафуру линија којој се компоненте исцртавају, боју компонената, линије и позадине компоненте, текст који се уз неку компоненту исцртава, слике које прате неку компоненту, као и многи други атрибути визуелне презентације. Овде нису укључени атрибути и карактеристике које одређена логичка компонента има, већ се они постављају као параметри одређене имплементације специфицираног интерфејса.

Основне имплементације дефинисаног интерфејса укључују презентацију линије, вишеделне линије са правим и произвољним углом, елипса, правоугаоник, текст, слика, и дијаграм. Једна од имплементација овог интерфејса које наслеђује класу линија је и каса која представља графичку репрезентацију порта.

Визуелне компоненте симулатора моду да буду пројектоване тако да њихов изглед зависи од вредности коју неки од атрибута посматране компоненте може да има. У том случају је визуелне компоненте потребно пројектовати на тај начин да могу да посматрају вредности, атрибуте и параметре компонената. Користећи овај принцип пројектоване су посебне компоненте код којих визуелна презентација зависи од вредности атрибута. Ово је посебно изражено код пројектовања веза између компонената. Визуелна репрезентација веза је пројектована тако да распоред њених графичких елемената увек заузимају исти распоред, али се мења боја и дебљина компонената у зависности од ширине дате везе, односно вредности коју дата веза у датом тренутку има.

4.1.5 Слој физике компонената

Реализација слоја физике компонената треба да обезбеди механизме за дефинисање описа понашања физике коју одређене логичке компонената и веза њих имају. Приликом дефинисања физике компонената води се рачуна о стварним параметрима које нека компонента може да има. Овде компоненте слоја физике не комуницира директно једне са другима већ користе логичке везе које постоје између компонента логичког слоја. Хијерархија класа која дефинише физичке карактеристике компонената логичког слоја је дата на слици 50.



Слика 50: Хијерархија класа слоја физике симулатора

Као основни градивни блок у пакету rs.ac.bg.etf.zaki.sleep.logic.physics.* јавља се интерфејс SimComponentPhysics<V> чије имплементације треба да обезбеде конкретну имплементацију физике понашања компонената. Основне методе дефинисане интерфејсом SimComponentPhysics могу се поделити у неколико група: обрада догађаја, метода за приступ параметрима физике компоненте и методе за чување постављање параметара симулације. Методе за обраду догађаја се ослањају на резултате симулације које су произвеле логичке компоненте. Методе за опис физике компонената се ретко корите, али међу методама које описују рад слоја физике компонената оне су најзначајнијих део симулатора, јер представљају описе понаша физике симулираних компонената. Ове методе, у зависности од типа компоненте о којој се ради, треба да на основу сигнала пратеће компоненте израчуна своје стање. То ново стање физике компоненте описује се променљивама које могу да постоје унутар логичких компонената, али могу да зависе и од већег броја логичких компонената у комплетној хијерархији. Овде се не генеришу нови догађаји које је потребно

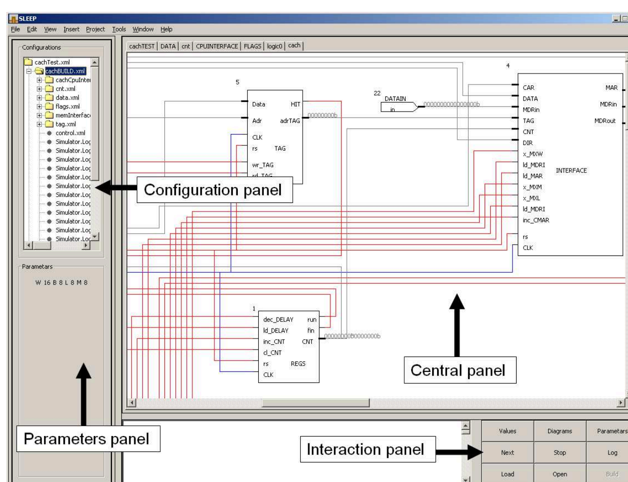
обрадити већ се само врши синхронизација између претпостављених резултата и остварених резултата симулације. Остале групе метода утичу на понашање и постављање параметара израчунавања. Већина физичких карактеристика је изражена промењљивама у покретном зарезу тако да постоје и методе које за задато име враћају одговарајући параметар и вредност.

Основна класа која имплементира интерфејс SimComponentPhysics користећи генерички параметар <DigitalValue> је класа DigitalComponentPhisicAbstract која се налази у пакету rs.ac.bg.etf.zaki.sleep.logic.physics.digital.*. Као основни параметри који издвајају ове дигиталне компоненте имају три специфициране вредности у покретном зарезу. Ради се о напону, струју и дисипацији који су дати у покретном зарезу. Овде је дат једноставан алгоритам за рачунање дисипације које се рачуна кумулативно за представљени период. Дисипација се израчунава на основу напона, који је у директној сразмери са вредностима које имају логичке компоненте, струје која се узима као параметар који се задаје, времена транзиције које се такође задаје као параметар, али и времена колико догу је дата вредност напона дуго постављена. За разлику од компонената логичког слоја компоненте слоја физике могу да мењају своје вредности и на основу једном постављених параметара јер је потрошња у директној сразмери и са вредношћу напона, потрошње струје, али и протеклог времена које се мења. Овакво понашање слоја физике га чини захтевним за израчунавање користећи догађајима вођене симулације јер се не разликује од временом вођених симулација. Приказивање резултата компонената које се налазе у слоју физике се обавља на исти начин као и приказивање резултата логичког слоја. Презентационе компоненте су исте, само се у овом случају користи већи опсег како би се приказале вредности из знатно ширег интервала.

4.2 Начин коришћења SLEEP симулатора

Симулатор SLEEP је симулатор дискретних догађаја који је развијен користећи предложену методологију развоја симулатора архитектуре и организације рачунара. Приликом имплементације предложене методологије водило се рачуна о раду са дигиталним компонентама, као и о извршавању симулације у конкурентном и дистрибуираном окружењу. Предефинисане

компоненте које се користе у симулатору су прилагођене за рад на нивоу трансфера између регистара, али је захваљујући примењеној методологији омогућен развој и других компонента који симулирају системе опште намене које припадају области различитој од архитектуре и организације рачунара, а који се могу описати дискретним догађајима. Типичан изглед основне радне површине SLEEP симулатора коју корисник може да види је приказан на слици 51. SLEEP симулатора омогућава кориснику да креира симулације система састављених од повезаних дигиталних модула и да симулира њихово понашање користећи разне алате који су доступни у оквиру главног екрана.



Слика 51: Типични изглед SLEEP симулатора

Приликом рада са SLEEP симулатором корисницима на располагању стоје следећи делови симулатора:

- Радна површина.
- Преглед компонената.
- Параметри симулације.
- Извршавање симулације.
- Праћење симулације.

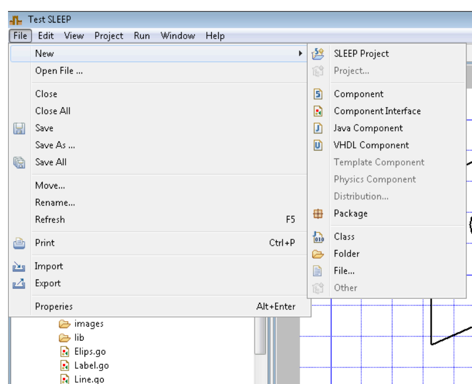
Приликом рада са симулатором корисник креира компоненте које је потребно симулирати на радној површи. Компоненте које се користе у симулатору могу бити креиране користећи различите технике. За сваку од техника креирања компонената постоји посебан алат који ће се приказати у оквиру радне површи. Уколико је потребно користити раније развијене компоненте које стоје на располагању у библиотеци компоненте у текућем пројекту или у екстерној

библиотеци компонената. Информације о коришћеним компонентама и пројектима, су приказане на површи за преглед компоненти. Кориснику су подразумевано доступне компоненте које се креирају у текућем пројекту, као и основне дигиталне компоненте, док се компоненте креиране у неким претходним пројектима морају посебно учитати. Поред основних хијерархијских компонената SLEEP симулатор омогућава рад и са конфигурабилним компонентама које омогућавају кориснику да зада почетне вредности и капацитете тих компонената. Параметризоване компоненте односно параметри тих компонената се приказују на посебној површини која служи за приказ параметара симулације. Након креирања компоненте прелази се на њену симулацију. Поступак симулације је омогућен користећи панел са алатима намењеним извршавању симулације. Кориснику је омогућено да покрене симулацију, и да зада временски тренутак на који жели да оде. Уколико симулација захтева интеракцију са корисником онда је потребно обезбедити да корисник може да континуално уноси вредности. Праћење самог тока симулације је омогућено користећи доњи део екрана на коме се налазе параметри који се могу пратити. Симулатор омогућава праћење тока симулације користећи већи број модова рада, користећи дијаграме, табелу са вредностима и директан испис резултата симулације. Резултати симулације су представљени као скуп вредности и тренутака у којима су те вредности промењене. Могуће је пратити сваку конекцију између компонената и веза које их спајају.

Уколико се веза може представити једном линијом она се у случају рада са дигиталним компонентама та веза може приказати са танком црвеном, плавом или тамно сивом линијом при чему боје одговарају вредностима једне, нула или стање високе импедансе, респективно. Ако веза представљају магистралу или више веза онда се она може посматрати као групу сигнала. Група сигнали су приказани са дебелим светло сивим линијама са вредностима приказаним у бинарном или хексадецималном формату. Симулатор омогућава да се графички представљају сигнали користећи временски дијаграми одабраних сигнала који се приказују у панелу са сигналима који се отвора активирањем одговарајућем таба на панелу за интеракције.

4.2.1 Креирање логичких компонената

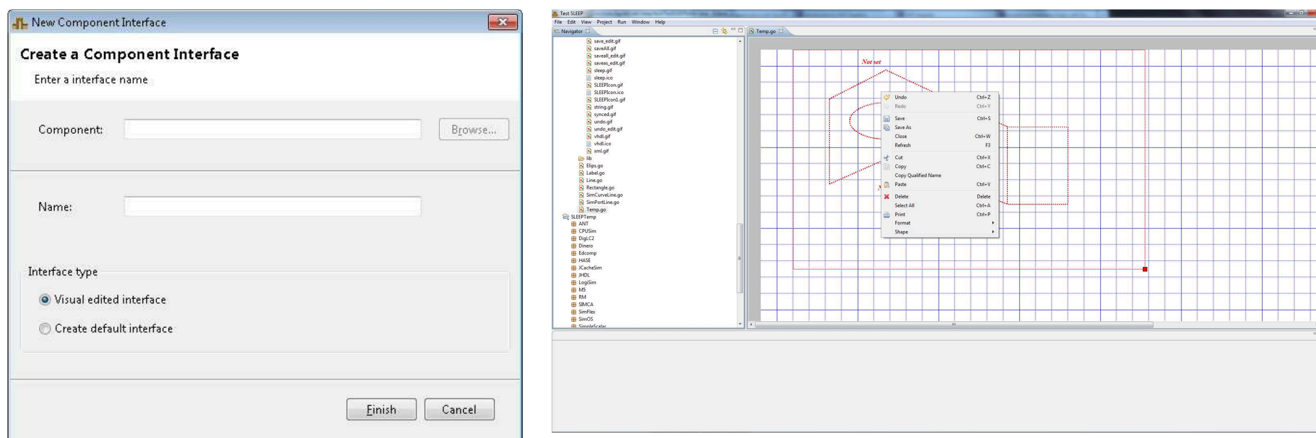
Основни градивни блок који се користи у поступку формирања симулатора је логичка компоненета која се симулира. Приликом креирања компонената могу се користити уграђени алати. Ови алати омогућава креирање хијерархијске компоненте користећи графички поступак, али и могућност креирања компонената користећи текстуални едитор и опис компонената дат помоћу Јава или VHDL програмског језика. Поступак креирања компоненте је таква да се одабере опција new и онда се одабере који тип компоненте треба да се реализује. Изглед екрана помоћу кога се бира тип компоненте који се реализује је приказан на слици 52.



Слика 52: Основни мени приликом креирања пројекта и компоненти SLEEP симулатор

4.2.1.1 Коришћење графичког едитора

За креирање логичких компонената најједноставнији начин је коришћење графичког едитора. Графички едитор се покреће уколико је логичка компонента која се развија креирала као графичка компоненета. На слици 53 је приказана сама радна површ која служи за повезивање логичких компонената у циљу формирања сложене хијерархијске компоненте. Радна површ служи за позиционирање одабраних логичких компонената у дводимензионалном простору.



Слика 53: Креирање интерфејса компоненте користећи Графички едитор SLEEP симулатора

Креирање, чување и проналажење компоненти и њихова симулација је подржан са различитим функцијама које се активирају помоћу главног менија и одговарајућег менија. Појава искачућег менија варира у зависности од положаја и активног контекста Централног панела. Сви менији и компоненте су контекстно осетљиви тако да њихова улога зависи од компоненти које су одабране. Размештање компонената се обавља на централном панелу користећи превлачење мишем и искакајући мени. Приликом рада са дигиталним компонентама кориснику на сваком искачућем менију стоје на располагању следеће функције: Додавање, померање, копирање и брисање логичких компонената. Додавање, померање, и брисање веза између компонената. Додавање, померање, и брисање група компонената и пратећих веза. Додавање, померање, копирање и брисање коментара. Увећавање и умањивање радне површине.Померање унапред и уназад компонената, веза и њихових група. Преглед особина и параметара компонената.

Компоненте које се могу постављати на радну површ долазе из различитих библиотека компонената. Убацивање ових компонената на радну површ се постиже или превлачењем одговарајуће компоненте из библиотеке на радну површ или двокликом на одговарајућу компоненту која ће се онда појавити на радној површи на месту на коме је последњи пут на радној површи било кликнуто. Ове компоненте које су учитане из библиотека могу да буду једноставне тако да немају унутрашњу структуру или сложене са хијерархијском структуром. Уколико се над неком компонентом направи двоклик онда се у посебном табу отвара дефиниција те компоненте приказана користећи неки од

начина за представљање компонента: графички или текстуални у зависности од начина на који је компонента реализована.

Приликом рада корисник може да одабере и параметризовану компоненту. Параметризоване компоненте садрже параметре које је могуће конфигурисати који најчешће описују величине, капацитета, или друге важне карактеристика компоненте (нпр. И коло са параметром који говори број улазних портова). Рад са параметризованим компонентама олакша стварање описа сложених система који омогућавају пуну хијерархијску параметризацију. Ово знали да параметри компонента једном нивоу хијерархије могу да зависе од параметара на следећем нивоу хијерархије. Ове параметре није неопходно поставити све до времена када је потребно обавити симулацију. Списак параметара селектоване компоненте добије кликом на компоненту када се појављује искачући неми са овом особинама, али и уколико се одабере опција додавања параметра у компоненту која се тренутно пројектује.

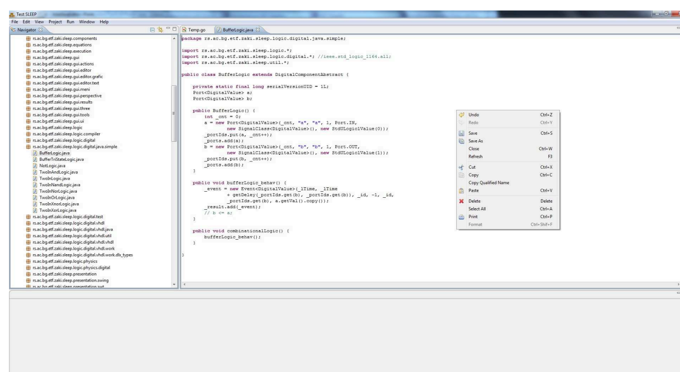
Графички едитор омогућава кретање компоненти, цртање веза између пинова компоненти, измену облика линија и додељивање коментара, лабела и слике унутар дате радне површи. Веза између две одабрана пина су нацртана као одговарајућа изломљена права линија. Облик линија се може модификовати превлачењем и отпуштањем неких од дужи те линије. Додавање лабела се може користе за именовање сигнала, групе сигнала и било какве корисне информације а може се креирати, мењати, преместити и обрисати користећи искачући мени. Коментари и слике се такође могу поставити на било ком месту у оквиру радне површи са циљем објашњење схватања рада симулиране компоненте и немају никаквог утицаја на сам ток симулације. Дефинисање параметара за компоненту се врши активирањем Параметар опције из искачућег менија.

Приликом рада са компонентама које се налазе на радној површини могуће их је померати на произвољну позицију унутар радне површи. Поред померања појединачних компонената могуће је померати и групе компонената по радној површи. Да би се формирала група компонената потребно је притиснути миша и вући га по радној површи. Све компоненте које се буду нашле унутар описаног квадрата припадају одабраној групи. Уколико су све компоненте унутар групе такве да нису повезане ни са једном компонентом изван групе биће одрађено

једноставно померање. Уколико су компоненте такве да су повезане са неком изван групе онда ће се компоненте унутар групе померати али ће се везе које повезују те компоненте са онима изван групе прилагођавати компонентама чија је позиција остала иста.

4.2.1.2 Креирање компоненти користећи Јава код

Нова компонента се такође може дефинисати користећи текстуални едитор. Поступак креирања компоненaта у програмском језику Јава захтева да се на почетку рада компоненaта декларише како Јава компонента, односно јава класа која треба да садржи опис компоненте. Да би компонента/класа могла да се интегрише у симулатор потребно је та класа имплементира интерфејс DigitalComponentAbstract. Ово ће бити аутоматски бити постављено након дефинисања имена класе која треба да се извршава. Овај начин дефинисања компоненaта омогућава дефинисање веома сложених система који не морају да буду дигитални уређаји већ могу бити произвољне компоненте дефинисане у произвољном програмском језику према коме постоје интерфејс приступа из програмског језика Јава. Овај приступ омогућава кориснику да дефинише веома сложен компоненте на нивоу понашање а не на структурном нивоу. Предност коришћења компоненти дефинисаних у програмском језику Јави је то симулација је бржи у односу на симулацију компоненте са истим понашањем дефинишу помоћу графички едитор. Мана коришћења компоненти дефинисано у Јави је у томе што корисник не може да види унутрашњу структуру компоненaта нити параметре у току симулације.

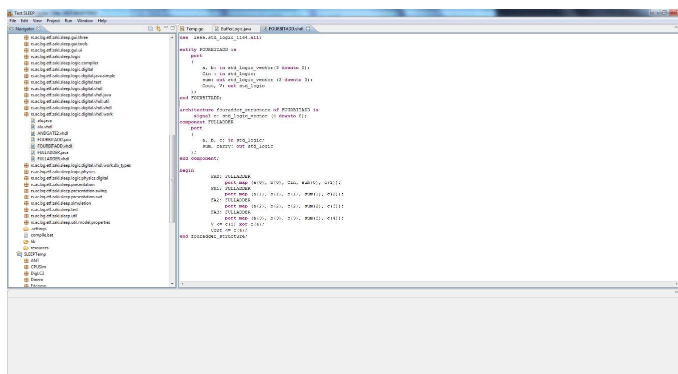


Слика 54: Развој компоненaта користећи текстуални едитор и Јава програмски језик

На слици 54 је дат приказ развоја генеричке двоулазне компоненте из које су изведене све двоулазне компоненте. Развој компоненти библиотеке врши се у bootstrapping начин применом свих основних и параметризованих компоненте са текстуалним уређивање. Да би компонента развијена користиће програмски језик Јава могла да се даље користи унутар пројекта потребно је превести дату компоненту из јава описа у опис који може да се извршава, у одговарајућу класу. Позивање операције за формирање компоненте се постиже притиском на дугме Build помоћу кога се креира класа као и одговарајући опис унутар симулационе компоненте који се чува унутар одговарајуће датотеке користећи XML опис. Да би се уштедело на простору приликом чувања компоненената компоненте се пакују у одговарајуће архиве, како би могле да се извршавају било на рачунару на коме је пројектоване, било на серверу или радним станицама у случају рада у дистрибуираном окружењу.

4.2.1.3 Креирање компоненти користећи VHDL код

Следећи начин за опис компоненената је коришћење текстуално едитора у коме се компоненте описују користећи VHDL код. Поступак креирања компоненената у језику за опис хардвера VHDL захтева да се на почетку приликом креирања компоненената, та компонента декларише како VHDL компонента. Да би компонента писана на овај начин могла да се интегрише у симулатор потребно је од те компоненте креира класа која имплементира интерфејс DigitalComponentAbstract. Овај приступ омогућава кориснику да дефинише веома сложен компоненте на нивоу понашање користећи описе доста компоненената који су расположиви. Предност коришћења компоненти дефинисаних програмском језику VHDL је постојање великог броја описа компоненената који су развијене користећи овај програмски језик, и што су параметри тих компоненената прилагођени стварним системима и њиховим кашњењима. Мана коришћења компоненти дефинисаних користећи језик VHDL су исте како код компоненената писаних користећи језик Јава, а је немогућност да корисник прати унутрашње понашање компоненте током симулације већ може да прати само сигнале који се дефинишу на њеним излазима.

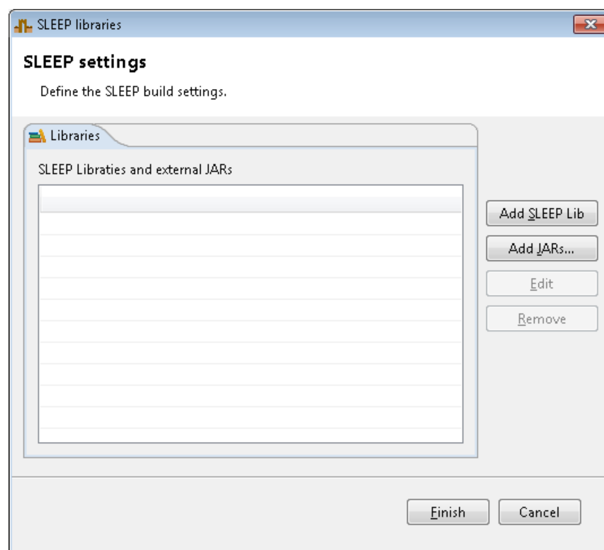


Слика 55: Развој компонената користећи текстуални едитор и VHDL програмски језик

На слици 55 је дат приказ развоја компоненте потпуног сабирача. И овај начин развоја компонената омогућава развој библиотека компонената дефинишући компоненте најнижег нивоа којима се након тога додељује графички интерфејс. Да би компонента развијена користиће језик VHDL могла да се даље користи унутар пројекта потребно је превести дату компоненту из описа у Јава класе. Позивање операције за формирање компоненте се постиже притиском на дугме Build помоћу кога се прво креира одговарајућа Јава датотека која садржи опис компонената, а након тога се од дате Јаве датотеке креира класа као и одговарајући опис унутар симулационе компоненте који се чува унутар одговарајуће датотеке користећи XML опис.

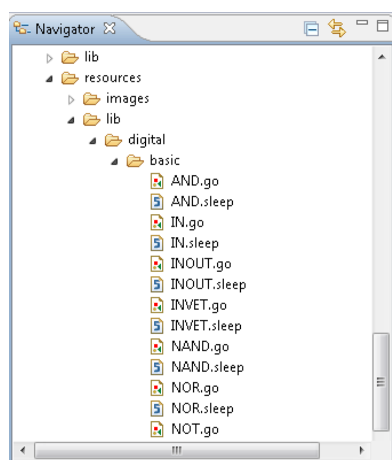
4.2.1.4 Библиотеке компонената

Приликом рада са дигиталним уређајимаociснички обично не започиње свој рад од почетка без коришћења већ развијених дигиталних компонената. Приликом креирања пројекта корисник задаје које библиотеке компонената жели да користи. Када укељући одређене библиотеке онда оне постају доступне током развоје текућег пројекта. Поступак учитавање библиотеке компонената је такав за захтева да се одабере опција Properties за дати пројекат и онда се у картици Build Path дода директоријум са библиотекама. На слици 56 је приказан поступак убацивања нове библиотеке у постојећи пројекат.



Слика 56: Додавање нове библиотеке у постојећи пројекат

На слици 57 је дат приказ хијерархијске представе доступних пакета и компонената које се могу користити унутар текућег пројекта. Панел на коме су видљиве све компоненте које су тренутно учитане се назива Navigator. учитавање компоненте се постиже тако што корисник одабере компоненту и онда или направи двоструки клик на њу или је превуче на радну површину.



Слика 57: Библиотеке за избор компонената симулације. Компоненте са екстензијом .sleep представљају логичке компоненте а са екстензијом .go графички интерфејс логичке компоненте

Унутар симулатора је доступна основна библиотека компонената која садржи дигитална кола која се могу користити пликом развоја система који се могу користити у области архитектуре и организације рачунара. Унутар стандардне библиотеке на располагању стоје следеће логичке комбинационе и секвенцијалне компоненте:

Стандардна логичка кола: And2, Or2, Nand2, Nor2, Xor2, Xnor2, And3, Or3, Nand3, Nor3, Xor3, Xnor3, And4, Or4, Nand4, Nor4, Xor4, Xnor4, Not1, Buff1, Tri1. Сва ова логичка кола имају један излазни порт који је типа OUT и одређен број улазних портова, типа IN. Број улазних портова је специфициран бројем садржаним у имену компоненте.

Стандардни комбинациони модули: MP2_1, DP1_2, DC1_2, CD2_1, Add1, CMP1, MP4_1, DP1_4, DC2_3, CD4_2, MP8_1, DP1_8, DC3_8, CD8_3. Број улазних и излазних портова је специфициран бројем поред садржаним у имену компоненте. Све ове компоненте су сачињене користећи стандардна логичка кола, тако да се може видети њихова интерна структура. Овде треба напоменути да није дозвољено модификовати компоненте које су учитане из неке библиотека, осим ако се те компоненте не учитају у пројекат и тада модификују. У том случају се раскида референца са старим компонента. Уколико се користе компоненте из другог пројекта и у том пројекту се компоненте промене ни у увом случају неће доћи до ажурирања компонента јер су оне копиране у дати пројеката. Уколико се желе модификовати нека компонента онда је потребно у текућем пројекту тражити освежавање одговарајуће библиотеке. То је операција Refresh.

Конфигурабилна стандардна логичка кола: AndN, OrN, NandN, NorN, XorN, XnorN, BuffN, TriN. Сва ова логичка кола, осим последња два, имају један излазни порт који је типа OUT и N улазних портова, типа IN. Број улазних портова се специфицира приликом покретања симулације или приликом коришћења компоненте. Том приликом се уноси вредност аргумента N. Унос конфигурабилних параметара неке компоненте се постиже попуњавањем одрешеног поља дате компоненте добијеног из спецификације дате компоненте (Properties). Поступак постављања параметара је приказан на слици 63.

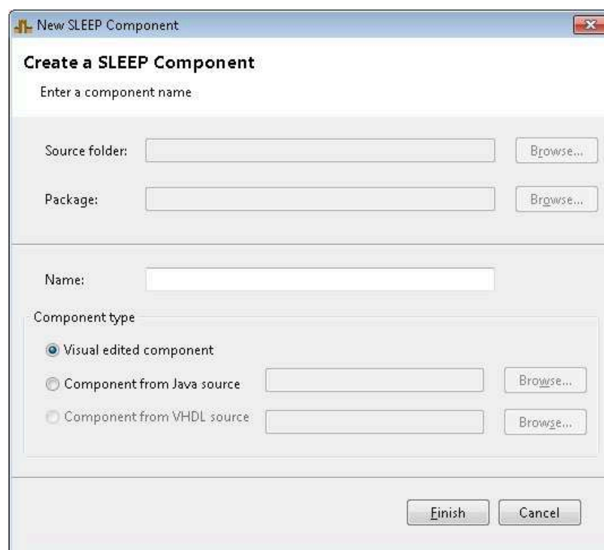
Конфигурабилни комбинациони модули: MPN, DPN, DCN, CDN, AddN, CMPN. Број улаза и излаза се конфигурише на исти начин као и код конфигурабилних логичких кола. Уколико би се покушало да се види интерна структура ових компонената тада би се уместо логичке шеме приказао одговарајући Јава код који описује понашање ових компонената.

Флип-флопови: DFF, RSFF, TFF, JKFF, DMSFF, RSMSFF, TMSFF, JKMSFF и DEFF. Овде се ради о стандардним секвенцијалним мрежама чији опис може да

буде дат на два начина. Опис компонената DFF, RSFF, TFF и JKFF је дат користећи Јава програмски језик док је опис преосталих флип-флопова дат користећи логичке шеме и стандардне логичке модуле са повратном спрегом.

4.2.2 Креирање графичког интерфејса компонената

Да би се креирала компонената коју је могуће користити у неком другом пројекту или у оквиру истог пројекта у поступку стварању неке нове компоненте потребно је дефинисати графички симбол компоненте. Дефинисање нове компоненте постиже се користећи посебан скуп алата који су саставни део окружења. Ови алати омогућава креирање графичког интерфејса компоненте користећи графички поступак, али и могућност креирања компонената користећи текстуални едитор и опис интерфејса компонената дат помоћу Јава програмског језика. Поступак креирања графичког интерфејса компоненте је таква да се одабере опција new и онда се одабере који тип графичког интерфејса компоненте треба да се реализује. Изглед екрана помоћу кога се бира тип графичког компоненте који се реализује је приказан на слици 58.

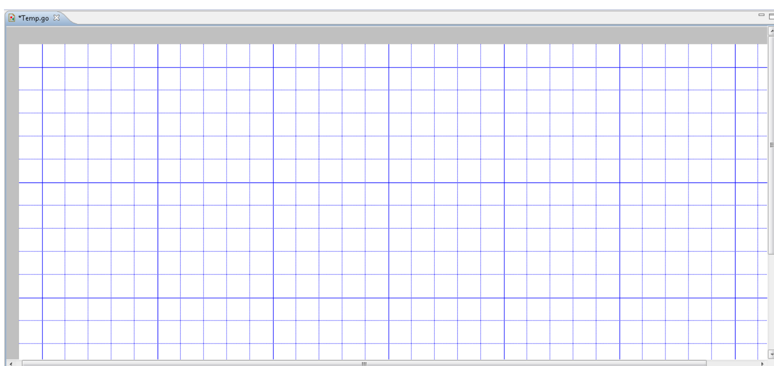


Слика 58: Поступак креирања логичке компоненте користећи SLEEP симулатор

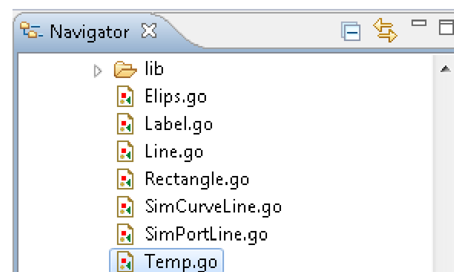
4.2.2.1 Коришћење графичког едитора

За креирање графичког интерфејса логичких компонената најједноставнији начин је коришћење графичког едитора (Symbol Graphical Editor). Графички едитор се покреће само уколико је графички интерфејс логичке компонента која се развија означен као графичка компонената. На слици 59 је приказана радна

површ која служи за повезивање графичких елемената у циљу формирања презентационог слоја логичке компоненте. Радна површ служи за позиционирање одабраних графичких елемента у дводимензионом простору. Овај дводимензионални простор на коме се могу распоређивати графички елементи је омеђен правоугаоником одређене величине.



Слика 59: Радна површ алата Symbol Graphical Editor



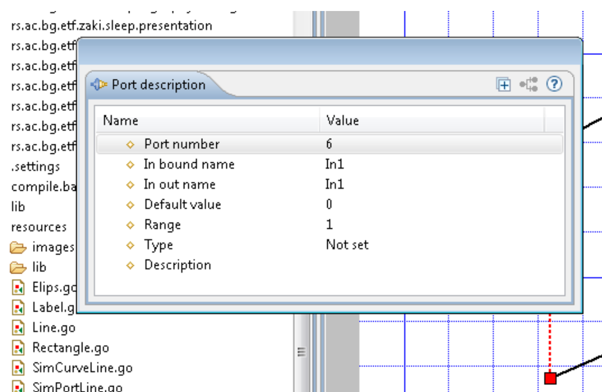
Слика 60: Доступне графичке компоненте

Приликом креирања графичког интерфејса компоненте користећи графички едитор корисницима на располагању палета компоненти који се могу користити за манипулацију радном површи: Elips, Label, Line, Port, Rectangle, Curve Line и Image. Опција Line служи за додавање изломљене праве линије на радну површ. Ова линија има декоративну улогу да потпомогне бољој визуелној репрезентацији пројектоване компоненте.

Одабир одговарајуће презентациони компоненте постиже се помоћу њеног избора или са десног дела или помоћу неког од командних дугмади у доњем делу радне површи. Графички симбол је дефинисан избором једног од неколико предефинисаних шаблона облика или цртањем нових облика коришћењем палете за цртањем компонената. Шаблонски облици могу бити проширен са новим нацртаним облицима. Сваки облик може да садржи у свом саставу праве линије, криве линије, слике, или лабеле. Поред додавања линије унутар симулатора постоји додавање квадрата на радну површ користећи опцију Rectangle. Квадрат, као и линија има декоративну улогу и димензија се поставља или постављањем вредности у опцију особина компонената или користећи миша који се повлачењем из доњег десног угла компоненте. Приликом рада са компонентама могуће је

додавати коментаре и текста на радну површ користећи опцију Lable. Текст који се може додати кодира се као UTF-8 тако да је могуће уносити и ћирилични текст. Дугме Image служи за додавање слике на радну површ. Слика се учитава са адресе на коју укаже корисник користећи стандардни Open дијалог. Сви поменути графички облици што се самог симулатора тиче имају декоративну улогу, али представљају основни начин на који корисници могу да виде симулиране компоненте.

Приликом рада са графичким интерфејсима компонената потребно је извршити повезивање презентационих и логичких портова компоненте. Ово је потребно урадити како би се приликом коришћења графичког едитора за повезивање логичких компонената знало на који начин су повезане поједине компоненте, односно како би се креирала листа повезаних компонената, netlist. Опција Port служи за додавање праве линије која репрезентује приступну тачку пројектоване компоненте. Поред линије појавиће се и дијалог за унос параметара дате приступне тачке. Портови су посебна врста линија које се користити као тачке за повезивање са осталим компонентама. Након завршетка пројектовања графичког интерфејса компоненте, неопходно је да се мапирати портове дате компоненте на визуелне компоненте портова. Порт се мапира попуњавањем одговарајућег дијалога који садржи улазни и излазни називе портова, тип портова (улазни, излазни, или улазно/излазни), подразумеване вредности, као и ширине портова. Логичке компоненте креиране користећи графички едитор приликом креирања графичког интерфејса имају могућност да аутоматски попуне параметре приступних тачака уколико се приступна тачка повеже са одговарајућим параметрима. Приликом рада са логичким компонентама које су дефинисане користећи текстуални едитор потребно је попунити сва поља везана за њихове приступне тачке. На слици 61 је дат пример повезивања улазног порта са параметрима логичке компоненте.



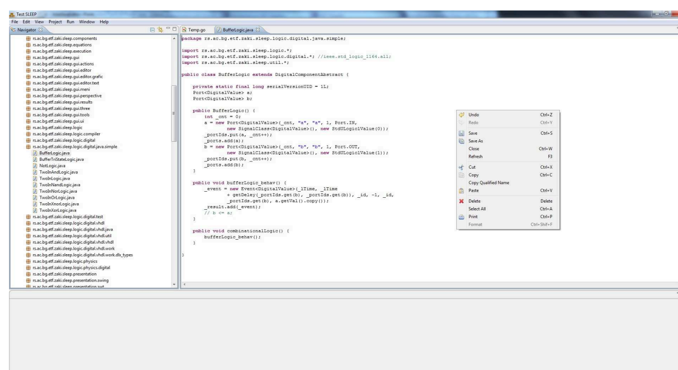
Слика 61: Повезивање приступне тачке компоненте са њеном графичком репрезентацијом

Графички интерфејс компоненте је могуће снимити независно од логичке компоненте за коју је повезана. На овај начин је могуће креирати библиотеке интерфејса компонената. Ове библиотеке се понашају на исти начин као библиотеке логичких компонената, те је и поступак убацивања ових библиотека у текући пројекат исти. Основне површине за смештање презентационих компонената на радну површину како би се креирао сложен интерфејс компоненте започиње тако што се из библиотеке изабере презентациона компонента која се жели убацити у пројекат. Избор презентационе компоненте се постиже проналажењем имена компоненте у левом горњем менију у облику стабла. Када се изабере презентациона компонента и на њу се кликне два пута она ће се појавити на радној површини. Тако изабрана компонента се може даље позиционирати унутар области графичке компоненте на радној површини.

4.2.2.2 Коришћење текстуалног едитора

Креирање графичког интерфејса логичке компоненте се такође може дефинисати користећи текстуални едитор. Поступак креирања компонената у програмском језику Јава захтева да се приликом креирања графичког интерфејса декларише да се ради о Јава компонента, односно јава класа која треба да садржи опис компоненте. Да би компонента/класа могла да се интегрише у симулатор потребно је та класа имплементира интерфејс DrawableAbstract. Ово ће бити аутоматски бити постављено након дефинисања имена класе која треба да се извршава. Овај начин дефинисања графичког интерфејса компонената омогућава дефинисање веома сложених и динамичких система који не морају да буду дигитални уређаји већ могу да мењају своју графички репрезентацију у

зависности од вредности појединих аргумената, вредности или параметара. Овај приступ омогућава кориснику да дефинише веома сложен графички интерфејс компоненте који је везан за логичку компоненту на нивоу понашање.



Слика 62: Развој корисничког интерфејса компоненте у Јава програмском језику

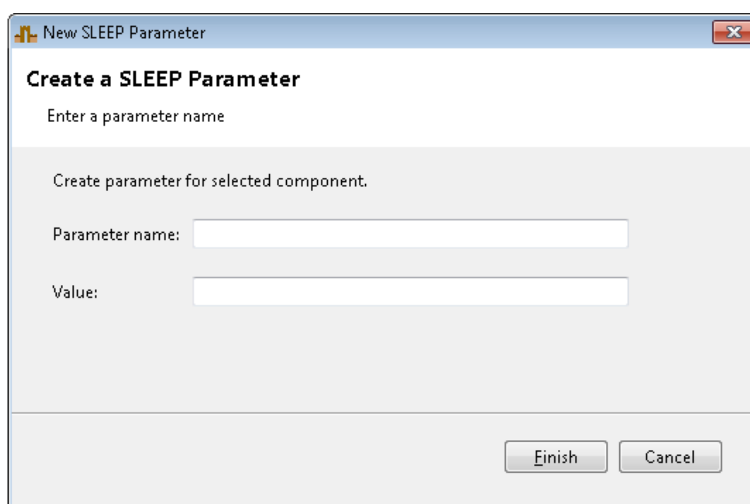
На слици 62 је дат приказ развоја графичког интерфејса компоненте која зависи од стања компоненте. Ради се о основним градивном блоку који се користи приликом рада са симулатором дигиталних система, о логичкој вези. Развој сложених интерфејса логичких компоненти омогућава кориснику да опише сложено понашање које неке компоненте захтевају, као што је испис резултата код седмосегментног дисплеја. Да би графичке компоненте развијене користећи програмски језик Јава могла да се даље користи унутар текућег пројекта потребно је превести дати опис превести из јава описа у опис који може да се извршава, у одговарајућу класу. Позивање операције за формирање компоненте се постиже притиском на дугме Build помоћу кога се креира класа као и одговарајући опис унутар симулационе компоненте који се чува унутар одговарајуће датотеке користећи XML опис.

4.2.3 Задавање параметара компонената

Приликом пројектовања компоненти користећи SLEEP симулатор могуће је поред стандардних компоненти чија је целокупна структура позната у време пројектовања формирати је посебан тип компоненти чији се параметри задају или приликом коришћења у неком наредном блоку или трку у време извршавања. Овај тип компоненти се назива параметризована компонента или шаблон компонента. Све ове компоненте морају да у свом опису као најнижи блок користе логичке компоненте које имају постављени скуп поља преко којим им се задаје

функционалност и особине компонената. Функционалност и особине компоненте зависе од постављених улазних параметара. Ове компоненте су пројектоване као генеричке компоненте за чије је коришћење потребно постављање одговарајућих параметара. Према утицају који параметри могу да имају на понашање компонената можемо их поделити на две групе: параметри са мултипликативним утицајем и на параметре са функционалним утицајем.

Приликом рада са параметризованим компонента код којих се под утицајем неког параметра мења мултипликативност неког аргумента мора се водити рачуна о правилним повезивању тих компоненте у систем. Ово је битно напоменути јер неке приступне тачке компонената са којима се задата компоненте повезује не морају да имају особину мултипликаввности. Уколико се ради о текстуалном опису компоненте онда цео опис остаје исти и нема никаквих разлика у реализацији. Разлике настају када се ради о параметризованим компонентама код којих опис није дат текстуално већ користећи повезивање логичких елемената. На слици 63 је дат начин како се параметризовани компонентама задаје неки параметар. Прво се одабере компоненте, па се оде на особине дате компоненте и онда се излиста списак параметара које је могуће поставити. Неки параметри су нумеричког типа тако да захтевају срачунавање вредности, док су неки не нумеричког типа, те се њихова вредност не срачунава. Срачунавање вредности израза се остварује користећи посебну библиотеку.

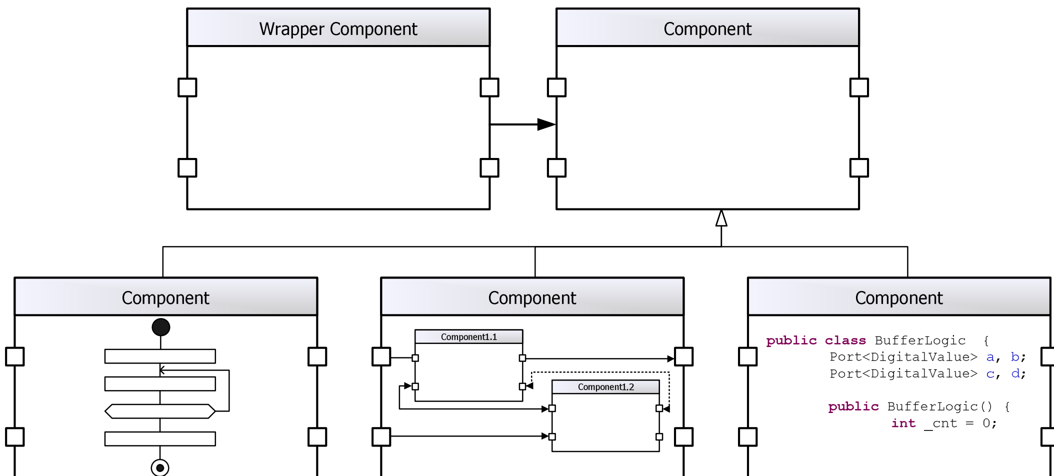


Слика 63: Креирање параметара компоненте

Када се ради са параметризованим компонената код којих се мења њихова функционалност, понашање, поступак пројектовања је идентичан пројектовању осталих параметризованих логичких компонената. Код овог типа компонената се задаје као параметар конфигурација која се жели обрадити а свака од тих конфигурација се онда независно пројектује. Ово је могуће одрадити уколико свака компонента поседује свој омотач у коме се налазе само основне информације о компоненти и неки параметри компоненте без детаља имплементације. Овакав опис захтева да компоненте има исте приступне тачке само се разликује по понашању. Пројектовање параметризованих компоненти захтева поред формирања саставних делова и формирање посебног дела који води рачуна о повезивању свих компоненти према одређеним правилима. Овакве компоненте поред правила за повезивање морају да поседују и део који се односи на начин распоређивања компонената на радној површи. Код SLEEP симулатора генеричке компоненте не могу имати различите интерфејсе приликом исцртавања већ имају јединствен интерфејс, док сама компонента одређује како ће се унутрашњост исцртавати.

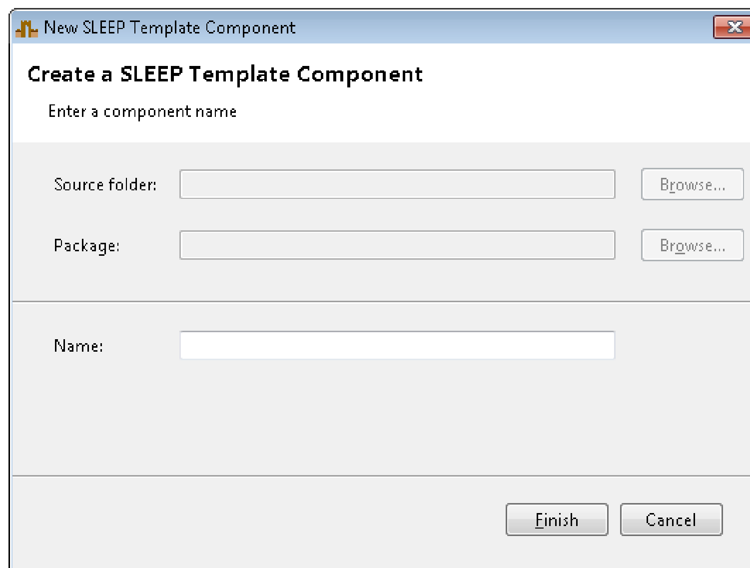
4.2.4 Шаблонске компоненте

Посебна врста параметризованих компонената су шаблонске компоненте. Ова група компонента омогућава кориснику да када зада одређени параметар компоненте може да види интерну структуру те компоненте коју је одабрао из шаблона. Бита је разлика јер се параметризованих компонената код којих се мења њихова функционалност кликом на одређену компоненту види њена интерна структура, а не структура компоненте која је постављена као шаблон. На слици 64 је дат пример шако се шаблонске компоненте користе.



Слика 64: Типични начини коришћења шаблонских компонената

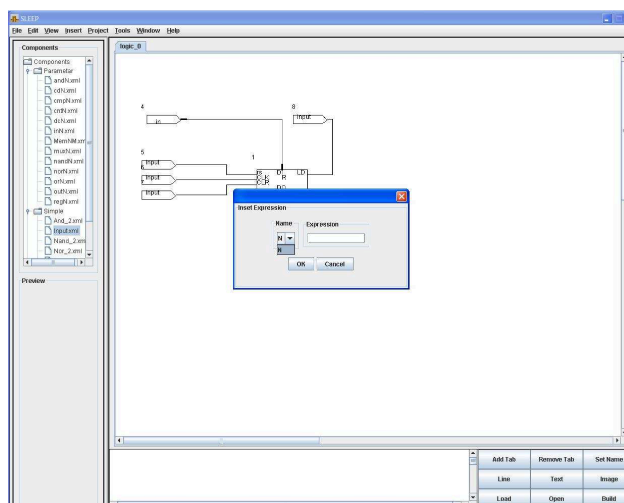
Поступак креирања шаблонских компонената се мало разликује од поступка креирања параметризованих компонената јер се као основни блок на почетку не узима креирање логичких компонената већ шаблонских компонената. На слици 65 је дат поступак креирања шаблонских компонената.



Слика 65: Креирање шаблонске компоненте

Овакав приступ пројектовању симулатора омогућава комбиновање и испитивањем понашања различитих имплементација компонената. Иста компонента, на пример мултиплексер са 4 информациона улаза може да буде реализован користећи И/ИЛИ/НЕ логичка кола, или опис може да буде дат

користећи језике Јава или VHDL, или као текстуалну датотеку са описом понашања. На слици 66 је дат опис промене параметра код шаблонских компонената.



Слика 66: Постављање параметара компоненте

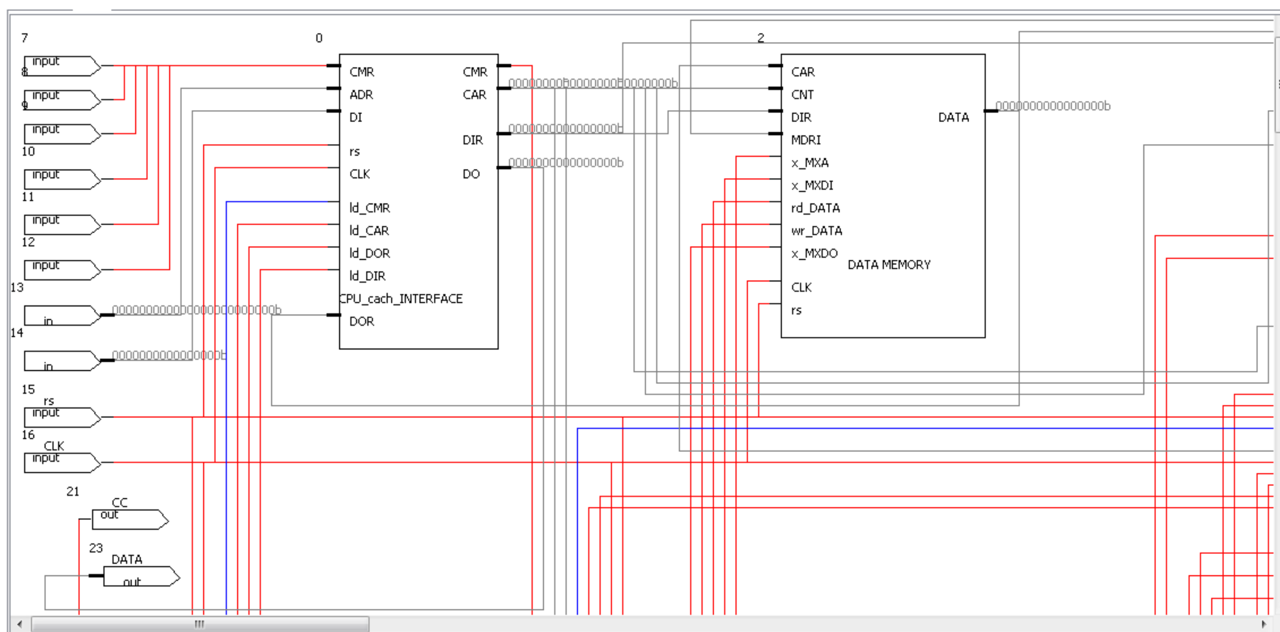
Коришћење шаблонских компонената омогућава униформин приступ моделовању архитектуре система, градивних блокова који ће се користити у поступку пројектовања, као и постепену замену датих блокова прелажењем на следећи ниво имплементационих детаља које садржи опис дате компоненте. Коришћење шаблона омогућава промену целокупне реализације компонената и поступка извршавања симулације на тај начин што се променини имплементације извршне класе док комплетан интерфејс и везе према другим компонентама остаје непромењен. Поступа коришћења шаблона је реализован захваљујући особини објеката оријентисаних језика да раде раздвајање имплементације и интерфејса, то јест услуга које треба да пружа одређени објекат од потписа те услуге.

4.2.5 Праћење тока симулације

SLEEP симулатор омогућава корисницима да у току рада посматрају поједине атрибуте компонената и веза унутар симулатора. Симулатор омогућава више начина за праћење атрибута у зависности од потреба корисника. Симулатор омогућава више начина праћења рада симулираног система. Први и основни начин праћења атрибута компонената и веза подразумева праћење промене

директно на презентационом слоју компонената. Остали видови праћења рада система подразумевају коришћење механизма за праћење промене атрибута користећи временске облике сигнала, табеларни или текстуални приказ рада система.

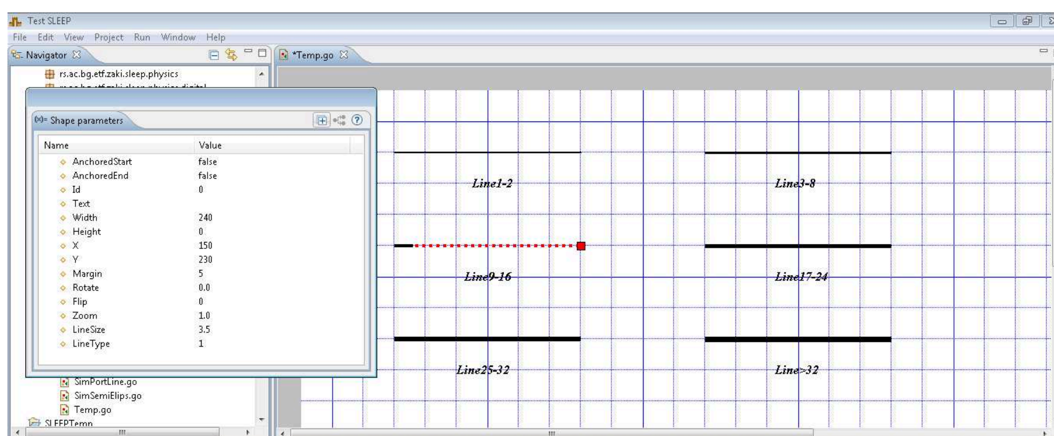
Праћење промена директно на презентационом слоју захтева коришћење компонената које умеју да интерпретирају интерно стање атрибута и у зависности од тог стања на презентационој површини користећи одговарајућу боју прикажу резултате. Овај приступ се може применити у случајевима када се жели посматрати понашање читавог система, не у временском домену већ надгледањем свих логичких компонената у неком одређеном тренутку. Овај вид коришћења симулатора је основни вид и даје најбољу прегледност тренутног стања компоненте, као и могућност за лакшу детекцију проблема. Поред добрих страна овај вид приказа резултата може да буде непогодан јер је потребно ићи кроз симулацију корак по корак што захтева пуно времена за визуелну инспекцију. На слици 67 је дат пример праћења рада делова система директним посматрањем компоненти.



Слика 67: Праћење стања система директним посматрањем компоненти

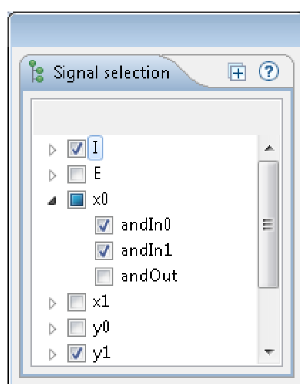
Приликом рада са дигиталним компонентама уведена су правила за интерпретацију појединих атрибута. Ова интерпретација се може огледати у томе

да се боја и дебљина линија које репрезентују сигнале мењају у зависности од вредности атрибута који се посматра. И овај приступ је захтевао захтева постојање класе која врши интерпретацију стања посматраних атрибута. На слици 68 су дати начини на које је сигнали током симулације интерпретирају у складу са вредностима које дигиталне компоненте могу да приме.



Слика 68: Интерпретација боје коју линија узима током симулације и дебљине коју линија има у зависности од вредности атрибута

Да би атрибут постао видљив потребно је потребно је одредити које атрибуте корисник жели до посматра током симулације. Поступак дохватања атрибута за приказ је дат на слици 69.



Слика 69: Додавање атрибута за праћење

Приликом праћења атрибута који описују понашање неког сигнала на магистрали обично поред вредности коју тај атрибут има у садашњем тренутку треба узети и читаву историју промена вредности атрибута. Дијаграм који се у овом случају приказује се састоји из x осе која репрезентује време и y осе која репрезентује вредност сигнала у појединим тренуцима. Ово је један од основних

начина за представљања вредности у симулатору. Пошто се овде ради о пројекту који барата само са дигиталним вредностима нема потребе нагласити како сигнал треба да изгледа јер је за дигиталне вредности развијен посебан мод за интерпретацију. Уколико би се симулатор користио у неке друге сврхе било би потребно нагласити у ком опсегу треба интерпретирати атрибут којом га бојом потенцирати и који део времена приказивати на једном екрану. На слици 70 је дат пример праћења већег броја временских сигнала.

Слика 70: Временски дијаграми праћених сигнала

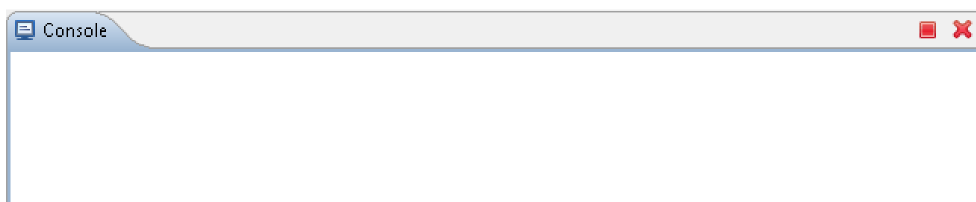
Следећи облик праћења стања симулираног система коју симулатор пружа је посматрање стање система у једном конкретном тренутку дато табеларно. Код овог приступа сваком атрибуту је додељено поље или скуп поља које посматра и о чијим променама обавештава посматраче. Овај приступ се најчешће користи у случајевима када се посматрају атрибути компонената, а ређе атрибути веза. На слици 71 је дат пример праћења стања система дат у табеларном облику.

Name	Value

Слика 71: Табеларно праћење стања система

Поред наведених приступа праћењу рада система постоји и могућност посматрања и сажети текстуални преглед читавог система или дела система у сваком назначеном тренутку. Овај приступ, за разлику од претходно назначених захтева интервенцију програмера приликом пројектовања симулираног система. Овај приказ је омогућен захваљујући постојању одговарајућег поља унутар симулатора у које се могу уписивати поруке које компоненте шаљу кориснику.

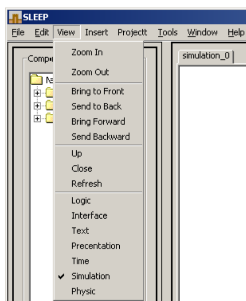
Уколико су компоненте писане користећи језик VHDL онда је испис у ово поље еквивалентан позиву методе write(). Овај начин праћења рада система се може користити у случајевима када се прати рад симулатора који је намењен коришћењу на лабораторијским вежбама где пројектант обавештава корисника о битним догађајима. Приликом исписа порука користећи овај вид комуникације са корисником поред саме поруке исписује се и извор поруке, то јест која компонента и у читавој хијерархији је генерисала поруку. Други вид када се ова опција може користити је у случају да се ради развој система па се остављају поруке које кориснику сигнализирају да је дошло или до проблема или до неких битних ситуација. Типови порука који су дефинисани у симулатору обухватају порука упозорења које указују на грешке у симулираном систему и порука која указује на грешке у раду симулатора. Да би се разликовале поруке приликом исписа користи се бојење тако да постоје три основне боје за поруке: црна за корисничке поруке, плава за поруке које сигнализирају на грешку у симулираном систему и црвене које сигнализирају да је дошло до грешке у симулатору. У овом пољу се могу исписивати и поруке које симулатор пружа о току извршавања симулације које представљају лог симулатора. Уколико се укључи опција да се поруке уместо у одговарајући лог фајл исписују овде онда у неким случајевима када се ради о симулираним системима који имају велики број компонента да комплетно прекрију ово поље и да кориснику учине отежано праћење симулације. Поруке које пропадају логу симулатора се исписују зеленом бојом. На слици 72 је дат приказ текстуалног извештавања о стању симулације на примеру симулатора процесора са проточном обрадом. Кориснику је омогућено да одабере који тип извештавања у овом пољу жели.



Слика 72: Конзола симулатора у којој је могуће текстуално праћење исписа стања система у појединим тренуцима

4.2.6 Задавање параметара конкурентне и дистрибуиране симулације

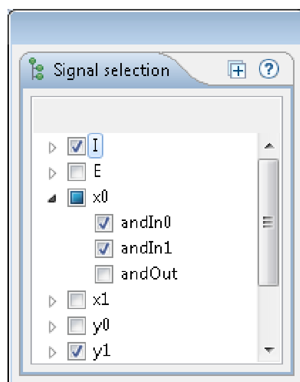
Приликом симулације рада одређеног дигиталног система потребно је дефинисати који алгоритам симулације ће се користити и у случају дистрибуираног извршавања и потребно је дефинисати и где ће се који делови датог система извршавати. Да би се ово омогућило потребно је прећи у посебан мод рада симулатора, то јест отворити посебну перспективу у којој је могуће одређивати која ће се компонента налазити на ком рачунару. Приказ овог мода рада симулатора је приказан на слици 73.



Слика 73: Мени за постављање параметара симулационог слоја

SLEEP симулатор подржава извршавања симулације користећи различите симулационе алгоритме укључујући рад са једном нити, рад са више нити и рад у дистрибуираном окружењу. Рад са више нити и у дистрибуираном окружењу захтева додатно конфигурисање, док рад са једном нити не захтева додатна подешавања. У случају извршавања у више нити корисник дефинише број нити и синхронизациони интервал. У случају дистрибуираних извршења корисник треба да дефинише и топологију мреже и компоненти мапирање. Топологија мреже се може дефинисати коришћењем опције Network Editor у оквиру рада са активном опцијом Simulation у главном менију. Network Editor омогућава кориснику да дефинише скуп чворова и њихових карактеристика, укључујући IP адресу и порт на коме слуша SLEEP сервер и радна станица. Приликом дефинисања мода рада односно алгоритма рада корисник се постепено води ка жаљеном алгоритму и окружењу користећи систем који наликује на коришћење чаробњака (wizard). Уколико се ради о извршавању код кога је потребно читаву симулацију извршити на једном рачунару унутар једне нити користећи основни алгоритам симулације

онда се ово вођење прекида. Ово је подразумевани вид рада симулатора. Уколико корисник пак одабере оптимистички или песимистички алгоритам извршавања симулације кориснику се нуди да одабере број делова на које симулирани систем треба да подели, а након тога и форма која кориснику омогућава да симулиране компоненте подели у целине. Поступак поделе компоненти на целине не приказан на слици 74.



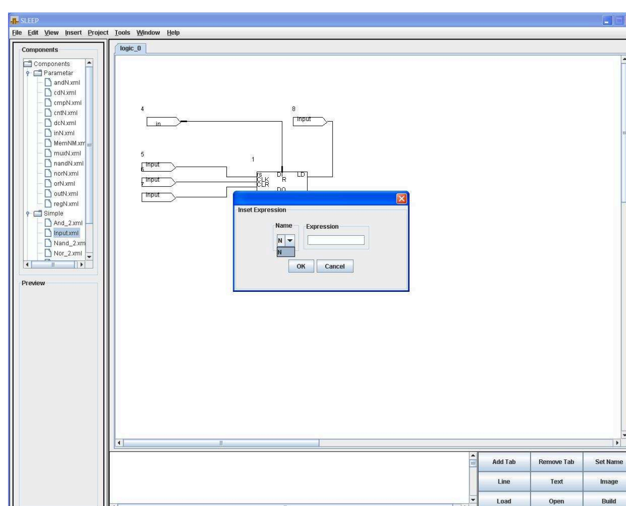
Слика 74: Одабир компонената за симулацију на специфицираном рачунару

Када је корисник поделио компоненте на целине има могућност да одреди да ли ће се дата целина извршавати на истом рачунару како и остале компоненте или ће се извршавање обавити на неком другом рачунару. Уколико се извршавање обавља на неком другом рачунару потребно је унети приступне параметре тих рачунара на којима се обавља дистрибуирана симулација. Ти параметри обухватају адресу и порт преко кога ће с комуникација обављати. Да би био могуће дистрибуирани извршавати симулацију потребно је подићи одговарајуће серверске процесе на поменутим рачунарима. Тек у том случају се стичу услови за дистрибуирану обраду симулације. Покретање симулације је исто као и у случају рада са једном нити притиском на дугме за покретање симулације у одговарајућем менију.

4.2.7 Задавање параметара физике компонената

Приликом креирања компоненти могуће је дефинисати и компоненту која описује физику дате логичке компоненте. За креиране логичке компонента може се дефинисати физика компоненте активирањем опције Physics Editor користећи одговарајућу опцију у главном менију. Physics Editor омогућава кориснику да дефинише скуп слушалаца за логичку компоненту која ће надгледати понашање

логично компоненте и која ће да креира одговарајуће извјештаје који се односе на физику компоненте (просечне / максималне / минималне дисипација, напон, струја, кашњења). Слушалац промене се дефинише користећи Јава програмски језик. Програм писан у Јави мора да имплементира предефинисане Јава интерфејсе. Овај приступ омогућава кориснику да прати различите аспекте логично понашање компоненте. За основне логичке компоненте дефинисана је једна компонента која описују физику понашања компонента када су у питању величине напона, струје, кашњења и дисипације. Да би се кратила поља која се обрађују пре почетка симулације је потребно поставити одговарајуће параметре који се односе на константе које су потребне како би се тражене физичке величине израчунале на одговарајући начин. Постављање ових параметара се обавља на идентичан начин као и постављање параметара логичке компоненте. Поступак постављања параметра је приказан на слици 75.



Слика 75: Постављање параметара физике компоненте

5 ЕВАЛУАЦИЈА РЕЗУЛТАТА

У овом поглављу су представљени резултати евалуације предложене решења пројектовања симулатора Архитектуре и организације рачунара са два становишта. Прво становиште се заснива на резултатима добијеним у настави Конкурентног и дистрибуираног програмирања где су студенти требали да пројектују дистрибуирани систем користећи већ развијене компоненте симулатора. Друго становиште се заснива на поређењу аналитичког модела рада симулатора Архитектуре и организације рачунара који се користи у настави са конкретним оптерећењима добијеним мерењима перформанси реалних симулатора.

5.1 Евалуација резултата симулатора добијених на основу коришћења у настави

У овој секције је представљен скуп лабораторијских вежби на којима је коришћен представљени симулатор у оквиру предмета Конкурентно и дистрибуирано програмирање. Лабораторијске вежбе су развијене сходно искуствима описаним у [62]-[70]. Представљени симулатор је коришћен више године у различитим облицима у настави, тако је у неким периодима коришћен као скуп библиотека које се користе у раду у конкурентном и дистрибуираном окружењу а које је потребно модификовати при томе не улазећи у структуру чему те библиотеке служе, у неким се користе као алат за тестирање развијених апликација, у неким као библиотека која обавља дистрибуирану обраду. Оно на шта треба посебно указати је да се симулатор користио у настави из поменутог предмета где су студенти имали за циљ да реализују конкурентну и дистрибуирану варијанту симулатора користећи секвенцијалну варијанту у дати низ алгоритама. Током периода од неколико година праћене су две групе студената, једна која није радила лабораторијске вежбе/пројекте и друга која је користила представљени симулатор или његове делове како би реализовала дистрибуирану апликацију. Евалуација је урађена са становишта постигнућа који су студенти остварили на испиту у зависности од тога да ли су пратили лабораторијске вежбе или не.

Оно што је специфично за лабораторијске вежбе је да студенти у истом

семестру када прате предмет Конкурентно и дистрибуирано програмирање прате и курс Архитектуре и организације рачунара 2 где се баве пројектовањем симулатора који се односе на ту област. Комбинујући знања из ове области студенти током лабораторијских вежби на предмету Конкурентно и дистрибуирано програмирање поред развоја дистрибуиране апликације исте принципе разматрају и са становишта модификовања симулатора. Током лабораторијских вежби, студенти се упознај како да модификују симулатор који има једну нит како би се омогућило покретање симулације у конкурентном и дистрибуираном окружењу. Тачније студенти развијају нове верзије симулатора које имају више нити и које су способне за рад у као дистрибуиране верзије симулатора. Поред тога, студенти се упознају како могу да користе симулатора као један од корака у процедури тестирања конкурентних апликација као саставни део методологије тестирања.

На основу препорука о избору области за лабораторијске вежби из предмета Конкурентног изнетих у секцији 2.2.1 „Преглед области Конкурентно и дистрибуирано програмирање“, а на основу фонда часова за поменути курс одабране су теме за четири лабораторијске вежбе као што је приказано у Табели 13. Студенти почињу са лабораторијском вежбом у вези са конкурентним програмирањем, затим следе две вежбе које се односе на дистрибуирано програмирање, и једна које се односи на мрежно програмирање, а друга која се односи на удаљене позиве метода у Јави, и завршне четврте вежбе која се односи на програмирање у гриду рачунара. Вежбе су тако организоване тако да се свака вежба надовезује на резултате претходних вежби што доводи до постепеног повећања сложености пројектованог система.

Ова секција представља SLEEP симулатор као основу за низ лабораторијских вежби и пројектних задатака. Свака лабораторија вежба даје преглед онога шта студенти треба да раде на вежбана и као и оно на чему често греше. Пројекат који студенти раде представља директну примену онога што је обрађивано на лабораторијским вежбама. Ова секција такође представља и методологију тестирања коју студенти користе током лабораторијских вежби и израде пројеката.

Табела 13: Покривањем одабраних тема предложеним лабораторијским вежбама

Одабране теме / Лабораторијске вежбе	Конкурентност					Дистрибуирани системи					
	Процеси и нити	Синхронизација	Алгоритми	Семафори	Монитори	Синхронизација	Алгоритми	Размена порука	Удаљени позиви процедура	Мрежно програмирање	Грид компјутинг
Конкурентно програмирање	+	+	+	+	+						
Мрежно програмирање	+	+	+			+	+	+		+	
Удаљени позиви метода		+			+	+	+		+		
Грид технологије								+	+	+	+

5.1.1 Организација лабораторијских вежби

Рад студената у лабораторији је организован у четири лабораторијске вежбе од којих свака траје по два и по сата. За сваку лабораторију вежбе студенти треба да развију посебан скуп класа, док су класе развијене на претходним вежбама, као и остатак кода симулатора запаковани у посебне архиве. У циљу покретања симулације који тестира њихове имплементације студенти морају да покрену развијене архиве у истом директоријуму како и добијене.

Читав сет лабораторијских вежби и пројеката се заснива на идејама из тренинга на основу суочавања са грешкама (the error management training) [102] и [103]. Овај вид обуке студената се заснива на принципима да је суштински део процеса учења то да студенти треба да се суоче са грешкама које могу да се појаве у приликом развоја и да науче како да се са изборе са таквим грешкама. Посебно, објашњавајући уобичајене грешке у конкурентном и дистрибуираном програмирању користећи Јаву помаже студентима да разумеју како да раде са нитима и како да их синхронизује, како да развију дистрибуиране апликације помоћу мрежног програмирања или удаљених позива метода (RMI).

У складу са идејама тренинга на основу суочавања са грешкама на лабораторијским вежбама се наглашава важност онога на чему се најчешће праве грешке и како те грешке отклонити. Листа најчешћих грешака на курсу конкурентног и дистрибуираног програмирања током лабораторијских вежби је дата у Табели 14. Вероватноће приказане у табели срачунате су појављивању грешака у студентским пројектнима и на испитима. Вероватноћа грешке означава

очекивани просечан број студената који ће да начини грешку док је радио на задатку који покрива одређену тип грешке. Списак представљен овде се нешто разликује од сличног списка [104] по томе што ставља већи нагласак на грешке специфичне за конкурентно и дистрибуирано програмирање. Честе грешке у случају конкурентног програмирања је недостатак обавештење да је стање неког објекта промењено за време мониторске операције. У случају дистрибуираног програмирања грешке које су последица неусклађености протокола, што за последицу има неспојиве типове података или неадекватни редослед у кораку унутар протокола, захтевају посебну пажњу.

Табела 14: Преглед најчешћих грешака у области конкурентног и дистрибуираног програмирања

Тип грешака			Вероватноћа
Конкурентно програмирање	1	Мртво блокирање - Deadlock	0.21
	2	Живо блокирање – Livelock	0.15
	3	Недостатак нотификације	0.21
	4	Погрешна мониторска дисциплина	0.12
	5	Ван опсега проблема	0.21
Дистрибуирано програмирање	6	Грешка у комуникационом протоколу	0.35
	7	Серијализација објеката	0.26
	8	Проблем раскида везе	0.28
	9	Грешке у комуникацији	0.19
	10	Грешка приликом удаљеном приступу објекту	0.09

У секцијама које следе за сваку лабораторијску вежбу ће бити дати описи проблем који се решава, опсег проблема и поступак решавање проблема. Следећу идеје тренинга на основу суочавања са грешкама описи проблема лабораторијских вежби су дати користећи минимално упутстава у циљу подстицања студената да активно истражују и експериментишу. У делу где се даје опсег проблема за сваку лабораторијску вежбу се посебно истиче које концепте конкурентног и дистрибуираног програмирања та вежба покрива. Поступак решавања проблема описује шта студенти најчешће раде приликом решавања датих проблема, а на чему обично грешке док покушавају да реше проблем.

Конкурентно програмирање

Опис проблема: Полазећи од SLEEP симулатора који се извршава у једној нити чија је архитектура дата на слици 22.а а чији је алгоритам извршавања дат на слици 32, развити верзију симулатора која има више нити чија је архитектура дата на слици 22.б а чији је алгоритам извршавања дат на слици 33.

Опсег проблема: У овој вежби, студенти уче основне концепте конкурентног програмирања као што су нит, синхронизација нити, поштанско сандуче и комуникација користећи поруке.

Решење проблема: Суштина ове лабораторијске вежбе је проблем "Произвођача-Потрошача". Студенти имплементира класу сандучића за поруке који има исти интерфејсом као и објекат реда који је на слици 33 (методе getMsg и putMsg). Класа сандучића је имплементирана као бафер коначног капацитета. Пример методе getMsg је приказан на слици 76.а. Недостатак обавештавања, подвучени код у примеру, (грешке 3 из Табеле 14) да је стање објекта коме је приступано промењено током наведене операције над монитором може да изазове мртво блокирање нити (грешка 1 из Табеле 14) или да проузрокује проблеме у извршавању симулације. Имплементација метода getMsg користећи браве, приказана на слици 76.б, је погодан пример за демонстрацију мртвог или живог блокирања нити (грешке 1 и 2 из Табеле 14). Погрешна манипулација са пољима која идентификују последњи елемент и број елемената у баферу може довести до грешке да проблем излази из опсега дефиниције (грешке 5 из Табеле 14) што значи да примена ради, али даје лоше резултате.

а)

```
public synchronized Msg getMsg(){

  while(size == 0){
    try {
      wait();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
  Msg msg = buffer[last];
  last = (last + 1) % buffer.length;
  size--;
  notifyAll();

  return msg;
}
```

б)

```
public Msg getMsg(){
  lock.lock();
  while(size == 0){
    try {
      full.await();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
  Msg msg = buffer[last];
  last = (last + 1) % buffer.length;
  size--;
  empty.signal();
  lock.unlock();
  return msg;
}
```

Слика 76: Имплементација методе getMsg за реализацију бафера коначног капацитета. (а) Користећи кључну реч synchronized; (б) Користећи браве Locks.

Верзија симулатора која ради са више нити а која је представљена на слици 22.б захтева имплементацију методе synchronize са слике 33. Метода synchronize се користи за синхронизацију нити у свакој итерације алгоритма извршавања и користи синхронизацију на баријери а њен код је дат на слици 77.а. Подвучени код дат на слици 77.б је пример коришћења погрешне мониторске дисциплине (грешка 4 из Табела 14). Грешка се јавља у случајевима када студенти не схватају разлику између дисциплине обавештавања „обавести и наставити“ (“signal and continue”), онако како је како је развијена у језику Јава, и дисциплине „обавести и сачекати“ (“signal and wait” или “signal and urgent wait”). У примеру се може видети да нит која стиже на баријеру као последњи у итерацији k умањује бројач блокираних нити на баријери и као први напушта ту баријеру. Та иста нит би могла да се врати на баријеру у итерацији k + 1, а све да су све остале нити и даље унутар баријере у итерацији k. Проблем настаје зато што нит прође баријеру без чекања да друге нити заврше итерацији k +1.

а)

```
public void synchronize() {
  synchronized (barrier) {
    barrier.count++;
    if (barrier.count==threadNum){
      barrier.count = 0;
      notifyAll();
    } else {
      try {
        wait();
      } catch (InterruptedException e){}
    }
  }
}
```

б)

```
public void synchronize() {
  synchronized (barrier) {
    barrier.count++;
    if (barrier.count==threadNum){
      notifyAll();
    } else {
      try {
        wait();
      } catch (InterruptedException e){}
    }
    barrier.count--;
  }
}
```

Слика 77: Имплементација методе за синхронизацију нити на баријери. (а) Исправно; (б) Погрешно.

Мрежно програмирање

Опис проблема: Полазећи од вишенитне верзије SLEEP симулатора реализоване у вежби Конкурентно програмирање, реализоване према архитектури са слике 22.б, развити дистрибуирану верзију симулатора са архитектуром приказаном на слици 22.в користећи мрежно програмирање у програмском језику Јава.

Опсег проблема: Циљ ове вежбе је да се студенти науче основне

концептима дистрибуираног програмирања, као и концепту активних монитора. Вежба укључује мрежно програмирање, клијент-сервер архитектуру, толеранцију на појаву грешке у случају неуспеха у раду мреже, дизајн протокола, као и верификацију компатибилности протокола.

```
public void run() {
	try {
		ServerSocket listener = new ServerSocket(port);
		while (running) {
			try {
				Socket client = listener.accept();
				processRequest(client);
			} catch (Exception e) {
			}
		}
		listener.close();
	} catch (Exception ex) {
		ex.printStackTrace();
	}
}
public void processRequest(Socket client) {
	new ServerThread(client, ID++, new ServerSideProtocol()).start();
}
```

Слика 78: Основна петља серверске нити задужена за пријем клијентских захтева

Решење проблема: Први део је решење се односи на имплементацију метода getMsg и putMsg клијентске стране апликације које су задужене за директну комуникацију са серверском страном. У другом делу решења студенти треба да имплементирају серверску страну апликације који ствара нит за сваки примљени клијентски захтев као што је приказано на слици 78. Исти проблем се опционо може решити креирањем нове класе која полиморфним позивом изведене методе processRequest не креира нову нит за сваки захтев већ користи базен нити које једну по једну упошљава.

Приликом пројектовања комуникационог протокола на апликативном нивоу између клијентске и серверске стране студенти треба да разматрају њихову комуникацију независно од имплементације начина транспорта порука. Класа која имплементира протокол на серверској страни прихвата клијентску операцију а након тога користи постојећу имплементацију класе сандучића (реализоване у вежби Конкурентно програмирање) како би извршила тражену операцију, као што је дато на слици 79. Честа грешка у дизајну протокола је недостатак опоравак од грешака изазваних различитим начинима на које учесници покушавају да комуницирају (грешка 6 из Табела 14). Још једна грешка се односи на проблеме

прекида комуникације узроковане слојем мреже или неправилностима у раду хардвера (грешка 8 из Табела 14). Оба грешке се решавају применом тактике одбрамбеног програмирања. Један од примера је и исказ try-catch који поседује finally грану за правилно ослобађање заузетих ресурса. Други пример је употреба else гране у за сваки if-else исказ.

```
public void conversation() {
	try {
		while (true) {
			String operation = communicator.readString();
			if ("PUT".equals(operation)) {
				Object data = communicator.readObject();
				buffer.putMsg(data);
				communicator.writeString("OK");
			} else if ("GET".equals(operation)) {
				Object data = buffer.getMsg();
				communicator.writeObject(data);
			} else break;
		}
	} catch (Exception e) {
	} finally {
		communicator.close();
	}
}
```

Слика 79: Протокол клијентске стране за прихватање клијентских захтева

Објекат који обавља комуникацију имплементира методе за комуникацију као што је приказано на слици 80.а. Имплементација користи Јавине библиотеке за рад са мрежом и размену објеката. Транспорт објеката се заснива на принципима серијализације структурираних типова података. Пример честа грешка која се јавља приликом серијализације објеката приказан је на слици 80.б (грешка 7 из Табела 14). До ове грешке може да дође када се користе токови података различитих типова над истом утичницом што може да доведе до неуспешне испоруке поруке и да доведе до мртвог блокирања.

<table>
<tr><td>

```
oin    =    new    ObjectInputStream(
            socket.getInputStream());

public Object readObject()
    throws CommException {
  try {
     return oin.readObject();
  } catch (Exception ex) {
    throw new CommException();
  }
}

public String readString()
    throws CommException {
  return (String)this.readObject();
}
```

</td><td>

```
bin      =      new      BufferedReader(
           new        InputStreamReader(
           socket.getInputStream()));

oin     =     new     ObjectInputStream(
            socket.getInputStream());

public Object readObject()
    throws CommException {
  try {
    return oin.readObject();
  } catch (Exception ex) {
    throw new CommException();
  }
}

public String readString()
    throws CommException {
  try {
    return bin.readLine();
  } catch (Exception ex) {
    throw new CommException();
  }
}
```

</td></tr>
<tr><td>а)</td><td>б)</td></tr>
</table>

Слика 80: Имплементације размене објеката. (а) Исправна; (б) Погрешна.

Дистрибуира верзија симулатора захтева додатну имплементацију методе за синхронизацију на баријери која је приказана на слици 33. У циљу подршке дистрибуираној синхронизацији на нивоу апликативног протокол потребно је проширити серверски протокол за обраду захтева и SYN опцијом. На овакав начин формирана серверска класа представља имплементацију активног монитора. Серверска страна протокол у оваквој имплементацији прихвата захтев за операцију и онда користи постојећу локалну имплементацију методе за синхронизацију на баријери (synchronize метода из вежбе Конкурентно програмирање) како би извршила операцију, као што је дато на слици 81. Фрагмент кода са Слике 81 проширује if-else исказ са Слике 79, и долази пре последњег else гране.

```
else if (“SYN”.equals(operation)) {
        local.synchronize();
        communicator.writeString("OK");
   }
```

Слика 81: Проширење протокола серверске стране ради дистрибуиране синхронизације. Ово проширење је потребно уметнути непосредно пре последње else гране на слици 79.

Удаљени позиви метода

Опис проблема: Полазећи од вишенитне верзије SLEEP симулатора реализоване у вежби Конкурентно програмирање, реализоване према архитектури са слике 33, развити дистрибуирану верзију симулатора са архитектуром приказаном на слици 34 користећи удаљене позиве метода (RMI) у програмском језику Јава.

Опсег проблема: Циљ ове вежбе је да научи студенте дистрибуираном програмирању користећи објектно оријентисани приступ. Креирање, повезивање и претрага објеката чине суштину приликом изучавања концепата приступа удаљеним објектима користећи удаљене позиве метода у јави (Java RMI). Концепт безбедности који је уграђен у програмски језик Јава је представљен користећи механизам безбедности заснован на коришћењу полиса.

```java
public class MsgBoxRMIClient
      extends MsgBox{

public MsgBoxRMIClient() {
  try {
    remoteBuffer      =      (MsgBoxRMIInterface)
Naming.lookup("rmi://rti29.etf.rs:1099/buffer");
  } catch (Exception e) {
    e.printStackTrace();
  }
}


public Msg getMsg() {
  try {
    return remoteBuffer.getMsg();
  } catch (RemoteException e) {
    throw new RuntimeException();
  }
}
```
а)

```java
public class MsgBoxRMIServer
      extends UnicastRemoteObject
      implements MsgBoxRMIInterface {
public        MsgBoxRMIServer()        throws
RemoteException {
  try {
    buffer = new MsgBox();
    LocateRegistry.createRegistry(1099);

Naming.bind("rmi://rti29.etf.rs:1099/buffer",
this);
    } catch (Exception e) {
   e.printStackTrace();
  }
}

public Msg getMsg() throws RemoteException {
  return buffer.getMsg();
}
```
б)

Слика 82: Део кода који представља имплементацију сандучића писаног користећи RMI. (а) Клијентска страна; (б) Серверска страна.

Решење проблема: Током израде ове лабораторијске вежбе студенти треба да користећи удаљене позиве метода у Јави (RMI) дефинишу класе клијентске и серверске стране које се понашају као посредник између симулатора и класа за рада са сандучићима развијеним и лабораторијској вежби Конкурентно програмирање, као што је приказано на слици 82. Класе клијентске страна, које су приказане на слици 82.а, имплементирају исти интерфејс као класа сандучића за поруке, а саме захтеве прослеђују класама на серверској страни, као што је

приказано на слици 82.б, који онда користе оригиналне класе сандучета за размену порука. Да би ово могло да функционише серверска страна треба да имплементира али и да огласи јавно видљивим удаљени интерфејс (MsgBoxRMIInterface) који омогућава клијентској страни да даљински приступите putMsg, getMsg и synchronize методама класе сандучета. Комуникација између клијента и сервера захтева од студената да редефинишу Јавину безбедносну политику са одговарајућим привилегијама задатим користећи фајлове са привилегијама за приступ удаљеним објектима.

Честа грешка које студенти приликом рада са RMI објектима праве је грешка приступа удаљеном објекту (грешка 10 из Табеле 14). Грешка се јавља у случајевима када студенти не схватају да ли повратна вредност коју враћа нека метода представља локалну копију удаљеног објект или је удаљена референца на објекат који се налази на удаљеном рачунару. Грешка настаје јер у оба случаја објекат треба да имплементира интерфејс Serializable док је у случају рада са удаљеном референцом објекат треба да се наслеђује класу UnicastRemoteObject. Још једна грешка која се јавља односи се на изузетке у комуникацији који се јављају у ситуацијама када серверска страна није доступна (грешка 9 из Табеле 14). Могуће решење које студенти могу да примене у решавању овог проблема је коришћење try-catch исказа ради обраде удаљених изузетака и емитује одговарајуће грешке.

Grid технологије

Опис проблема: За решења развијена у лабораторијским вежбама Мрежно програмирање и Удаљени позиви метода направити подршку за извршавање на грид мрежној инфраструктури.

Опсег проблема: Циљ ове лабораторијске вежбе је да студенте упозна са концептима који су развијени у грид инфраструктуром као што су апстракција посла, језик за опис послова (*.jdl датотека), брокера ресурса, евидентирању и чувању информација, елеменат за израчунавање, радним станицама, елементима за чување података.

Решење проблема: Студенти треба да направе по један посао за сваку реализовану имплементацију симулатора и да омогуће извршавање тих послова на гриду. Пример jdl датотеке за покретање независних послова који

специфицирају задатак који треба да буде извршен на гриду је приказан на слици 83. Послови не прецизирају тачну локацију где задаци треба да буду извршени и зато студенти морају да дефинишу фиксне почетну тачку комуникације, који ће се користити за успостављање комуникације између зависних задатака који су покренути на различитим радним станицама.

```
Executable = "/storage/exp_soft/java/jdk1.6.0/bin/java";
Arguments = " -jar /storage/exp_soft/drs/20000144/Worker.jar STUD144 4001
147.91.12.83 4003";
StdOutput = "message.txt";
StdError = "stderror.txt";
OutputSandbox = {"message.txt","stderror.txt"};
Requirements = RegExp("rti29.etf.bg.ac.rs*",other.GlueCEUniqueID);
```

Слика 83: Пример jdl датотеке која омогућава започињање независног посла на грид инфраструктури

Сви послови који се извршавају у оквиру једног грид сајта раде на различитим радним станицама и користе заједнички диск за смештање података. Ради отклањања проблема који могу да настану због проблема са различитим верзијама Јаве које су инсталиране на различитим радним станицама студенти користе могућност да Јаву инсталирају на једном заједничком месту на дељеном диску. На овај начин студенти избегавају грешке у вези са двосмисленостима које долазе из разлика у инсталацији и конфигурацији између чворова са радним станицама. Студенти науче да ни не морају да инсталирају одговарајуће окружење већ да могу у оквиру посла да проследе комплетну инсталацију.

5.1.2 Поступак тестирања

Током лабораторијске вежбе пажњу студената се константно усмерава на листу најчешћих грешака. Нажалост, студенти почињу да разумеју грешке тек када се сусретану са њима, обично у току фазе тестирања пројекта. Традиционалне методе тестирања не дају сваки пут задовољавајуће резултате у случају да треба се примене на тестирање конкурентних и дистрибуираних програма. Разлог за ово лежи у чињеници да се традиционалне методе заснивану на томе да је одзив излаза на улазне побуде увек исти, што код кода који се извршава у паралели са неким другим не мора да буде случај јер излаз у том случају може да зависи од редоследа по коме су се ти делови кода извршавали један у односу на други. Употреба алата за тражење грешака (debugger) или инструментација кода у виду убацивање редова који приказују стање објеката

штампањем порука на стандардни излаз може да сакрије грешке у синхронизацији. Штавише, неке грешке се не манифестује у сваком извршавања програма већ се појављују само при одређеним брзинама извршавања. У циљу да се студенти упознају са техникама тестирања, предложен је поступак који комбинује мануелно, формално и функционално тестирање.

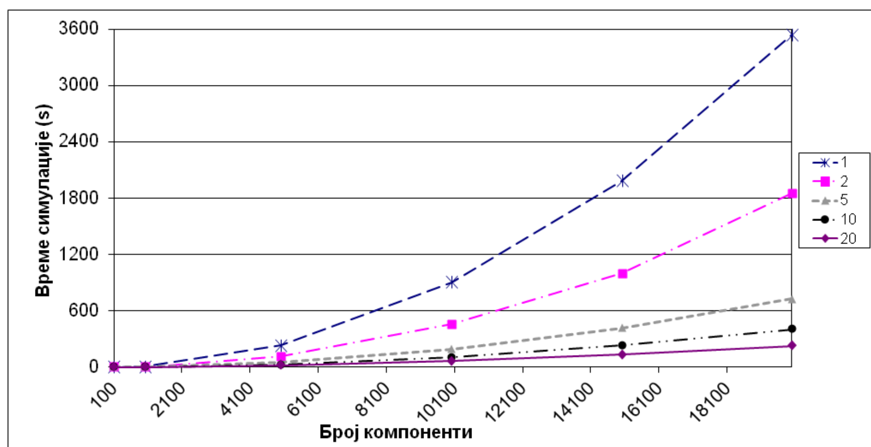
Током фазе мануелног тестирања, студенти раде у паровима и прегледају једни другима програме. Гледајући у програм који је написала особа у пару студенти једни другима помажу да схвате шта су најчешће грешке и како да превазиђу сличне грешке у свом коду. Штавише, међусобно преиспитивање исправности програма побољшава укупну квалитет програма у смислу читљивости и одрживости [105]. Циљ формалног тестирања је да се помогне студентима да тестирањем избегну недетерминизам који би могли довести до мртвог или живог блокирања у конкурентним апликацијама. За сваку лабораторију вежбу су припремљени тестови који користе JPF (Java Pathfinder) алат за формалну верификацију кода [106]. За случај вежбе из дистрибуираног програмирања алат JPF се допуњује са NetStub библиотекама [107]. Формално тестирање пружа могућност да се поред синхронизационих грешака уоче и грешке које проистичу из појаве изузетака приликом комуникације. Функционално тестирање има за циљ да допуни формално тестирање у домену ван обима грешке за које формално тестови имају тенденцију да трају сувише дуго. Типичан скуп грешака који се могу отклонити овом техником представљају грешке погрешног досега проблема које настају када програм ради, не блокира се, али резултати нису онакви какви се очекују. Овај тип грешака се испитује тако се користи симулатор који има само један ток контроле и симулира детерминистичку мрежу, а онда се његови излази пореде са резултатима добијеним на вишенитној конкурентној и дистрибуираној имплементацији симулатора. Овакав вид тестирања омогућава да се са великом вероватноћом уочи да грешка постоји зато што симулатор поседује велики број приступа у различитим секвенцама приступања, али овај приступ не омогућава да се тачно одреди место где је грешка настала. Предложена процедура тестирања конкурентних и дистрибуираних апликација се састоји у комбиновању мануелног, формалног и функционалног испитивање које може бити од велике помоћи

студентима приликом рада на пројекту.

5.1.3 Студентски пројекти

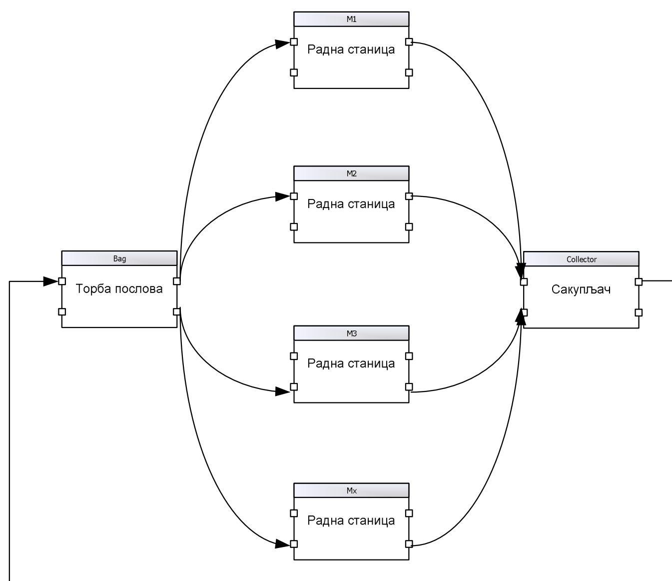
Након завршетка лабораторијских вежби студенти почињу да раде на не обавезним пројектима. Пројекти су по свом обиму изнад нивоа сложености лабораторијских вежби, служе да би студенти показали целокупно знање које је стечено на курсу. Сви пројекти имају идентичне структуре, али се баве различитим проблемима синхронизације и дистрибуиране обраде. Пројектни задаци се дефинишу три пута годишње, а примери неки пројеката укључују су дистрибуиране симулације анализе шема дигиталних логичких уређаја, проблем кретање n тела у гравитационом пољу, алгоритме за разбијање лозинки и дистрибуирану обраду слике. Студенти развијају дистрибуиране апликација на основу библиотека SLEEP симулатора, а при томе примењују симулационе алгоритме који су дати на сликама 33 или 34. Након примене датих техника од студената се очекује да стечена знања испробају тако што развијене апликације тестирају у дистрибуираној грид инфраструктуру. Примери одрађених студентских пројеката се могу наћи на [108].

Од студената се очекује да тестирају скалабилност и оптерећење мреже које настаје задавањем различитих параметара у симулацији имплементираних решења. Ово се тражи како би студенти схватили ограничења које паралелизација може да наметне као и како кашњења која постоје у систему утичу на брзину рада. Тест окружење на коме студенти могу да раде је AEGIS05-ETFBG грид сајт са 32 радна чворова (од којих се највише 20 користило за пробне сврхе). На рачунарима је инсталиран Scientific Linux 4.8 на коме је подигнута грид инфраструктура средњег слоја gLite 3.1. Сваки чвор има једнопроцесорски систем са 2 GB RAM меморије, 25 GB и слободног простора на диску. Чворови су међусобно повезани преко 100 Mb/s LAN мреже која има топологију звезде. Чворови деле home директоријум и као алтернативни начин комуникације могу међусобно да комуницирају користећи MPI или заједнички, дељени, диск. Грид сајт је део SEE-GRID инфраструктуре [109].



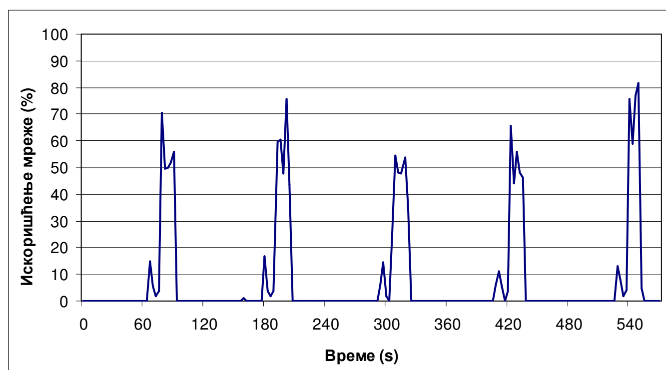
Слика 84: Време извршавања симулације у зависности од броја елемената које се симулирају и од броја доступних радних станица

Студенти тестирају скалабилност својих решења тако што мере време потребно за извршења одређеног посла у конфигурацији која се састоји од различитог броја радних станица. Пример време извршења симулације за проблем кретање n тела у гравитационом пољу користећи конзервативни алгоритам симулације дат на слици 85. Са слике 84. се види да је време извршавања у обрнутој сразмери са бројем радних станица, али да постоје извесна одступања. Разлози за одступања која се јављају је нешто што студенти треба да науче током развоја апликације радећи на пројектном задатку. Један од разлога за одступање лежи у чињеници да предложено решење није идеална "торба послова", због међузависности између симулиране компоненте. Свака радна станица има посебан торбу послова коју је потребно синхронизовати са другим чворовима, као и са централним сервером на крају сваке итерације. Други разлог за одступање лежи у ограниченом пропусном опсегу мреже у којој се ови послови извршавају. Трећи разлог је коришћење централног сервера, као фиксне комуникационе тачку која онемогућава оптимистичну симулацију извршавања и дефинише времена интеракције према карактеристикама најспоријег чвора који ради обраду послова.



Слика 85: Конфигурација компонената које омогућавају примену “Hearth Beat” и конзервативног алгоритма симулације

Студенти тестирају оптерећења мреже која производе њихова решења у циљу да одреде однос између времена комуникације и времена обраде и да ли постоји зависност између њих. Решења су обично засноване на “Hearth Beat” алгоритму који доста личи на конзервативни алгоритам симулације. Овај алгоритам је примењен на систем представљен на слици 86 чиме су ова два алгоритма стопљена у један. Пример оптерећења мреже са конфигурацијом датој на претходној слици која користи централни сервер за размену порука између радних станица приликом решавања проблема кретања n тела у гравитационом пољу је дат на слици 86. Оптерећења мреже је периодична и унутар сваке итерације могу се уочити два врха. Већи од ова два врха је последица чињенице да централни сервер шаље нове податке свим радним станицама на почетку сваке итерације. Ниже врх је последица чињенице да радне станице шаљу резултате израчунавања на централни сервер, и ти подаци су расподељени током временског интервала који чине само мала количина података у односу на примљене податке. Тестирање оптерећења мреже треба да помогне студентима да схвате да мрежна комуникација може доста да ограничава употребљивост дистрибуираних програма.



Слика 86: Мрежно оптерећење приликом коришћења конфигурације која одговара hearth beat алгоритму и конзервативном алгоритму симулације код коришћења SLEEP библиотека за симулацију проблема кретања n тела у гравитационом пољу

5.1.4 Евалуација коришћења симулатора и библиотека

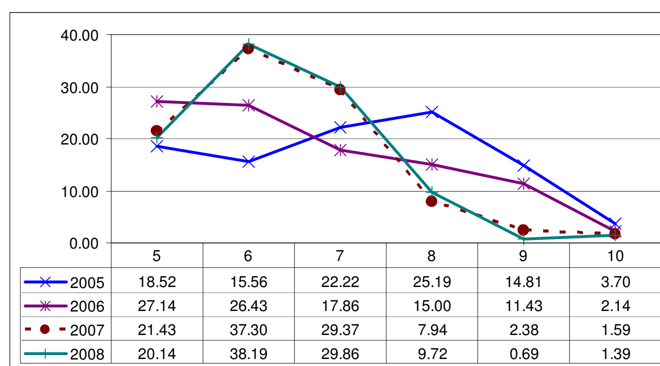
У овој секцији је дата евалуација коришћења SLEEP симулатора и његових библиотека у настави из Конкурентног и дистрибуираног програмирања. Приликом развоја симулатора утрошено је преко десет човек месеци како би се развили сви делови. У циљу процене предложеног коришћења SLEEP симулатора спроведена је квалитативне и квантитативне анализа. Евалуација разматра период од 2004 до 2008 у коме је рађено увођење делова симулатора и његово почетно коришћење. У школској години 2004/05 овај симулатор и лабораторијске вежбе нису постојале, школске 2005/06 првој верзији SLEEP је пројектована и уведена у лабораторију, школске 2006/07 уведена је нову верзију лабораторијских вежби са проширеном листом најчешћих грешке у свакој вежби, док је школске 2007/08 нова верзија лабораторијских вежби са коришћењем грид технологија уведена. Школске 2008/09 и 2009/10 није било значајнијих промена у концепту лабораторијских вежби.

На крају сваке школске године, Електротехнички факултет Универзитета у Београду, спроводи квалитативна евалуација. Евалуација се спроводи као обавезна и анонимна анкете на коју морају да одговоре сви студенти који су у претходних годину дана положили наведени курс. Анкета се спроводи пре уписа у наредну годину студија. Одговарањем на питања студенти процењују квалитета курсева које су пратили. Сваки курс се може нумерички оценити оцењује као одличан, врло добар, добар, довољан, и незадовољавајући. Током периода посматрања скоро 600 студената је учествовало у анкети за предмет конкурентно

и дистрибуирано програмирање. Резултати показују да просечна оцена за овај курс променила са "добар" да "врло добро", иако области које су изучаване као и наставници на курсу нису мењани током периода посматрања. Просечне оцене на другим курсевима на истом нивоу нису значајније варирали у току периода посматрања. Осим нумеричког оцењивања, студенти имају прилику да дају своје личне коментар о курсу. Иако анкета не обухвата питања о лабораторијским вежбама у својим коментарима студенти су оценили лабораторијске вежбе као веома корисне.

Статистичке методе су коришћене приликом квантитативне процена у посматраном периоду. Посматране су две групе студената. Прва, експериментална, група је учествовала у лабораторијским вежбама и изради пројеката, а друга, контролна, није. Наставници на курсу нису форму групе, већ је било дозвољено да студенти својевољно бирају групу којој ће припадати. Студенти нису били свесни чињенице да ће њихова постигнућа бити анализирана. Слике 87 и 88 показују успех студената на курс у посматраном периоду. Успех студената се оцењивао оценама из интервала од 5 до 10, где је оцена 5 била намењена студентима који су били успешни у мање од 50% градива на предмету.

Слика 87 показује успех контролне групе. Пошто описани систем није био у употреби у 2005, а пошто је корелација између сваке посматране године на графу између 0.72 и 0.99 може се закључити да ниво знање које студенти примају током посматраног периода није варирао.

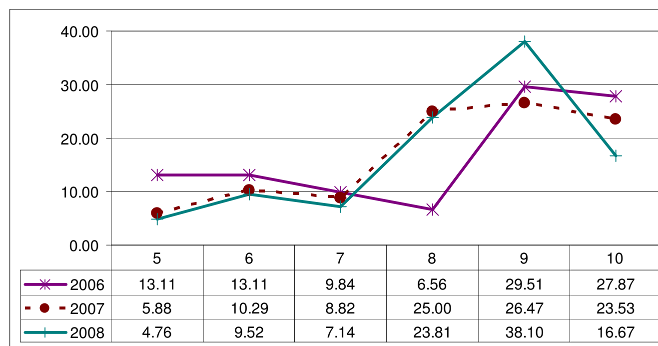


	5	6	7	8	9	10
2005	18.52	15.56	22.22	25.19	14.81	3.70
2006	27.14	26.43	17.86	15.00	11.43	2.14
2007	21.43	37.30	29.37	7.94	2.38	1.59
2008	20.14	38.19	29.86	9.72	0.69	1.39

Слика 87: Успех контролне групе током посматраног периода 2005-2008

Слика 88 приказује успех студената који су у посматраном периоду учествовали у лабораторијским вежбама. Поређење школске године 2005/06, 2006/07, 2007/08 и може се уочити да са сваке године број студената који су

прешли праг од 50% и постигла више оцене повећава.



	5	6	7	8	9	10
2006	13.11	13.11	9.84	6.56	29.51	27.87
2007	5.88	10.29	8.82	25.00	26.47	23.53
2008	4.76	9.52	7.14	23.81	38.10	16.67

Слика 88: Успех експерименталне група током посматраног периода 2006-2008

За анализу успеха студената коришћена је дво факторска ANOVA анализа. Први фактор у анализи је било посматрање година, са три нивоа (2006, 2007, 2008). Други фактор је било посматрање група студената која је учествовала у испитивању, са два нивоа (експериментална, контролна). Зависна промењљиве у овој тесту је била оцена коју су студенти посматране групе постигли у посматраној години.

Главни ефекат фактора групе посматрања је значајан (F (1, 575) = 210.244, p = 0.0000), што значи да постоји разлика између експерименталне и контролне групе. Главни ефекат године фактор године посматрања није статистички значајна (F (2, 575) = 0.05835, 0.94333), што значи да нема разлике у достигнућима током година. Ово значи да проширење листе најчешћих грешака и увођење грид технологија није утицало на постигнућа студената. Објашњење ове тврдње можда лежи у чињеници да после одређеног броја објашњених карактеристичних грешака долази до засићења и студенти не могу да прате толике промене. Интеракција између два фактора није статистички значајна (F (2, 575) = 1.61090, p = 0.20061), што значи да је разлика између експерименталне и контролне групе не зависи од посматрања године. Такође, може се рећи да је профил онога што студенти остваре током година иста без обзира да ли се добија из експерименталне или контролне групе.

Поређење експерименталне и контролне групе током периода посматрања показују да је тачка пресека између резултата контроле и експерименталне групе је на оцени 8, то јест, број студената са вишим оценама расте док број студената са нижим оценама опада. Такође је утврђено да је број студената који су нису

прешли праг 50% смањен. Оно што се може приметити је да су студенти експерименталне групе на испиту у просеку имали за 0,61 већу просечну оцену од просечне оцене коју су остварили студенти контролне групе.

5.2 Евалуација резултата добијених на основу експерименталних резултата и аналитичког модела

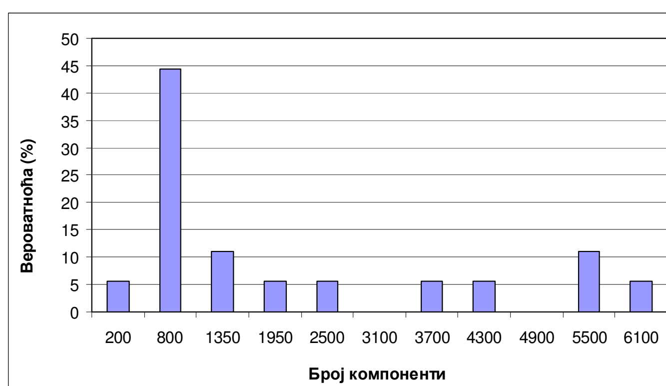
У овој секцији су представљени експериментални резултати добијени приликом рада са симулаторима архитектуре и организације рачунара. Секција је подељена у два дела. Прво део чине резултати мерења добијени на постојећим симулаторима са циљем утврђивања карактеристичног оптерећења. Други део чине експериментални резултати рада на предложеном симулатору сходно представљеном моделу и карактеристичном оптерећењу. Кроз ово је представљено очекивано понашање симулатора приликом рада у конкурентном и дистрибуираном окружењу и могући добици на перформансама у односу на симулатор који се извршава унутар једног тока контроле. Ово је демонстрирано користећи наменски развијено окружење и компоненте који могу да генеришу генеричке конфигурације са параметрима који одговарају реалним симулаторима. Овакав симулатор треба да омогуће праћење и опис перформанс са знатно више детаља у односи на реалне симулације, реални симулације моду да дају само коначан скуп шема са тачкасто распоређеним тест параметрима, а користећи генерички омогућен је унос ширег скупа улазних параметара. Овакав симулатор генерише карактеристично оптерећење у опсегу у коме могу да се варирају параметри који су издвојени у секцији 2.1.2 „Преглед постојећих симулатора из области Архитектуре и организације рачунара“, 2.1.3 „Преглед карактеристика симулатора“.

5.2.1 Експериментални резултати

У овој секцији су представљени експериментални резултати добијени током симулације користећи SLEEP као и карактеристична оптерећења која се могу даље користити за проверу аналитичког модела. Тест окружење на коме су испитиване перформансе SLEEP симулатора представљао је грид сајт AEGIS05-ETFBG са 32 чвора која могу да раде паралелну обраду (20 се користило за

потребе овог тестирања док су преосталих 12 коришћени за друге научне симулације). На овим рачунарима је инсталиран 32-битни Scientific Linux 4.8 на коме је додатно инсталиран средњи слој gLite 3.1 као и Јава 1.6. Свака радна станица поседује једнопроцесорски систем са 256KB кеш меморије, 2 GB RAM меморије и 25 GB слободног простора на диску за потребе смештања међурезултата. Чворови су повезани преко 100 MB/s LAN мреже која има топологију звезде. Чворови деле home директоријум и могу, уколико је то некој апликацији потребно, да комуницирају једни са другима и преко MPI интерфејса или користећи дељени диск. Овај Грид сајт је део SEE-GRID/AEGIS инфраструктуре [109].

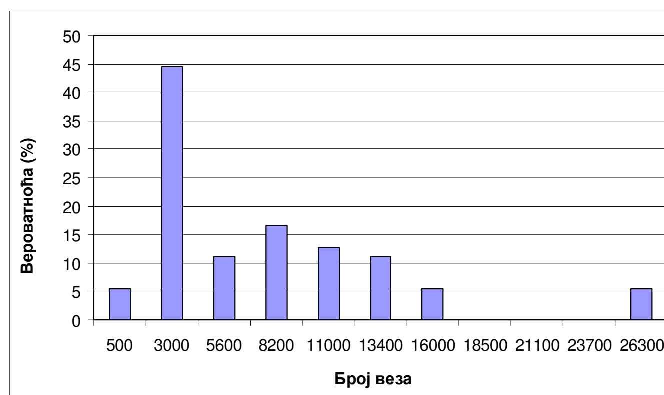
Приликом анализе симулатора из области архитектуре и организације рачунара посебна пажња је била посвећена нумеричким карактеристикама које су имали параметри симулације који су били посматрани у секцији 3.2 „Аналитички модел алгоритама симулација". На основу спроведене анализе може се закључити да број компоненти у симулаторима који се користе у области архитектуре и организације рачунара варира у границама од 200 до 6000. На основу ових мерења се може закључити да је расподела времена обраде компоненте имала просечну вредност од 1908.39, стандардну девијацију 1959.66 и коефицијент спљоштености (куртосис) -0.36. Експериментално добијена расподела је дата су на слици 89.



Слика 89: Расподела броја компонената унутар симулатора у области Архитектуре и организације рачунара посматраних у интервалу од 200 до 6000

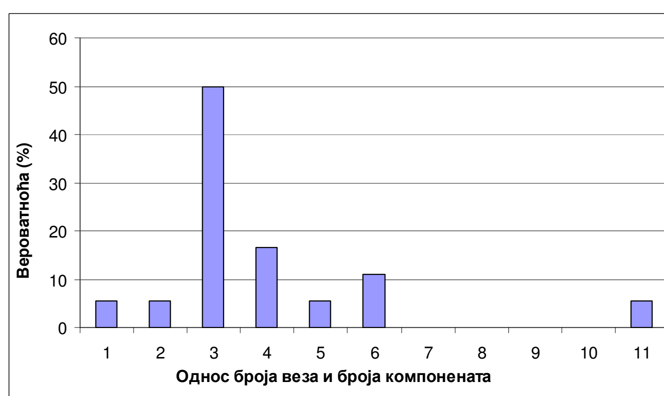
На истом скупу симулатора спроведена је и анализа броја веза које постоје у симулаторима који се користе у области архитектуре и организације рачунара варира у границама од 400 до 25000, са хистограмом расподеле који је дат на слици 90. На основу ових мерења се може закључити да је расподела времена

обраде компоненте имала просечну вредност од 6010.33, стандардну девијацију 6881.07 и коефицијент спљоштености 3.35.



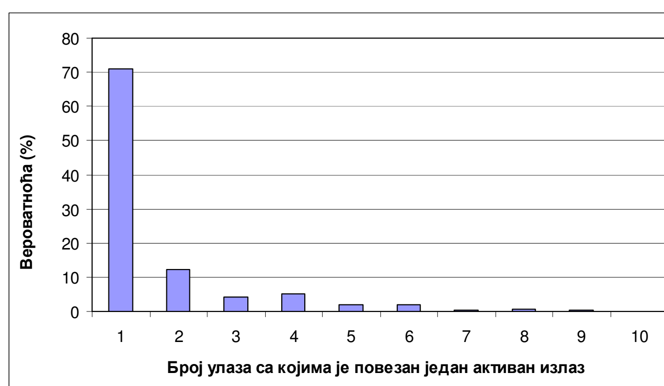
Слика 90: Расподела броја веза унутар симулатора у области Архитектуре и организације рачунара посматраних у интервалу од 400 до 25000

Параметри број компонената и број веза унутар шеме су се посматрале на нивоу шеме и није се улазило у њихову унутрашњу структуру по симулаторима, јер је нису ни имали, али је зато таква анализа била неопходна за случај броја компонената са којима су повезани излази једне компоненте. На основу спроведене анализе може се закључити да се однос број веза и броја компонената по анализираним шемама креће од 1 па максималног броја. Овај максимални број одговара броју секвенцијалних мрежа шеме која се симулира и одговара вези генератора такта и наведених секвенцијалних мрежа. На основу ових мерења се може закључити да је расподела броја веза по једној компоненти имала просечну вредност од 3.06, стандардну девијацију 1.23 и коефицијент спљоштености 8.66. Расподела експерименталних резултата је дат на слици 91.



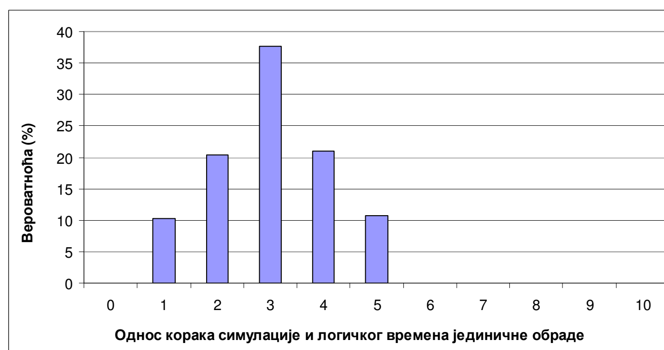
Слика 91: Расподела односа броја веза и броја компоненти унутар симулатора у области Архитектуре и организације рачунара

Параметри број улазних портова са којим је повезан неки излазни порт се посматрале на нивоу шеме. На основу спроведене анализе може се закључити да се однос број улазних портова по излазном порту преко кога се шаље догађај креће од 1 па до укупног броја компоненти. Овај максимални број одговара броју секвенцијалних мрежа шеме која се симулира и одговара вези генератора такта и наведених секвенцијалних мрежа. На основу ових мерења се може закључити да је расподела броја веза по једној компоненти имала просечну вредност од 1.66, стандардну девијацију 3.23 и коефицијент спљоштености 45.73. Расподела експерименталних резултата је дат на слици 92.



Слика 92: Расподела броја улазних портова повезаних на излазни порт унутар симулатора у области Архитектуре и организације рачунара

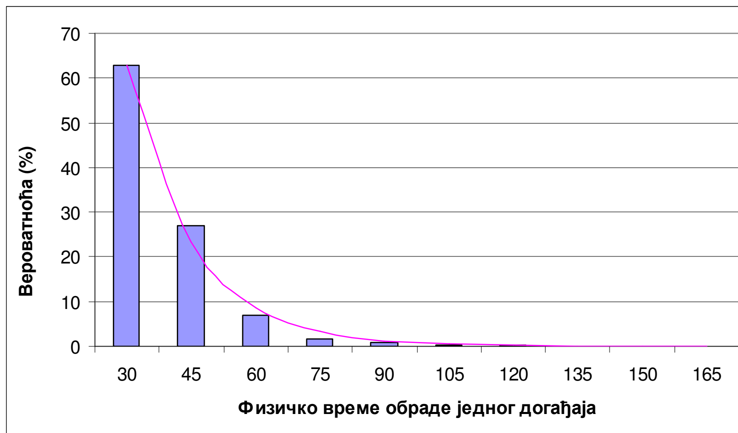
У изразима који описују понашање симулираног система се јављају изрази који представљају однос посматраног корака симулације и јединичког логичког времена ограде. Однос ова два параметра је доста сложен и зависи од конкретне имплементације симулираног система. Ког симулатора у области архитектуре и организације овај однос зависи од дубине логичке шеме, али и намене у области која се посматра. На основу спроведене анализе може се закључити да се однос корака симулације и интервала логичке обраде од 1 па до бесконачности. Овај максимални број задатих итерација симулације и дубине анализиране мреже. На основу ових мерења се може закључити да је расподела имала просечну вредност од 3.01, стандардну девијацију 1.12 и коефицијент спљоштености -0.60. Расподела експерименталних резултата је дат на слици 93.



Слика 93: Расподела корака симулације и јединичног времена обраде унутар симулатора у области Архитектуре и организације рачунара

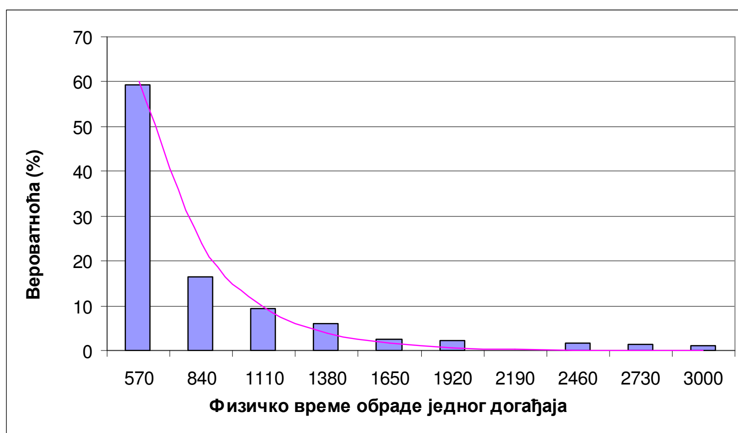
Када се посматра физичко времена обраде једног логичког догађаја код постојећих симулатора мора се посматрати тај период унутар саме симулације, односно колико тај временски параметар варира унутар те једне симулације. На основу анализе могу се уочити три карактеристична случаја. Један који одговара симулацијама дигиталних система састављених искључиво од компонената на нивоу логичких колам, други који одговара симулацијама система састављених искључиво од функционалних јединица, и трећи који одговара симулацијама читавих система.

Код симулатора архитектуре и организације рачунара који симулирају понашање система састављених искључиво од логичких кола добијени су резултати који су приказани на слици 94. На основу ових мерења се може закључити да је расподела времена обраде компоненте имала просечну вредност од 32.02, стандардну девијацију 37.65 и коефицијент спљоштености 360.10. Оваква расподела ова три параметра указује на то да се код анализираних система време обраде једног догађаја може моделовати експоненцијалном расподелом средњим вредношћу 32. Сумарни приказ експерименталних резултата је дат на слици 94.



Слика 94: Расподела физичког време обраде једног догађаја код система састављених искључиво од логичких кола

Код симулатора архитектуре и организације рачунара који симулирају понашање система састављених од функционалних јединица добијени су резултати који су приказани на слици 95. На основу ових мерења се може закључити да је расподела времена обраде компоненте имала просечну вредност од 864.44, стандардну девијацију 1890.96 и коефицијент спљоштености 684.16. Оваква расподела ова три параметра указује на то да се код анализираних система време обраде једног догађаја може моделовати сумом константне и експоненцијалне расподеле. Сумарни приказ експерименталних резултата је дат на слици 95.



Слика 95: Расподела физичког време обраде једног догађаја код система састављених од функционалних јединица

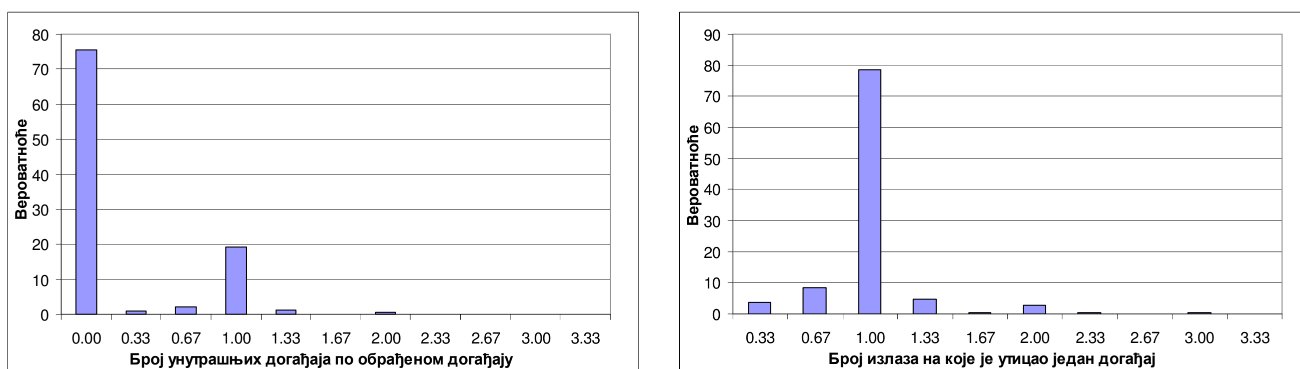
Код симулације понашање читавих система ког које су компоненте

комплексни подсистеми може посматрати интеракција која постоји приликом симулације кретања тела у гравитационом пољу које представља другу групу симулатора спроведене је анализа која доводе до закључка да време обраде веома мало осцилује око средње вредности. Приликом ових анализа дошло се до расподеле који дају карактеристике времена обраде које су приказане на слици 95. Време обраде се односи на израчунавање релативног помераја приликом интеракције два тела у гравитационом пољу у систему који ке чинило већи број тела. На основу ових мерења се може закључити да је расподела времена обраде компоненте имала просечну вредност од 398.33, стандардну девијацију 8.64 и коефицијент спљоштености 2.26. Оваква расподела ова три параметра указује на то да је код анализираних система време обраде једног догађаја приближно константно и може се моделовати средњим вредношћу 400. Сумарни приказ експерименталних резултата је дат на слици 96.

Слика 96: Расподела физичког време обраде једног догађаја приликом симулације кретања тела у гравитационом пољу у SLEEP симулатору током симулација

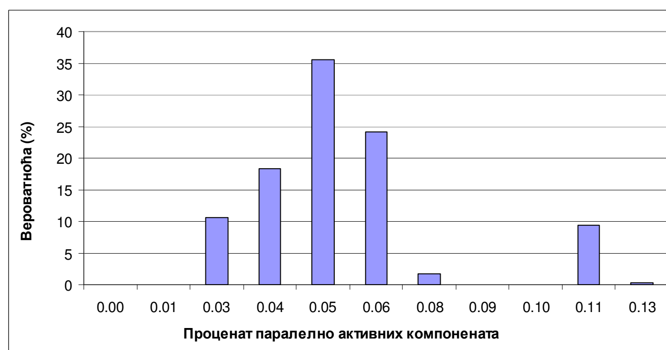
Када се посматра вероватноћа да се по иницијалном догађају генерише нови догађај мора се посматрати извршавање симулације и утврдити колико износи број догађаја који су иницирали креирање барем једног новог догађаја. На основу анализе могу се уочити два карактеристична случаја. Један који одговара ситуацијама када се не генерише ни један догађај и друга када се генерише више догађаја на различитим излазима дигиталне компоненте. Код симулатора архитектуре и организације рачунара добијени су резултати који су приказани на

слици 97. На основу ових мерења се може закључити да се у 74.72% случајева не генерише ни један нови догађај. Нови догађај се генерише са вероватноћом од 25.28%. Број излаза на које ће бити послати нови догађаји је приказан на истој слици од б). Расподела број излаза је имала просечну вредност од 1.03, стандардну девијацију 0.27 и коефицијент спљоштености 8.55. Оваква расподела ова три параметра указује на то да се код анализираних система број излаза на које је утицао један примљени догађај може моделовати константом вредношћу 1.



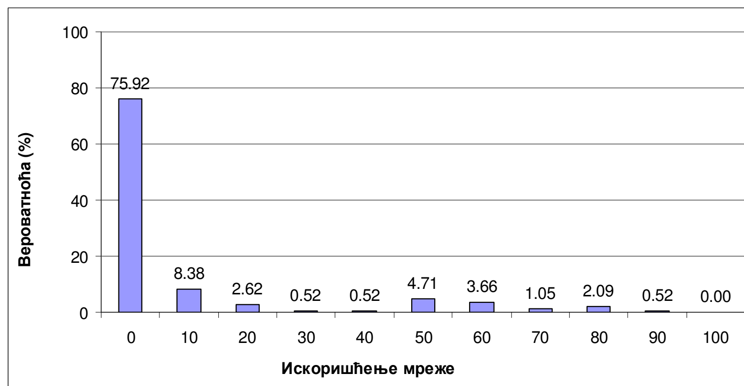
Слика 97: Расподела вероватноћа да се по иницијалном догађају генерише нови догађај (а), број излаза на које је утицао један примљени догађај (б)

Проценат паралелно активних компонената је сложен параметар који описује колико се паралелних догађаја обрађује а који су иницијално креирани. Овај параметар обједињује интеракцију више параметара као што су број секвенцијалних компонената, број комбинационих компонената, број веза излазног поста са повезаним улазним портовима, дужину логичке обраде једне компоненте, дужину трајања једног такта, максимални ниво пропагације сигнала у симулираних мрежи. Код симулатора архитектуре и организације рачунара добијени су резултати који су приказани на слици 98. На основу ових мерења се може закључити да се проценат паралелно активних компонената имала просечну вредност од 0.0407, стандардну девијацију 0.01 и коефицијент спљоштености 0.56.



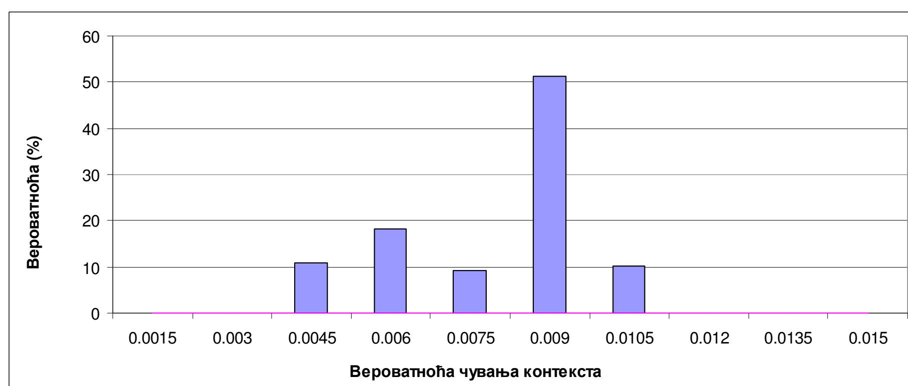
Слика 98: Проценат паралелно активних компоненти код симулатора у области Архитектура и организација рачунара

На основу спроведене анализе може се закључити да број компоненти у симулаторима који се користе у области архитектуре и организације рачунара варира у области од 200 до 6000 а ове компоненте су биле приближно равномерно распоређене по доступним радним станицама. Имплементирани симулациони алгоритми омогућавају SLEEP симулатору да обави симулације у дистрибуираном окружењу. Скалабилност и оптерећења мреже тестирано је користећи низ логичких шема које су биле генерисане за потребе тестирања. Тест оптерећења мреже се вршио да би се утврдило у којој мери мрежа представља ограничавајући фактор у симулацијама. Хистограм оптерећења мреже на централни сервер за тест који се састојао у симулацији шеме која садржи 2000 компоненте које ду равномерно дистрибуиране на 20 радних станица је дат на слици 99. Оптерећења мреже показује да приликом рада мреже постоје три критичне тачке која одговарају трима врховима на хистограму. Први врх је у нули и одговара времену обраде догађаја на радним станицама, у овом случају мрежа се налази у стању мировања и тада нема обраде. Други врх је последица чињенице да радне станице шаљу резултате прорачуна на централни сервер у виду малог броја догађаја што одговара малој количини података, што одговара искоришћену мреже од око 8%. Овај други врх одговара времену транспорта поруке између ране станице и сервера tk/tp. Трећи врх је последица чињенице да централни сервер шаље нове догађаје свим радним станицама на почетку итерације, доносећи коришћење мреже до 60%.



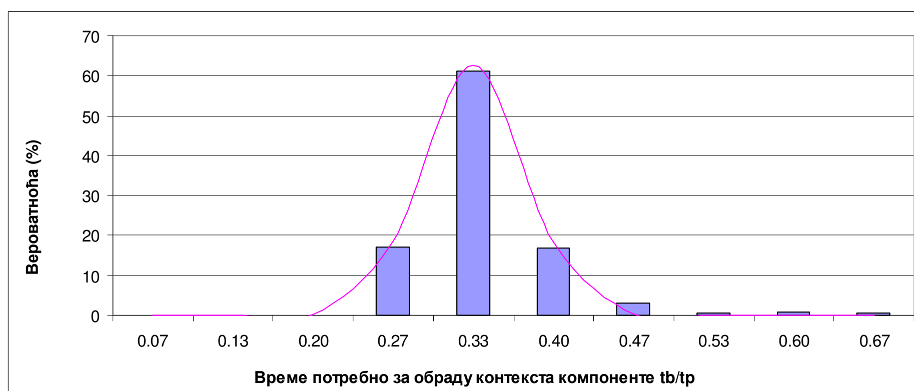
Слика 99: Расподела искоришћења мреже приликом коришћења SLEEP симулатора током тестирања

Вероватноћа чувања контекста зависи од имплементације алгоритма који се користи, као и од карактеристика симулираног система. Пошто посматрани симулатори нису имали оптимистични приступ симулацији узето је да се контекст чува у сваком синхронизационом интервалу. За синхронизациони интервал је узет један такт, тако да се вероватноћа чувања контекста добија на основу свих обрађених догађаја и обрнуто је пропорционална том броју. Када се посматра физичко времена потребно да се обради контекст једне компонента код постојећих симулатора мора се посматрати период унутар једне симулације. Овом анализом се дошло до расподеле времена у односу на номинални јединични догађај Pc обраде која је приказан на слици 100. На основу ових мерења се може закључити да је расподела вероватноће чувања контекста имала просечну вредност од 0.0072, стандардну девијацију 0.00 и коефицијент спљоштености -0.61.



Слика 100: Расподела вероватноће чувања контекста код симулатора архитектуре и организације рачунара

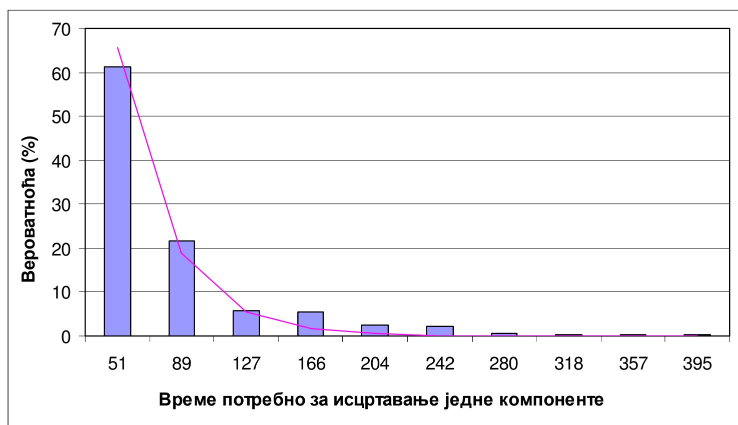
Када се посматра физичко времена потребно да се обради контекст једне компонента код постојећих симулатора мора се посматрати период унутар једне симулације. Овом анализом се дошло до расподеле времена у односу на номинални јединични догађај tb обраде која је приказан на слици 101. На основу ових мерења се може закључити да је расподела логичког времена имала просечну вредност од 0.31 у односу на физичко време обраде логичке компоненте, стандардну девијацију 0.24 и коефицијент спљоштености 113.38. Оваква расподела ова три параметра указује на то да је код анализираних система физичко време потребно за обраду контекста једне компоненте има расподелу која доста прецизно одговара Стандардног нормалној расподели. Експоненцијална расподела која описује ову карактеристике логичког времена има математичко очекивање 0.3 и приказана је на истој слици.



Слика 101: Расподела времена потребног за обраду контекста једне компоненте

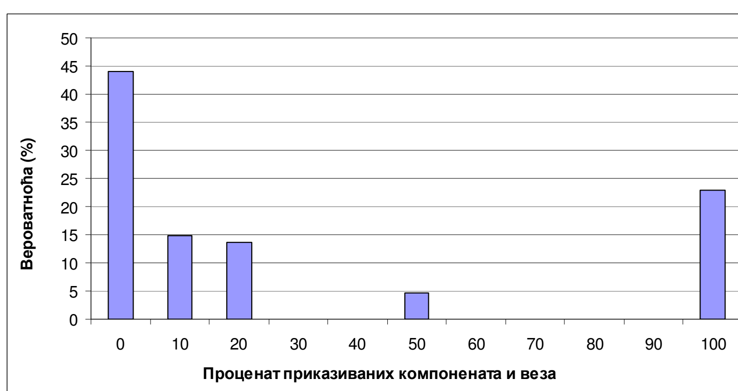
Када се посматра физичко времена потребно да се исцрта једна презентациона компонента код постојећих симулатора мора се посматрати период унутар једне симулације. Овом анализом се дошло до расподеле времена у односу на номинални јединични догађај td обраде која је приказан на слици 102. На основу ових мерења се може закључити да је расподела логичког времена имала просечну вредност од 61.53, стандардну девијацију 54.07 и коефицијент спљоштености 7.51. Оваква расподела ова три параметра указује на то да је код анализираних система физичко време потребно за исцртавање једне презентационе компоненте има расподелу која доста прецизно одговара Експоненцијалној расподели. Експоненцијална расподела која описује ову карактеристике логичког времена има математичко очекивање 60.72 и приказана

је на истој слици.



Слика 102: Расподела времена потребног за графички приказ компоненте која је обрадила догађај током симулација

Да би се обезбедило да се на адекватан начин опише понашање симулирани систем потребно је поред одређивања физичког времена потребног да се исцрта једна презентациона компонента одредити и колико је компонената потребно исцртати. До овог параметра се долази тако што је код симулатора из области Архитектуре и организације рачуната вршено пребројавање колико компонената постоји по једном презентационом екрану. Овом анализом се дошло до расподеле која је приказан на слици 103. На основу ових мерења се може закључити да је број екрана не којима се може приказати једна компонента 20 односно да је вероватноћа приказивања компоненте имала стандардну девијацију 1.93 и коефицијент спљоштености 22.57.



Слика 103: Расподела вероватноће приказа компоената и веза унутар симулатора у области Архитектуре и организације рачунара

5.2.2 Евалуација аналитичког модела на основу експерименталних резултата

Евалуација резултата аналитичког модела треба да покаже у којој мери модел одступа од измерених резултата и да да смернице око коришћења паралелицаије у приликом симулација у области архитектуре и организације рачунара које се примењују у настави. У овој секцији је представљено поређење измерених вредности током симулације користећи SLEEP са карактеристичним оптерећење добијеним развојем појединих симулатора и очекиваних резултата на основу предложеног аналитичког облик за очекивани добитак и анализу перформанси.

Табела 15: Параметри симулације, као и типичне вредности добијене током анализе

Параметар	Име	Типична вредност током тестирања	Подразумевана вредност
Број компоненти	M	200-6000	1900
Број логичких веза	V	400-25000	6000
Број веза по порту	F	1.1-10	1.66
Трајање симулације	L/t_l	1-∞	3
Време трајања догађаја	t_l	1-10	1
Време обраде догађаја	t_p	5-10000	100
Вероватноћа стварања догађаја	Pf	0-1	0.25
Проценат паралелно активних компоненти	Pe	0-1	0.05
Број процесора	N	2-32	4
Вероватноћа да порука остаје на истом рачунару	Pb	0.03-0.5	1/N
Време транспорта догађаја	t_k/t_p	0.01-0.8	0.1
Вероватноћа формирања контекста	Pc	0-1	0.0075
Време обраде контекста	t_b/t_p	0.05-1	0.33
Време приказа компоненте	t_d/t_p	0-4	1
Вероватноћа приказа компоненте	Pd	0-1	0.05
Време физике компоненте	t_f/t_p	0	0

За потребе евалуације аналитичког модела употребљена је симулација. Симулација је заснована на моделу описаном у секцији 3.2 „Аналитички модел алгоритама симулација“, са синтетичким оптерећењем заснованим на коришћењу

параметара описаним у секцији 5.2.1 „Експериментални резултати“. Улазни параметри у свим експериментима су постављени на вредности које би одговарале у типичним тест апликацијама. У Табели 15 су дати опсези вредности у којима се параметри крећу, као и вредности које су узете у експериментима када је тај параметар био фиксиран. Сваки од параметара је вариран у засебном експерименту. У наставку су приказани експерименти који описују карактеристике предложеног решења.

Насупрот аналитичком моделу, симулациони модел имплементира симулације које се извршавају на N рачунара као конкурентне процесе. Сваки од процеса одабира секвенцу све док логичко време симулације не достигне L и док се не обраде сви догађаји распоређени за тај временски интервал. Пошто је са аспекта овог рада битан тренутак када је симулација обрадила неки догађај узето је да је расподела догађаја унутар симулације је униформна на L. Секвенца приступа остаје непромењена за све време животног циклуса једне симулације. Ово значи да након рестартовања симулација покушава да изврши идентичну секвенцу операција. Нове симулације се започињу све док се не догоди довољно велики број рестарта за постизање интервала поверења од 99% и релативне грешке од 5%.

Поређење симулатора са оптимистичним алгоритмом симулације користећи аналитички модел.

У случају да је вероватноће рестарта симулација иста у симулаторима који имају инкрементално чување контекста и симулатора који се увек враћају на почетак симулације, очекивани релативни добитак ако се посматрају само симулације које су имале рестарт:

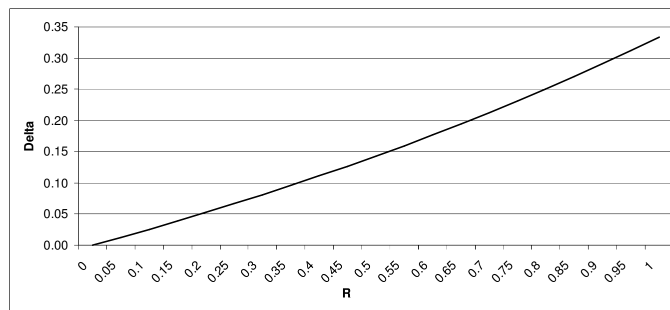
$$\delta = \frac{E(T_{R-NR}) - E(T_{R-INC})}{E(T_{R-NR})}$$

$$R_{INC} = R_{NR} = R$$

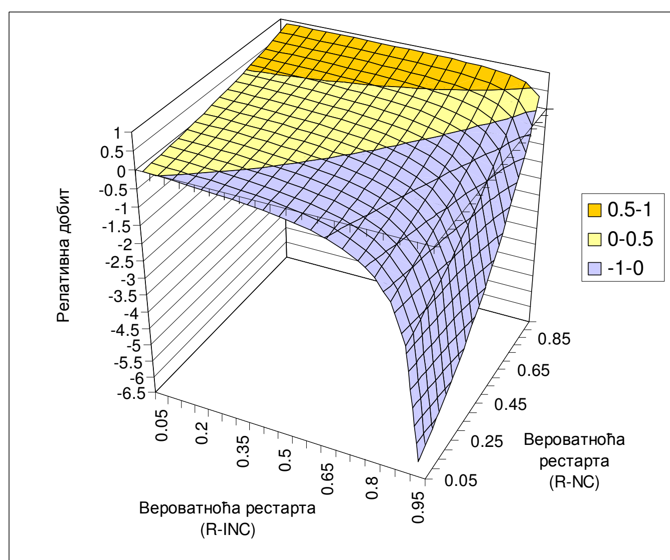
$$\delta = \frac{R}{(4-R)}$$

Одавде се може закључити да према предложеном моделу инкрементално чување контекста има боље карактеристике у односу на симулаторе који се увек враћају на почетак симулације. Добитак ових симулатора је у интервалу од 0 од

0,33. На слици 104 је представљен релативни добитак код симулатора који имају инкрементално чување контекста и симулатора који се увек враћају на почетак симулације.



Слика 104: Релативни добитак код симулатора који имају инкрементално чување контекста и симулатора који се увек враћају на почетак симулације



Слика 105: Релативни добитак код симулатора који имају инкрементално чување контекста и симулатора који се увек враћају на почетак симулације у ситуацији да се разликује вероватноћа рестарта симулације

У случају да се вероватноћа рестартра симулације разликује код симулатора који имају инкрементално чување контекста и симулатора код којих се симулација увек враћа на почетак онда може доћи до већих разлика између ових симулатора. Иако није предвиђена предложеним моделом ова разлика се може појавити у случајевима када симулација код симулатора који инкрементално чувају контекст може брже поново доћи до тренутка када приступа податку због

кога је настао конфликт. У случају када се битно промени вероватноћа рестарта систем са инкременталним чувањем контекста може због већег броја рестарта имати лошије перформансе. На основу аналитичког облика за очекивано време извршавања симулација график који приказује очекивани добитак у случају да се вероватноће рестарта разликују је дат на слици 105.

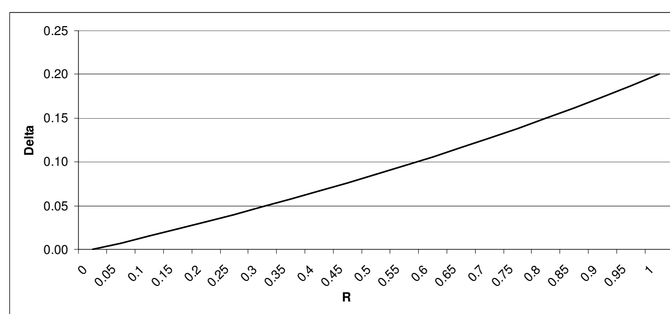
У случају да је вероватноће рестарта симулација иста у симулаторима који имају периодично чување контекста и симулатора који се увек враћају на почетак симулације, очекивани релативни добитак ако се посматрају само симулације које су имале рестарт:

$$\delta = \frac{E(T_{R-NR}) - E(T_{R-PR})}{E(T_{R-NR})}$$

$$R_{PR} = R_{NR} = R$$

$$\delta = \frac{(n-1)}{(n+1)} \cdot \frac{R}{(4-R)}$$

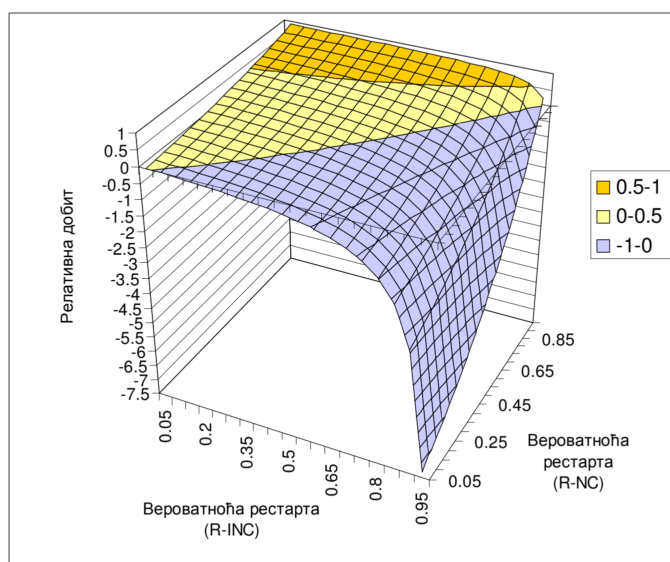
Одавде се може закључити да према предложеном моделу периодино чување контекста има боље карактеристике у односу на симулаторе који се увек враћају на почетак симулације. Добитак ових симулатора је у интервалу од 0 од 0,33. На слици 106 је представљен релативни добитак код симулатора који имају периодично чување контекста и симулатора који се увек враћају на почетак симулације у случају да n једнако 4.



Слика 106: Релативни добитак код симулатора који имају периодично чување контекста и симулатора који се увек враћају на почетак симулације

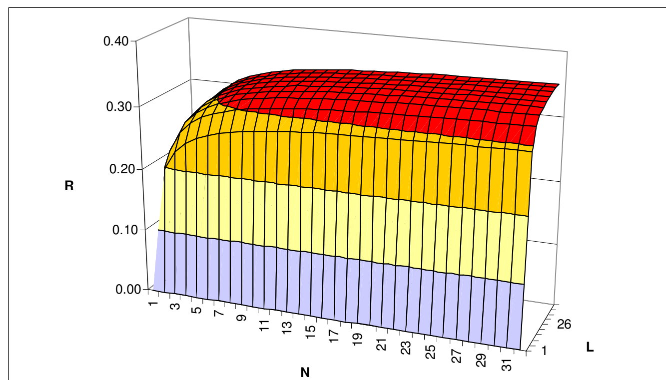
У случају да се вероватноћа рестартра симулације разликује код симулатора који имају периодично чување контекста и симулатора код којих се симулација увек враћа на почетак онда може доћи до већих разлика између ових симулатора. Иако није предвиђена предложеним моделом ова разлика се може појавити у случајевима када симулација код симулатора који периодично чувају

контекст може брже поново доћи до тренутка када приступа податку због кога је настао конфликт. У случају када се битно промени вероватноћа рестарта систем са периодичним чувањем контекста може због већег броја рестарта имати лошије перформансе. На основу аналитичког облика за очекивано време извршавања симулација график који приказује очекивани добитак у случају да се вероватноће рестарта разликују је дат на слици 107 у случају да n једнако 4.

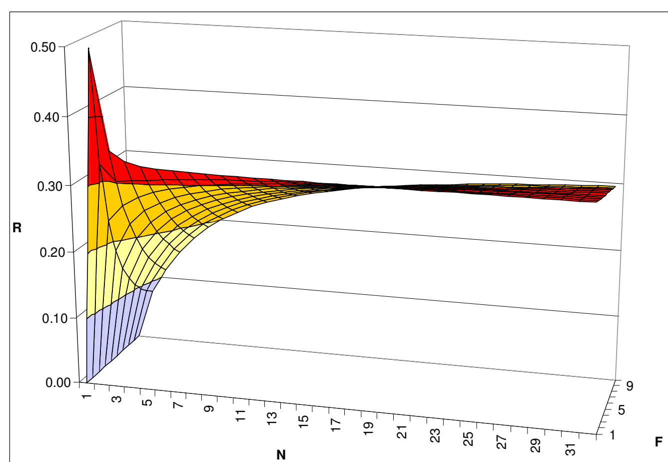


Слика 107: Релативни добитак код симулатора који имају инкрементално чување контекста и симулатора који се увек враћају на почетак симулације у ситуацији да се разликује вероватноћа рестарта симулације

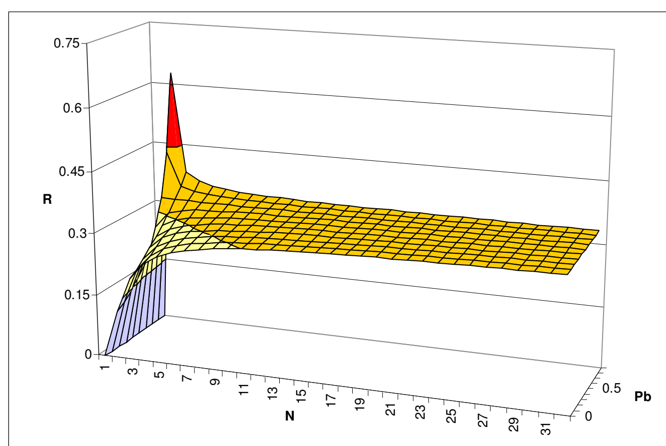
Према аналитичком моделу вероватноћа рестарта симулације зависи од већег броја параметара симулације. Ови параметри су број процесора на којима се симулација извршава, логичко времена трајање симулације, логичко време трајања једног догађаја, време потребно да се подаци о једном догађају транспортују са једног рачунара на други рачунар, вероватноће да се догађај са једног рачунара транспортује на други рачунар, и броја рачунара на које је потребно проследити догађај. У зависности од конкретне вредности постављених параметара ови параметри имају већи или мањи значај. Да би се испитао значај ових параметара биће испитан зависност између њих док ће остали параметри имати фиксне вредности према предложеној Табели 15. Резултати добијени на основу аналитичког модела су дати на сликама 108-110.



Слика 108: Вероватноћа рестарта симулације у зависности од броја радних станица и логичког времена симулације



Слика 109: Вероватноћа рестарта симулације у зависности од броја радних станица и броја компонената повезаних са компонентом која је емитовала догађај



Слика 110: Вероватноћа рестарта симулације у зависности од броја радних станица и вероватноће да порука остаје на истом рачунару

Поређење времена извршавања симулације

Време које је потребно да се симулација обави се може посматрати са два аспекта. Први је аспект време које је потребно да се обаве израчунавања следећег стања у које симулација прелази под неким условом, а други аспект представља време потребно да се израчуна следеће стање симулације али и да се прикажу резултати симулације. Ова два аспекта посматрања симулације могу доста да се разликују по карактеристикама јер је време које је потребно да се симулација израчуна у неким случајевима упоредиво са временом које је потребно да се резултати симулације у интерактивном моду рада прикажу.

Скраћенице коришћене приликом графичког приказа резултата:

TP – Временски вођена симулација (аналитички модел)

TPEx – Временски вођена симулација (експериментални резултати)

EPC – Конзервативна догађајима вођена симулација (аналитички модел)

EPCEx – Конзервативна догађајима вођена симулација (експериментални резултати)

EPOI – Оптимистичка догађајима вођена симулација са инкременталним чувањем контекста (аналитички модел)

EPOIEx – Оптимистичка догађајима вођена симулација са инкременталним чувањем контекста (експериментални резултати)

EPOP – Оптимистичка догађајима вођена симулација са периодичним чувањем контекста (аналитички модел)

EPOPEx – Оптимистичка догађајима вођена симулација са периодичним чувањем контекста (експериментални резултати)

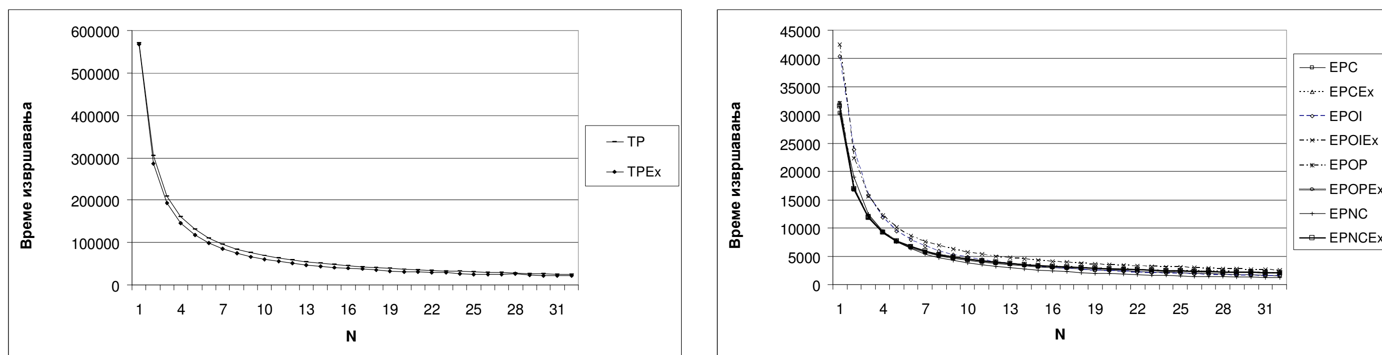
EPNC – Оптимистичка догађајима вођена симулација без чувањем контекста (аналитички модел)

EPNCEx – Оптимистичка догађајима вођена симулација без чувањем контекста (експериментални резултати)

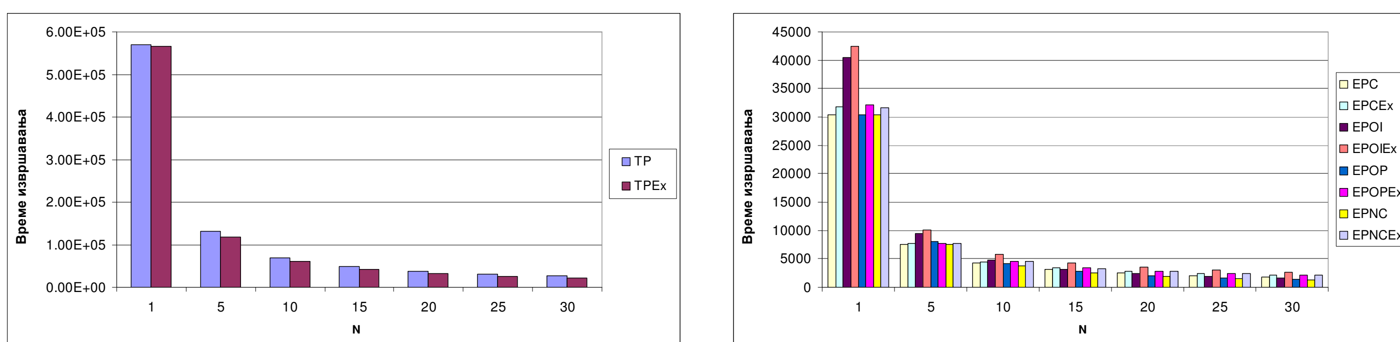
А – Зависност трајања симулације од броја рачунара

Резултати аналитичког модела и експерименталних мерења времена извршавања извршног дела симулације у системима без интеракције са корисником су приказани на сликама 111-112. У датим експериментима је

вариран број процесора на којима се симулација извршава док су остали параметри узети сходно Табели 15 са подразумеваним вредностима.



Слика 111: Резултати аналитичког модела и експерименталних мерења времена извршавања извршног дела симулације у системима без интеракције са корисником у функцији броја расположивих радних станица

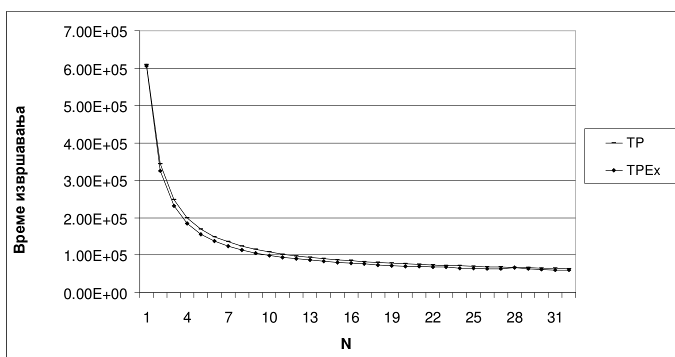


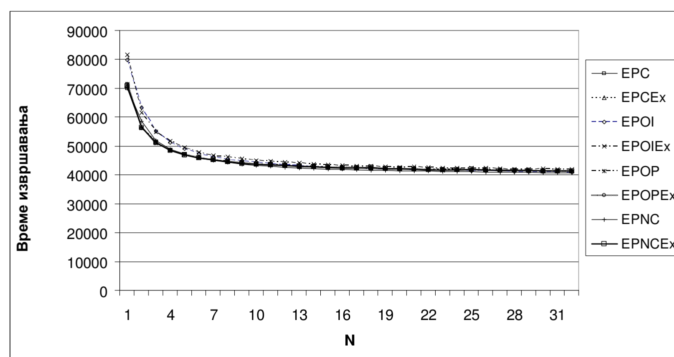
Слика 112: Резултати аналитичког модела и експерименталних мерења времена извршавања извршног дела симулације у системима без интеракције са корисником у функцији броја расположивих радних станица (детаљни приказ)

Према резултатима које дају и модели и експерименти може се приметити да је време симулације најкраће код симулатора који користе конзервативни приступ синхронизацији. У поређењу са другим симулаторима дискретних догађаја које користе оптимистични приступ симулацији карактеристике су боље у интервалу од 5 до 10 %. Овакав резултат је последица тога што постоји релативно мали број рестарта симулације на посматраним симулаторима. Пошто је број рестарта веома мали симулатори који користе оптимистички приступ симулацији троше доста времена на операцијама чувања контекста симулатора

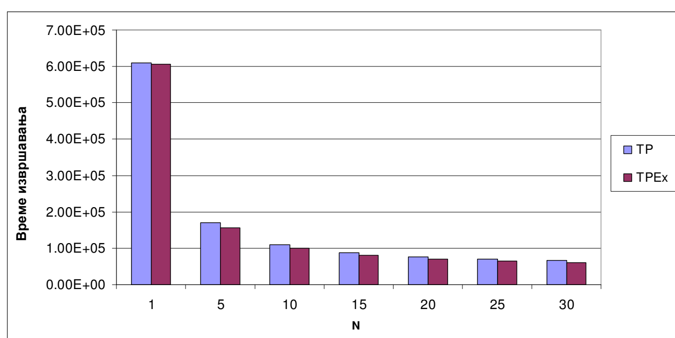
која онда остају не искоришћена. Симулатори који користе временом вођене симулације имају време извршавања које је за ред величина дуже од времена симулатора дискретних догађаја, тако да се без додатних оптимизација тешко могу поредити по перформансама са симулаторима дискретних догађаја. Оптимизације би се огледале у томе да би се обрада обављала само уколико би се установило да је било промена на улазним приступним тачкама. Уколико се гледа поређење аналитичког модела и експерименталних резултата одступање је у границама мањим од 5% за дати опсег улазних параметара.

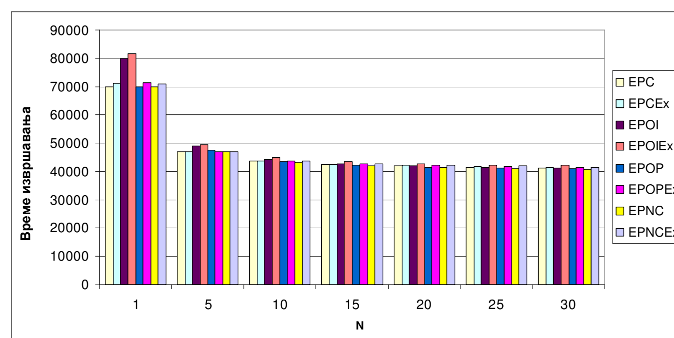
Резултати аналитичког модела и експерименталних мерења времена извршавања комплетне симулације у системима са интеракције са корисником у функцији броја расположивих радних станица су приказани на сликама 113-114. У датим експериментима је вариран број процесора на којима се симулација извршава док су остали параметри узети сходно Табели 15.





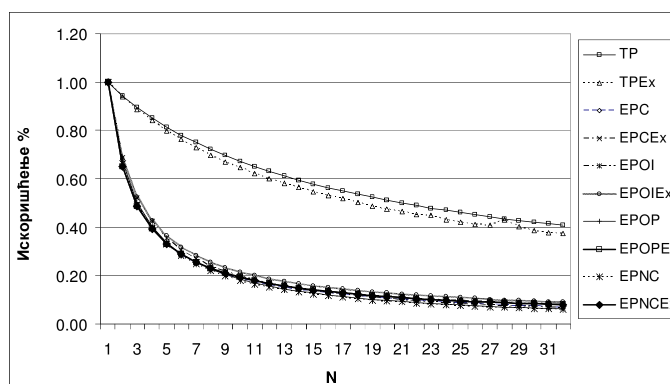
Слика 113: Резултати аналитичког модела и експерименталних мерења времена извршавања комплетне симулације у системима са интеракције са корисником у функцији броја расположивих радних станица





Слика 114: Резултати аналитичког модела и експерименталних мерења времена извршавања комплетне симулације у системима са интеракције са корисником у функцији броја расположивих радних станица (детаљни приказ)

Према предложеним моделима види се да је и овде време симулације најкраће код симулатора који користе конзервативни приступ синхронизацији, али је тај проценат убрзања доста мањи, 2 до 4 %. Овакав резултат је последица тога што је време које је потребно за исцртавање симулације упоредиво и приближно једнако укупном времену које је потребно за комплетно извршавање симулације, као и релативно малог број рестарта симулација.

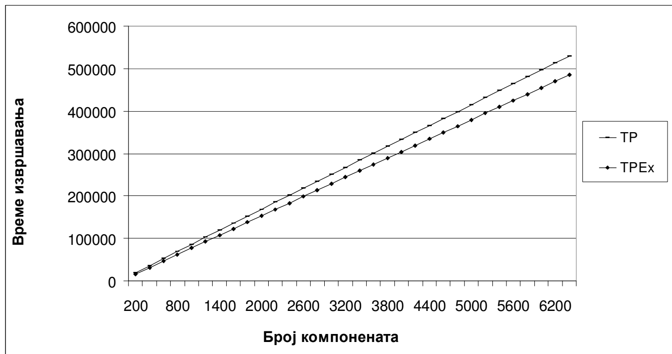


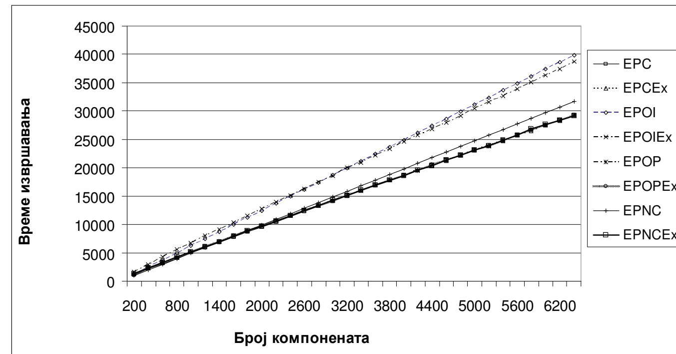
Слика 115: Резултати аналитичког модела и експерименталних мерења искоришћености радних станица приликом симулације у функцији броја расположивих радних станица

Резултати аналитичког модела и експерименталних мерења искоришћења процесора у функцији броја радних станица је приказан на слици 115. Може се приметити да искоришћење сваке од радних станица пада са повећањем броја радних станица. На пример у систему који има 4 радне станице искоришћење сваке од њих је испод 30% а укупан добитак на перформансама са повећањем броја радних станица је занемарљив. Повећање броја расположивих радни станица изнад одређене границе не доприноси значајно убрзању рада симулатора.

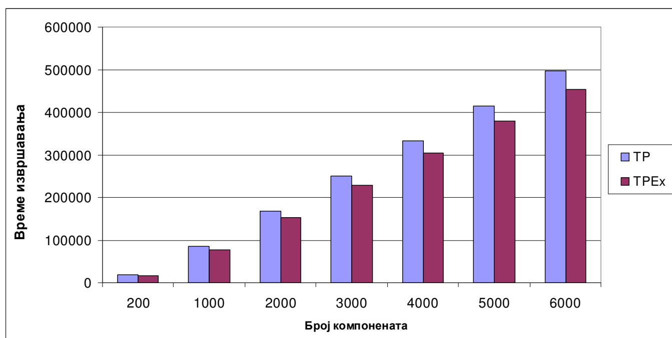
Б – Зависност време трајања симулације од броја симулираној компоненти

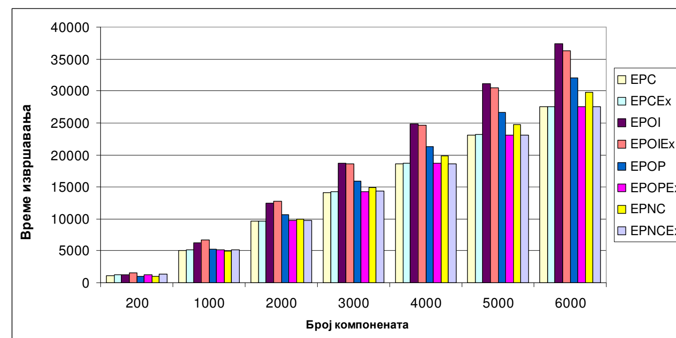
Резултати аналитичког модела и експерименталних мерења времена извршавања извршног дела симулације у системима без интеракције са корисником су приказани на сликама 116-117. У датим експериментима је вариран број симулираних компоненти при фиксном односу са бројем веза док су остали параметри узети сходно Табели 15 са подразумеваним вредностима.





Слика 116: Резултати аналитичког модела и експерименталних мерења времена извршавања извршног дела симулације у системима без интеракције са корисником у функцији броја симулираних компоненти



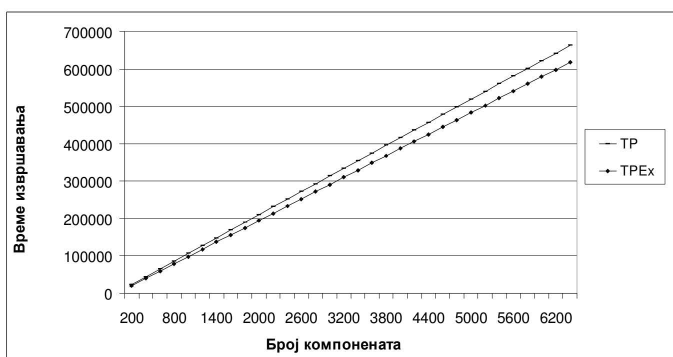


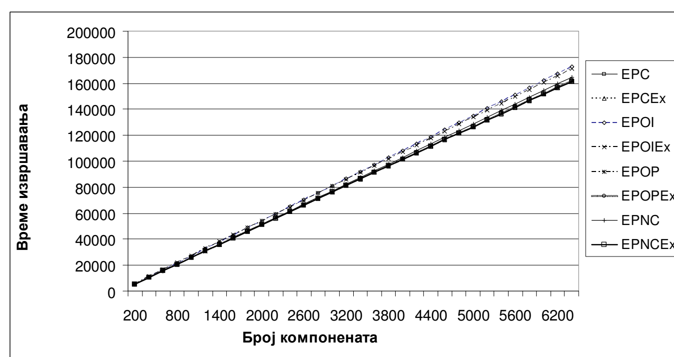
Слика 117: Резултати аналитичког модела и експерименталних мерења времена извршавања извршног дела симулације у системима без интеракције са корисником у функцији броја симулираних компоненти (детаљни приказ)

Према резултатима које дају и модели и експерименти може се приметити да је време симулације и у овом случају најкраће код симулатора који користе конзервативни приступ синхронизацији. У поређењу са другим симулаторима дискретних догађаја које користе оптимистични приступ симулацији карактеристике су боље у интервалу од 5 до 25 %. Овакав резултат је последица тога што постоји веома мали број рестарта симулације на посматраним симулаторима јер не постоји повратна спрега између симулираних компонената. Пошто је број рестарта веома мали симулатори који користе оптимистички приступ симулацији троше доста времена на операцијама чувања контекста симулатора која онда остају не искоришћена. Симулатори који користе временом вођене симулације имају време извршавања које је за ред величина дуже од

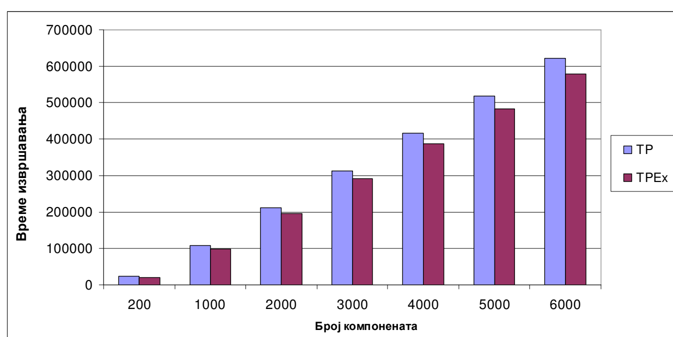
времена симулатора дискретних догађаја.

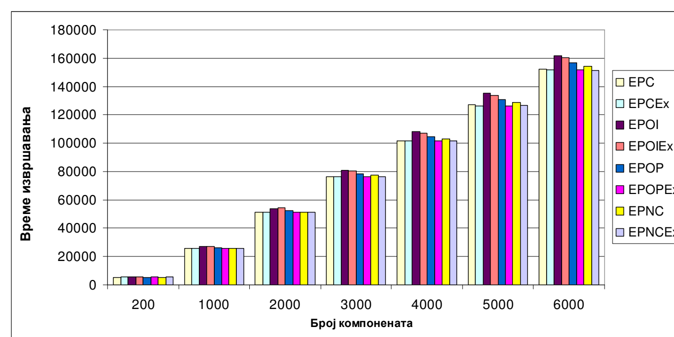
Резултати аналитичког модела и експерименталних мерења времена извршавања комплетне симулације у системима са интеракције са корисником у функцији броја симулираних компоненти при фиксном односу броја компоненти и броја веза су приказани на сликама 118-119. У датим експериментима је вариран број симулираних компоненти док су остали параметри узети сходно Табели 15.





Слика 118: Резултати аналитичког модела и експерименталних мерења времена извршавања комплетне симулације у системима са интеракције са корисником у функцији броја симулираних компонената

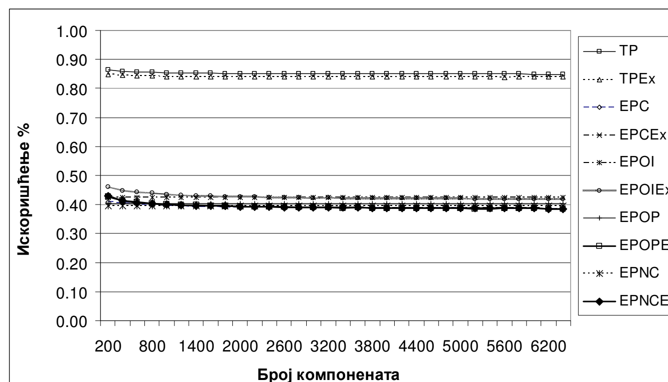




Слика 119: Резултати аналитичког модела и експерименталних мерења времена извршавања комплетне симулације у системима са интеракције са корисником у функцији броја симулираних компонената (детаљни приказ)

Према предложеним моделима види се да је и овде време симулације најкраће код симулатора који користе конзервативни приступ синхронизацији, али је тај проценат убрзања доста мањи, 2 до 12 %. Овакав резултат је последица тога што је време које је потребно за исцртавање симулације упоредиво и приближно једнако укупном времену које је потребно за комплетно извршавање

симулације, као и релативно малог број рестарта симулација.



Слика 120: Резултати аналитичког модела и експерименталних мерења искоришћености радних станица приликом симулације у функцији броја симулираних компонената

Резултати аналитичког модела и експерименталних мерења искоришћења процесора у функцији броја радних станица је приказан на слици 120. Може се приметити да искоришћење сваке од радних станица пада са повећањем броја радних станица. У посматраним експериментима посматране су симулације на 4 радне станице и може се приметити да је код симулатора дискретних догађаја искоришћење радних станица око 40% док је код временски вођене симулације искоришћење око 85%. Повећање броја симулираних компоненти не утиче на искоришћење јер оно веома брзо улази у засићење.

6 ЗАКЉУЧАК

У овом раду је разматран методолошки приступ дизајну симулатора из области архитектуре и организације рачунара који треба да омогући развој симулатора дигиталних система произвољног нивоа сложености способних за рад у конкурентном и дистрибуираном окружењу. Да би се омогућио формирање методологије на почетку рада је приказан преглед наставе у области архитектуре и организације рачунара на основним студијама, као и преглед области пројектовања симулатора где је посебан акценат био стављен на области конкурентног и дистрибуираног програмирања које студенти треба да познају као би могли да развију симулаторе који омогућавају рад у таквом окружењу. На основу спроведене евалуације симулатора који се користе у настави из области архитектуре и организације рачунара а који имају расположив изворни код предложено је решење које се заснива на коришћењу слојевите архитектуре код које је сваки слој одговоран за други вид обраде и комуникације. Предложено решење се састоји из коришћења пет слојева: логичког, извршног, презентационог, симулационог, и слоја физике, а детаљи везани за процедуре и објашњења техника које се користе за реализацију ових слојева су приказани у наставку рада. За сваки слој предложеног решења је дат аналитички модел процене времена извршавања симулације у датом слој у зависности од улазних параметара приликом рада у конкурентном и дистрибуираном окружењу. Централни део рада описује симулатор дискретних догађаја опште намене развијен према описаној методологији као симулатор архитектуре и организације рачунара који је способан за рад у конкурентном и дистрибуираном окружењу. Опис симулатора и његових делова је дат са становишта детаља имплементације где су представљени пакети реализовани на основу предложене методологије, као и са становишта коришћења где су описане карактеристичне ситуације у којима се симулатор може користити. На основу имплементације симулатора и пратећих библиотека развијене су лабораторијске вежбе и пројекти из предмета конкурентно и дистрибуирано програмирање, које су представљене у наставку рада као и евалуација постигнутих резултата у настави. Поред ове евалуације на

крају рада је представљена и евалуација симулатора са становишта експерименталних резултата и са становишта аналитичког модела као би се утврдило у којим случајевима и у ком обиму се могу користити симулатори развијени сходно описаној методологији.

Развијена је методологија која се заснива на коришћењу слојевите архитектуре код које је сваки слој одговоран за различите облике комуникације и обраде. Одлука да се користи слојевита архитектура приликом пројектовања симулатора архитектуре и организације рачунара способних за извршавање у конкурентном и дистрибуираном окружењу омогућила је да се на једноставан начин раздвоје поступци обраде од поступака комуникације, синхронизације и интеракције између слојева без улажења у начине на које се подаци користе; На основу предложене методологије је без потешкоћа развијен SLEEP, симулатор архитектуре и организације рачунара са слојевитом архитектуром који се може користити и као симулатор дискретних догађаја опште намене. Показало се да предложени приступ развоју SLEEP симулатора доводи до стварања могућности за извршавање симулације користећи рад са једном нити, рад са више нити и рад у дистрибуираном окружењу; На основу евалуације резултата добијених током коришћења симулатора као један од закључака се издваја чињеница да се поступак пројектовања симулатора архитектуре и организације може користити у настави из конкурентног и дистрибуираног програмирања као полазна основа за поступак паралелизације апликација са једном током контроле. Пред тога симулатор се може користити и приликом тестирања тако паралелизованих апликација јер на конзистентан начин ствара велико оптерећење које у великој мери покрива понашање програма са већим бројем токова контроле; На основу евалуације резултата добијених на основу аналитичког модела и мерења перформанси може се закључити да се време извршавања симулације у области наставе архитектуре и организације рачунара може значајно смањити у конкурентном и дистрибуираном окружењу у случају ниског степена интеракције са корисником. У случају просечног нивоа интеракције ограничавајући фактор приликом извршавања симулација представља време потребно за приказ резултата симулације које је упоредиво са временом потребним за обраду саме симулације.

Предложено решење се заснива на коришћењу слојевите архитектуре, али би поред праволинијске имплементације и презентације у појединим корацима било боље користити додатно прилагођен приступ. Показало се да би приликом коришћења у настави из конкурентног и дистрибуираног програмирања било боље сакрите поједине слојеве јер студенте у већини случаја занима не само развој симулационог слоја симулатора већ и осталих слојева како би стекли ширу слику о систему. Откривање слојева и њиховог изворног кода може непотребно да продужи развој дистрибуираних апликација, па је да би се скратило време израде пројеката SLEEP симулатор студентима представљан само у облику библиотека које су могли да користе приликом развоја и тестирања апликација. Пројектна одлука да се за чување и транспорт података о компонентама у токовима података и датотекама користе уграђене Јава библиотеке, а не наменски развијен формат записа шеме симулираног система се показала веома добром приликом континуалног рада са симулатором. Приликом преласка са једне верзије симулатора на другу у случају да је било промена у називима пакета и основних класа симулатора се показало да би било боље да је коришћен сопствени формат записа јер је овако потпуно изгубљено све што је у претходним верзијама развијено од корисничких компонената. Предложени аналитички модел даје задовољавајуће резултате у случајевима адекватне расподеле компонената по дистрибуираним рачунарима у складу са представљеним ограничењима. Уколико постоје значајне разлике у оптерећењу које доводе до постојања празног хода приликом кога не постоје компоненте које се могу обрађивати, модел не даје прецизне резултате.

Даљи развој SLEEP симулатора може да иде у више праваца. Основни правац даљег развоја би били формирање већег броја наменских библиотека које могу да описују комерцијално доступне компоненте које се срећу у области архитектуре и организације рачунара. Ове библиотеке би поред описа понашања требале да поседују и податке о хардверу на коме би се симулација обавила како би се на адекватан начин урадила временска анализа и искористили потенцијали догађајима вођене симулације. Следећи правац унапређење симулатора би се односио на повећање обима кода које уграђени преводиоци могу да остваре. Ова унапређења би се односила на како на преводилац VHDL језика у интерну Јава

репрезентацију где би требало поред описа структуре симулираног система омогућити и опис понашања симулираног система. Постоји могућност развоја имплементације датог интерфејса за превођење које би могле да покрију и друге језике који се користе у поступку симулације као што је Matlab. Садашња имплементација корисничког интерфејса SLEEP симулатора је реализована користећи MVC пројектну узорак и имплементирана користећи и Swing и SWT технологије. Користећи то да постоји имплементација у SWT технологији ова имплементација се може прилагодити тако да постане додатак (plugin) за Eclipse окружење. Распоређивање компонената по рачунарима је сада препуштено кориснику да самостално одреди где која компонента треба да се смести, што може да знатно умањи перформансе приликом дистрибуираног извршавања. Као један од модула би требали имплементирати алгоритам за аутоматско распоређивање компонената на основу пробног извршавања. Током пробног извршавања би се прецизни одредила априорна статистика понашања симулираних компонената на основу чега би се компоненте смештале у дистрибуираном окружењу. Један од праваца унапређења симулатора би могао да се заснива на анализи корисничке радне површи. Приликом анализе начина на који кориснички интерфејс утиче на корисника, студента, уочено је да корисници нижих година студија поистовећују окружење и предмет тако да је за њих боље да је окружење интуитивно и да не садржи превише напредних опција. У том добу напредне опције могу да сметају и да студенти због тих опција не могу да схвате суштину јер се фокусирају на детаље. Напредни корисници на вишим годинама студија или на последипломским студијама када се сусретну са окружењем не инсистирају толико на изглед компоненти већ на могућностима који окружење пружа и брзини којом се симулација обавља.

Литература

[1] B. Nikolic, Z. Radivojevic, J. Djordjevic, and V. Milutinovic, "A Survey and Evaluation of Simulators Suitable for Teaching Courses in Computer Architecture and Organization," IEEE Transactions on Education, vol. 52, no. 4, pp. 449-458, November 2009.

[2] IEEE Computer Society/ACM Computing Curriculum - Computer Engineering, "Computing Curricula - Computer Engineering (CCCE) Task Force," Available from: http://www.eng.auburn.edu/ece/CCCE/ (accessed August 2005).

[3] ACM/IEEE Computer Society, "Computer Science Curriculum 2008: An Interim Revision of CS 2001," December 2008, Available: http://www.computer.org/portal/cms_docs_ieeecs/ieeecs/education/cc2001/ComputerScience2008.pdf.

[4] J. Đorđević, "Arhitektura i organizacija računara – učenje pomoću računara," Akademska misao, Beograd 2004.

[5] J. Đorđević, "Prirucnik iz arhitekture i organizacije racunara," Elektrotehnicki fakultet univerziteta u Beogradu, Beograd 1998.

[6] D. Ellard, D. Holland, N. Murphy, and M. Seltzer, "On the Design of a New CPU Architecture for Pedagogical Purposes," Proceedings of the Workshop on Computer Architecture Education, Anchorage, AK, USA, 2002, pp. 28-34.

[7] Harvard University, "Welcome to the Ant Home Page," Available from: http://www.ant.harvard.edu/ (accessed August 2005).

[8] D. Skrien, "CPU Sim 3.1: A tool for simulating computer architectures for computer organization classes," ACM Journal of Educational Resources in Computing (JERIC), vol. 1, pp. 46-59, December 2001.

[9] D. Skrien, "CPU Sim Home Page," Available from: http://www.cs.colby.edu/djskrien/CPUSim/ (accessed April 2008).

[10] A. Cohen, and O. Temam, "Digital LC-2: From bits & gates to a Little Computer," Workshop On Computer Architecture Education, Anchorage, AK, USA, 2002, Article No. 11.

[11] A. Cohen, and O. Temam, “DigLC2 - Gate-level LC-2 Simulator,” Available from: http://www-rocq.inria.fr/~acohen/teach/diglc2.html.en (accessed June 2005).

[12] Y. Zhang, and G. B. Adams III, “An Interactive, Visual Simulator for the DLX Pipeline,” Workshop On Computer Architecture Education (WCAE-03), San Antonio, Texas, USA, 1997, Article No. 2.

[13] G. Adamas, “DLXview v0.9 - Home Page,” Available from: http://yara.ecn.purdue.edu/~teamaaa/dlxview/ (accessed August 2005).

[14] J. Djordjevic, B. Nikolic, and A. Milenkovic, “Flexible Web-based Educational System for Teaching Computer Architecture and Organization,” IEEE Transactions on Education, vol. 48, no. 2, pp. 264-273, May 2005.

[15] J. Djordjevic, B. Nikolic, and A. Milenkovic, “Computer Architecture Laboratory,” Available from: http://rti.etf.bg.ac.rs/rti/ef2ar/labvezbe/CISCA.zip (accessed June 2011).

[16] R. N. Ibbett, “HASE DLX Simulation Model,” IEEE Computer Society, IEEE Micro, Special Issue on Computer Architecture Education, vol. 20, no. 3, pp. 38-47, May/June 2000.

[17] Institute for Computing Systems Architecture, School of Informatics, University of Edinburgh, “HASE - a computer architecture simulation environment,” Available from: http://www.icsa.inf.ed.ac.uk/research/groups/hase/ (accessed August 2005).

[18] R. Ibbett, and F. Mallet, “Computer Architecture Simulation Applets For Use In Teaching,” Frontiers in Education, 2003. FIE 2003. 33rd Annual, Boulder, USA, 2003, vol. 2, pp. F1C-20-5.

[19] Institute for Computing Systems Architecture, School of Informatics, University of Edinburgh, “COMPUTER ARCHITECTURE: HASE Dinero,” Available from: http://www.icsa.informatics.ed.ac.uk/research/groups/hase/projects/dinero/ (accessed August 2005).

[20] I. Branovic, R. Giorgi, C.A. Prete, "Web-based training on Computer Architecture: The case for JCachesim," IEEE Workshop on Computer Architecture Education (WCAE-02), Anchorage, AK, USA, 2002, pp. 56-60.

[21] R. Giorgi, “JCacheSim-University of Siena, Faculty of Computer Engineering,” Available from: http://www.dii.unisi.it/~giorgi/jcachesim/ (accessed August 2005).

[22] B. Hutchings, P. Bellows, J. Hawkins, S. Hemmert, B. Nelson, and M. Rytting, “A CAD Suite for High-Performance FPGA Design,” Field-Programmable Custom Computing Machines, 1999. FCCM '99. Proceedings. Seventh Annual IEEE Symposium on, Napa Valley, CA, USA, 1999, pp. 12-24.

[23] Brigham Young University, “JHDL Overview,” Available from: http://www.jhdl.org/overview.html (accessed August 2005).

[24] P. Bellows, B. Hutchings, “JHDL - An HDL for Reconfigurable Systems,” FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on, Napa Valley, CA, USA, 1998, pp. 175-184.

[25] C. Burch, “Logisim: A graphical system for logic circuit design and simulation,” Journal of Educational Resources in Computing, vol. 2, no. 1, pp. 5-16, 2002.

[26] C. Burch, “Logisim,” Available from: www.cburch.com/logisim/ (accessed May 2007).

[27] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, “The M5 Simulator: Modeling Networked Systems,” IEEE Micro, vol. 26, no. 4, pp. 52-60, July/August 2006.

[28] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, “Main Page - M5,” Available from: http://www.m5sim.org (accessed September 2007).

[29] E. Pastor, and F. Sánchez, “La Máquina Rudimentaria: un Procesador Pedagógico,” III Jornadas de Enseñanza Universitaria sobre Informática (JENUI'97), Madrid, Spain, 1997, pp. 395-402.

[30] E. Pastor, F. Sanchez, and A. M. del Corral, “A Rudimentary Machine: Experiences in the Design of a Pedagogic Computer,” The Workshop on Computer Architecture Education (WCAE-98), Barcelona, Spain, 1998, Article No. 7.

[31] Universitat Politècnica de Catalunya, “Página Oficial de la Máquina Rudimentaria,” Available from: http://docencia.ac.upc.edu/eines/MR/ (accessed June 2008).

[32] C. Hughes, V. Pai, P. Ranganathan, and S. Adve, “RSIM: Simulating Shared-Memory Multiprocessors with ILP Processors,” IEEE Computer, vol. 35, no. 2, pp. 40-49, February 2002.

[33] V. S. Pai, P. Ranganathan, and S. V. Adve, “RSIM: An Execution-Driven Simulator for ILP Based Shared Memory Multiprocessors and Uniprocessors,” IEEE TCCA Newsletter, pp. 37-48, October 1997.

[34] S. Adve, “RSIM Home Page,” Available from: http://rsim.cs.uiuc.edu/rsim/ (accessed May 2008).

[35] J. Huang, “The Simulator for Multithreaded Computer Architecture Release 1.2,” Laboratory for Advanced Research in Computing Technology and Compilers, University of Minnesota, Minneapolis, MN, USA, Technical Report No: ARCTiC-00-05, 2000.

[36] J. Huang, and D. J. Lilja, “An Efficient Strategy for Developing a Simulator for a Novel Concurrent Multithreaded Processor Architecture,” Six Int'l Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, Montreal, Canada, 1998, pp. 185-191.

[37] J. Huang, “SIMCA Home Page,” Available from: http://agassiz.cs.umn.edu/Tools/SIMCA/simca.html (accessed May 2008).

[38] N. Hardavellas, S. Somogyi, T. F. Wenisch, R. E. Wunderlich, S. Chen, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk, “SIMFLEX: A Fast, Accurate, Flexible Full-System Simulation Framework for Performance Evaluation of Server Architecture,” ACM SIGMETRICS Performance Evaluation Review, vol. 31, no. 4, pp. 31-35, March 2004.

[39] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe, “SimFlex: Statistical Sampling of Computer Architecture Simulation,” IEEE Micro, vol. 26, no. 4, pp. 18-31, July/August 2006.

[40] Electrical and Computer Engineering at Carnegie Mellon University, “SimFlex: Fast & Accurate Scalable Simulation,” Available from: http://www.ece.cmu.edu/~simflex/ (accessed July 2007).

[41] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, “Simics: A Full System Simulation Platform,” Computer, vol. 35, pp. 50-58, February 2002.

[42] Virtutech, “Virtutech Simics - Embedded Systems Simulation Platform, Virtual Hardware for Embedded Software Development,” Available from: http://www.virtutech.com/ (accessed August 2005).

[43] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod, "Using the SimOS Machine Simulator to Study Complex Computer Systems," ACM Transactions on Modeling and Computer Simulation, vol. 7, no. 1, pp. 78-103, January 1997.

[44] R. Bosch, "The SimOS Home Page," Available from: http://simos.stanford.edu/ (accessed August 2005).

[45] T. M. Austin, "The SimpleScalar Tool set as an Instructional Tool: Experiences and Future Directions," 4th Annual Workshop on Computer Architecture Education (WCAE4), Las Vegas, Nevada, USA, 1998, Article No. 1.

[46] D. Burger, and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0," ACM SIGARCH Computer Architecture News, vol. 25, pp. 13-25, June 1997.

[47] The SimpleScalar Toolset, "SimpleScalar LLC," Available from: http://www.simplescalar.com (accessed August 2005).

[48] N. Grbanovic, B. Nikolic, and J. Djordjevic, "The VSDS environment based laboratory in computer architecture and organization," Computer Applications in Engineering Education, vol. 19, pp. n/a, doi: 10.1002/cae.20353, May 2009.

[49] N. Grbanovic, "Interaktivni generator vizuelnih simulatora digitalnih sistema," Doktorska disertacija, Elektrotehnički fakultet Univerzitet u Beogradu, Decembar 2009.

[50] N. Grbanovic, J. Djordjevic, and B. Nikolic, "Logic Design Laboratory," Available from: http://rti.etf.bg.ac.rs/rti/oo1pot/simulator/IGoVSoDS_SVE_v1.1.228_17.02.2008.zip (accessed February 2008).

[51] V. Krishnaswamy, J. Casas, T. Tetzlaff, "A Switch Level Fault Simulation Environment", Annual ACM IEEE Design Automation Conference, pp. 780-785, 2000.

[52] M. Migliardi, V. Sunderam, "Distributed, reconfigurable simulation in Harness," In Proceedings of Parallel and Distributed Processing Techniques and Applications Conference, pp. 730-736, Las Vegas, June 1999.

[53] A. Diaz-Calderon, C. Paredis, P. Khosla, "Organization and selection of reconfigurable models," Winter Simulation Conference, Orlando, USA, December 2000, vol. 1, pp. 386-393.

[54] G. Meister, “A survey on parallel logic simulation,” Technical report, Department of Computer Science, University of Saarland, Germany, pp. 1-45, September 1993.

[55] B. Nikolic, Z. Radivojevic, J. Djordjevic, and V. Milutinovic, "Survey of simulators," Available from: http://rti.etf.bg.ac.rs/rti/ri3aor/Simulators/index.html (accessed April 2008).

[56] B. Nikolic, “Sistem za učenje arhitekture i organizacije računara na daljinu,” Doktorska disertacija, Elektrotehnički fakultet Univerzitet u Beogradu, 2005.

[57] K. E. Holbert and G. G. Karady, "Strategies, Challenges and Prospects for Active Learning in the Computer-Based Classroom," IEEE Transactions on Education, vol. 52, no. 1, pp. 31-38, February 2009.

[58] ShanghaiRanking Consultancy, “Academic Ranking of Worldwide Universities 2010,” Shanghai Ranking Consultancy, Available: http://www.arwu.org/ARWU2010.jsp.

[59] U. Lero, “Poredjenje i analiza top 100 univerziteta sa sangajske liste,” projekat iz predmeta Konkurentno i distribuirano programiranje, Elektrotehnički falultet, 2010.

[60] A. Eckerdal, R. McCartney, J. E. Moström, M. Ratcliffe, K. Sanders, and C. Zander, “Putting threshold concepts into context in computer science education,” Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education, Bologna, Italy, 2006, pp. 103-107.

[61] J. Mache and A. Apon, “Teaching Grid Computing: Topics, Exercises, and Experiences,” IEEE Transactions on Education, vol. 50, no. 1, pp. 3-9, February 2007.

[62] H.-ur Rehman, R.’a A. Said, and Y. Al-assaf, "An Integrated Approach for Strategic Development of Engineering Curricula: Focus on Students' Design Skills," IEEE Transactions on Education, vol. 52, no. 4, pp. 470-481, November 2009.

[63] J. Magee, J. Kramer and D. Giannakopoulou, "Analysing the Behaviour of Distributed Software Architectures: a Case Study," Fifth IEEE Workshop on Future Trends of Distributed Computing Systems, Tunisia, October 1997, pp. 240-247.

[64] S. Uchitel, J. Kramer, and J. Magee, “Incremental elaboration of scenario-based specifications and behavior models using implied scenarios,“ ACM Transactions on Software Engineering and Methodology, vol. 13, no. 1, pp. 37-85, January 2004.

[65] P. J. Brooke and R. F. Paige, "Lazy Exploration and Checking of CSP Models with CSPsim," The 30th Communicating Process Architectures Conference, Guildford, UK, 2007, pp. 33-49.

[66] G. Malnati, C. M. Cuva, and C. Barberis, "JThreadSpy: A Tool for Improving the Effectiveness of Concurrent System Teaching and Learning," Proceedings of the 2008 International Conference on Computer Science and Software Engineering, Wuhan, China, 2008, vol. 5, pp. 549-552.

[67] J. B. G. Perez-Schofield, F. O. Soler, E. G. Roselló, and M. P. Cota, "Towards an object-oriented programming system for education," Computer Applications in Engineering Education, vol. 14, no. 4, pp. 243-255, 2006.

[68] C. Burger and K. Rothermel, "A Framework to Support Teaching in Distributed Systems," ACM Journal on Educational Resources in Computing, vol. 1, no. 1, pp. 1-13, August 2001.

[69] G. Licea, J. R. Juárez, L. G. Martínez, and L. Aguilar, "Developing programming tools to reach a deeper understanding of advanced programming concepts," Computer Applications in Engineering Education, vol. 16, no. 4, pp. 305-314, 2008.

[70] C. Kerer, G. Reif, T. Gschwind, E. Kirda, R. Kurmanowytsch, and M. Paralic, "ShareMe: Running a Distributed Systems Lab for 600 Students With Three Faculty Members," IEEE Transactions On Education, vol. 48, no. 3, pp. 430-437, August 2005.

[71] Z. Radivojević, M, Cvetanović, and Z. Jovanović, "Reengineering the SLEEP simulator in a concurrent and distributed programming course," Computer Applications in Engineering Education, vol. 19, no. n/a, doi: 10.1002/cae.20527, February 2011, ISSN: 1099-0542, IF 0.321.

[72] Z. Radivojević, M. Cvetanović, J. Đorđević, "Design of the simulator for teaching computer architecture and organization," Proceedings - 2011 2nd Eastern European Regional Conference on the Engineering of Computer Based Systems, ECBS-EERC 2011, pp. 124-130, Bratislava, Slovakia, September 5–6, 2011, DOI: 10.1109/ECBS-EERC.2011.26.

[73] Z. Radivojević, M. Cvetanović "Teaching the simulator design in Java," 11th Workshop Software Engineering Education and Reverse Engineering, Ohrid, Macedonia, 22-27 August 2011.

[74] Z. Radivojević, M. Cvetanović „Integracija JPC simulatora u konfigurabilni simulator keš memorije,“ ETRAN LIV – Donji Milanovac, Srbija, 7-11 juna 2010.

[75] Z. Radivojević, M. Cvetanović "Introduction to Grid Computing to students attending Concurrent and Distributed Programming courses," 8th Workshop Software Engineering Education and Reverse Engineering, Durres, Albania, 8-13 September 2008.

[76] M. Cvetanović, Z. Radivojević "Vizuelni simulator baziran na grid tehnologijama u nastavi iz konkurentnog i distribuiranog programiranja," ETRAN LII - Palić, Srbija, 8-12. jun 2008.

[77] J. Vujnić, Z. Radivojević, M. Cvetanović, "OLIGARCH Dizajn simulatora arhitekture računara," TELFOR XIV – Beograd, Srbija i Crna Gora, 21-23 Novembar 2006.

[78] Z. Radivojević, M. Cvetanović "Dizajn simulatora diskretnih događaja opšte namene," ETRAN L – Beograd, Srbija, 6-8. jun 2006.

[79] V. Cortellessa, G. Iazeolla, "Performance analysis of optimistic parallel simulations with limited rolled back events," Simulations Practice and Theory, Journal of Simulation Practice and Theory, vol.7, no.4, pp. 325-347, July 1999.

[80] S. C. Tay, Y. M. Teo, R. Ayani, "Performance analysis of Time Warp simulation with cascading rollbacks," Proceedings. Twelfth Workshop on Parallel and Distributed Simulation, 1998. PADS 98, pp. 30-37, Banff, Canada.

[81] A. Gupta, I. F. Akyildiz, R. M. Fujimoto, "Performance Analysis of Time Warp with Multiple Homogeneous Processors," IEEE Transactions on Software Engineering, vol.17, no.10, pp. 1013-1027, October 1991.

[82] G. Rodríguez, X. C. Pardo, M. J. Martín, P. González, "Performance evaluation of an application-level checkpointing solution on grids," Future Generation Computer Systems, vol.26, no.7, pp. 1012-1023, July, 2010.

[83] B. D. McLeod, "Performance evaluation of n-processor time warp using stochastic activity networks," Master's Thesis, Dept. of Electrical and Computer Engineering, University of Arizona, May 1993.

[84] Q. Liu, G. Wainer, "A Performance Evaluation of the Lightweight Time Warp Protocol in Optimistic Parallel Simulation of DEVS-based Environmental Models," Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation, Washington, USA, pp. 27-34, 2009.

[85] B. Lubachevsky, A. Weiss, A. Shwartz, "An Analysis of Rollback-Based Simulation," ACM Transactions on Modeling and Computer Simulation, vol.1, no.2, pp. 154-193, April 1991.

[86] B.L. Noble, R.D. Chamberlain, "Analytic Performance Model for Speculative, Synchronous, Discrete-Event Simulation," Proceedings of Fourteenth Workshop on Parallel and Distributed Simulation (PADS 2000), Bologne, Italy, pp. 35-44, 2000.

[87] Lee Schruben, "Analytical simulation modeling," Proceedings of the 2008 Winter Simulation Conference, pp. 113-121, 2008.

[88] F. Quaglia, A. Santoro, "Modeling and optimization of non-blocking checkpointing for optimistic simulation on myrinet clusters," Journal of Parallel and Distributed Computing, vol.65, no.6, pp. 667-677, June 2005.

[89] A, Heindl, G. Pokam, "An analytic framework for performance modeling of software transactional memory," Computer Networks: The International Journal of Computer and Telecommunications Networking, vol.53, no.8, pp. 1202-1214, June, 2009.

[90] E. M. Dashofy, A. van der Hoek, and R. N. Taylor, “A Comprehensive Approach for the Development of Modular Software Architecture Description Languages,” ACM Transactions on Software Engineering and Methodology, vol. 14, no. 2, pp. 199-245, April 2005.

[91] R. Fujimoto, “Parallel and distributed simulation,“ Proceedings of 1999 Winter Simulation Conference, Phoenix, AZ, USA, 1999, pp. 122-131.

[92] A. Park and R. Fujimoto, "Optimistic Parallel Simulation over Public Resource-Computing Infrastructures and Desktop Grids," in 12th IEEE International Symposium on Distributed Simulation and Real Time Applications, Vancouver, BC, Canada, 2008, pp. 149-156.

[93] Lj. Samarđić, Z. Radivojević, M. Cvetanović, "Implementacija simulacionog nivoa simulatora diskretnih događaja bazirana na distribuiranoj obradi," ETRAN LIII – Vrnjačka Banja, Srbija, 15-18 Jun 2009.

[94] Z. Radivojević, Lj. Samarđić, M. Cvetanović "Implementation Of The Discrete Event Simulator Based On Distributed Processing" - Software Engineering Education and Reverse Engineering” Ivanjica, Serbia, 6 – 11 September 2010.

[95] The Globus Project, “The Globus Alliance,” Available from: http://www.globus.org, (accessed March 2008).

[96] M. H. Elsheikh, “CellularBASIC: On-phone Open Source Mobile BASIC Interpreter for J2ME,“ http://cellbasic.sourceforge.net, (accessed February 2009).

[97] Murlen, “FScriptME,“ http://fscript.sourceforge.net/fscriptME, (accessed February 2009).

[98] F. Koch, “3APL-M: Platform for Lightweight Deliberative Agents,“ http://www.cs.uu.nl/3apl-m/mprolog.html, (accessed February 2009).

[99] C. Setera, “ZeeME,“ http://setera.org/ZeeME, (accessed February 2009).

[100] D. N. Welton, W. Kechel “Hecl - The Mobile Scripting Language,“ http://www.hecl.org, (accessed February 2009).

[101] S. Whiteside “Simkin Scripting Language,“ http://www.simkin.co.uk/simkin_language.html, (accessed February 2009).

[102] N. Keith and M. Frese, “Effectiveness of error management training: A meta-analysis,” Journal of Applied Psychology, vol. 93, no.1, pp. 59-69, January 2008.

[103] D. Heimbeck, M. Frese, S. Sonnentag, and N. Keith, “Integrating errors into the training process: The function of error management instructions and the role of goal orientation,” Personnel Psychology, vol. 56, no. 3, pp. 333-361, December 2006.

[104] J. Lönnberg, “Student Errors in Concurrent Programming Assignments,“ in Proc. 6th Baltic Sea Conf. Computing Education Research: Koli Calling 2006, Uppsala, Sweden, 2006, pp. 145-146.

[105] A. Zeller, “Making Students Read and Review Code,” ACM SIGCSE Bulletin, vol. 32, no. 3, pp. 89-92, September 2000.

[106] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model Checking Programs," Automated Software Engineering Journal, vol. 10, no. 2, pp. 203-232, April 2003.

[107] E. D. Barlas and T. Bultan. “NetStub: A Framework for Verification of Distributed Java Applications,” Proceedings of the 22nd IEEE/ACM International

Conference on Automated Software Engineering, New York, NY, USA, 2007, pp. 24-33.

[108] Z. Radivojevic, “Konkurentno i distribuirano programiranje,” Available from: http://rti.etf.bg.ac.rs/rti/ri4drs/domaci/Prethodni/index.html (accessed June 2011).

[109] SEE-GRID-SCI, “SEE-GRID eInfrastructure for regional eScience,” Seventh Framework Programme, EC project contract number 211338, 2008-2010.

Прилози

Прилог А: Преглед курсева на Електротехничком факултету

<table>
<tr><td colspan="8">Опис курса</td></tr>
<tr><td>Шифра курса:</td><td>ОО1ОРТ</td><td>Ниво студија:</td><td>Основне</td><td>ЕЦТС</td><td>5</td><td>Семестар:</td><td>2</td></tr>
<tr><td>Назив курса:</td><td colspan="3">Основи рачунарске технике</td><td colspan="3">Година студија:</td><td>1</td></tr>
<tr><td>Предуслови:</td><td colspan="3">Нема</td><td colspan="2">Тип курса:</td><td colspan="2">Изборни</td></tr>
<tr><td>Наставник:</td><td colspan="7">Јован Ђорђевић</td></tr>
<tr><td>Ангажовани:</td><td colspan="7">Ђура Мастиловић, Захарије Радивојевић, Марија Обрадовић</td></tr>
<tr><td>Циљ курса:</td><td colspan="7">✓Увод у логичко пројектовање, прекидачке функције и кола, логичке и меморијске елементе, анализа и синтеза комбинационих и секвенцијалних прекидачких мрежа као и стандардне комбинационе и секвенцијалне модуле.
✓Разумевање структуре и функционисања прекидачких мрежа.
✓Развој вештина потребних за пројектовање прекидачких мрежа састављених од логичких и меморијских елемената, као и од стандардних комбинационих и секвенцијалних модула.</td></tr>
<tr><td>Садржај курса:</td><td colspan="7">1. Булова алгебра. 1.1. Аксиоме. 1.2. Теореме. 1.3. Булова алгебра над скупом два елемента. 2. Прекидачке функције. 2.1. Основе. 2.2. Представљање прекидачких функција буловим изразима. Производи и суме. Дисјунктивне и коњуктивне нормалне форме. Прекидачке функције са једном или две променљиве. 2.3 Представљање нормалних форми помоћу кубова. 3. Минимизација прекидачких функција. 3.1 Основе. 3.2 Одређивање минималне нормалне форме користећи Карноове карте. 4. Функција и структура прекидачких кола. 4.1 Основе. 4.2 Логички елементи. 4.3 Структура и анализа комбинационих мрежа. 4.4 Меморијски елементи. Асинхрони флип-флопови. Синхрони флип-флопови. 4.5 Структура и функција секвенцијалних мрежа. Структурна шема и типови секвенцијалних мрежа. Анализа секвенцијалних мрежа. 5. Синтеза комбинационих мрежа. 5.1 Синтеза комбинационих мрежа користећи НЕ, ИЛИ и И елементе. 5.2 Синтеза комбинационих мрежа користећи НИЛИ и НИ елементе. 6. Синтеза секвенцијалних мрежа. 6.1 Одређивање функција прелаза и излаза секвенцијалних мрежа. Табела и граф стања. Кодирање стања. 6.2 Пројектовање секвенцијалних мрежа. 6.3 Пројектовање мрежа са синхроним флип-флоповима. Синхрони флип-флопови у једноставним и комплексном шемама. 7. Стандардни комбинациони модули. 7.1 Декодер. 7.2 Кодер. 7.3 Мултиплексер. 7.4 Демултиплексер. 7.5 Померач. 7.6 Инкрементер и декрементер. 7.7 Сабирач и одузимач. 7.8 Аритметичка јединица. 7.9 Логичка јединица. 7.10 Атритметичко логичка јединица. 7.11 Компаратор. 8. Стандардни секвенцијални модули. 8.1 Регистар. 8.2 Бројач. 8.3 Регистар са више функција. 8.4 Меморија са равноправним приступом.</td></tr>
<tr><td>Извођење наставе:</td><td colspan="7">45 часова предавања + 30 часова вежби на табли. Два колоквијума након прве и друге трећине курса.</td></tr>
<tr><td>Литература:</td><td colspan="7">1. Основи рачунарске технике, Б. Лазић, Академска мисао, 2006.
2. Основи рачунарске технике 1, http://rti7020.etf.bg.ac.yu</td></tr>
<tr><td>Начин полагања:</td><td colspan="7">Испит – Испит се састоји из три дела. Прва два колоквијума носе по 30 поена, док трећи колоквијум (испит) носи 40 поена. Колоквијуми се могу надокнадити у посебном термину.</td></tr>
<tr><td>Језик наставе:</td><td>Српски</td><td>Датум:</td><td>20.12.2005.</td><td colspan="2">Потпис:</td><td colspan="2"></td></tr>
</table>

<table>
<tr><th colspan="8">Опис курса</th></tr>
<tr><td>Шифра курса:</td><td>OO1ПОТ</td><td>Ниво студија:</td><td>Основне</td><td>ЕЦТС</td><td>3</td><td>Семестар:</td><td>2</td></tr>
<tr><td>Назив курса:</td><td colspan="3">Практикум из основа рачунарске технике</td><td colspan="3">Година студија:</td><td>1</td></tr>
<tr><td>Предуслови:</td><td colspan="3">Нема</td><td colspan="2">Тип курса:</td><td colspan="2">Изборни</td></tr>
<tr><td>Наставник:</td><td colspan="7">Бошко Николић</td></tr>
<tr><td>Ангажовани:</td><td colspan="7">Марија Обрадовић, Захарије Радивојевић</td></tr>
<tr><td>Циљ курса:</td><td colspan="7">✓Упознавање са основним комбинационим и секвенцијалним модулима.
✓Разумевање структуре и функционисања прекидачких мрежа.
✓самосталан развој и симулација једноставнох прекидачких мрежа.</td></tr>
<tr><td>Садржај курса:</td><td colspan="7">Развој и симулација основних комбинационих модула користећи визуелни симулатор: мултиплексер, демултиплексер, декодер, кодер, серијски сабирач, паралелни сабирач, аритметичка јединица, логичка јединица, аритметичко логичка јединица, компаратор.
Развој и симулација основних секвенцијалних модула користећи визуелни симулатор: Флип-флопови (Д, Т, РС, ЈК) реализовани користећи НИ и НИЛИ кола, регистри и бројачи реализовани користећи развијене флип-флопове.</td></tr>
<tr><td>Извођење наставе:</td><td colspan="7">5 часова предавања + 25 часова вежби у лабораторији.
Приближно 30 часова самосталног рада и вежби (1 час недељно током семестра и приближно 15 часова у току испитног рока).</td></tr>
<tr><td>Литература:</td><td colspan="7">1. Основи рачунарске технике 1 – практикум, Ј. Ђорђевић, Б. Николић, З. Радивојевић, Академска мисао, 2004.
2. Основи рачунарске технике, Б. Лазић, Академска мисао, 2006.
3. Основи рачунарске технике 1, http://rti7020.etf.bg.ac.yu</td></tr>
<tr><td>Начин полагања:</td><td colspan="7">Лабораторија – Током семестра, 8 лабораторијских вежби, свака 5 поена.
Испит – Испит траје 2 сата и састоји се из два задатка, носи 60 поена. Да би се положило потребно је на испиту остварити барем 55%.</td></tr>
<tr><td>Језик наставе:</td><td>Српски</td><td>Датум:</td><td colspan="2">17.02.2006.</td><td>Потпис:</td><td colspan="2"></td></tr>
</table>

Опис курса							
Шифра курса:	ИР2ОР2	**Ниво студија:**	Основне	**ЕЦТС**	**6**	Семестар:	3
Назив курса:	Основи рачунарске технике 2			Година студија:			2
Предуслови:	Основи рачунарске технике 1			**Тип курса:**		Обавезни	
Наставник:	Јован Ђорђевић						
Ангажовани:	Захарије Радивојевић, Марија Обрадовић						
Циљ курса:	✓Увод у пројектовање дигиталних система, структура и функција дигиталних рачунара, као и осврт на архитектуру и организацију рачунара. ✓Разумевање структуре и функције дигиталних система и рачунара. ✓Развој вештина потребних за пројектовање дигиталних система састављених од логичких и меморијских елемената, као и од стандардних комбинационих и секвенцијалних модула.						
Садржај курса:	1. **Пројектовање дигиталних система**. 1.1 Операциона и управљачка јединица. Операције. Подаци. Микрооперације. Структура дигиталних система. Структура јединица дигиталних система. Операциона јединица и дијаграм сигнала. Управљачка јединица. Развој операционе и управљачке јединице. 1.2 Операциона јединица и дијаграми података. Основне аритметичке, логичке и померачке операције. Структура операционе јединице и дијаграма тока за случај основних операција. Сложене аритметичке операције. Структура операционе јединице и дијаграма тока за случај сложених операција. 1.3 Управљачка јединица. Реализација помоћу елемената за кашњење. Реализација као стандардна секвенцијална мрежа помоћу флип-флопова, регистара и бројача. Реализација помоћу бројача корака и декодера. Микропрограмска реализација.. 2. **Дигитални рачунари**. 2.1 Структура рачунара. Меморија. Процесор. Улазно/излазни систем. Магистрала. 2.2 Рад рачунара. Фазе у извршењу инструкције. 2.3 Архитектура и организација рачунара. 2.4 Архитектура. Програмски модел. Програмски бројач, регистри, адресе, базни и индексни регистри, регистри опште намене, показивач на врх стека и програмска статусна реч. Типови података. Означени и неозначени цели бројеви. Бројеви у покретном зарезу. Формати инструкција. Троадресни, двоадресни, једноадресни, нулаадресни и променљиви формати. Начини адресирања. Регистарско директно и индиректно, меморијско директно и индиректно, базно, индексно, базно-индексно, аутоинкрементирање, ауто декрементирање, релативно и непосредно. Инструкције преноса, аритметичке, логичке, померачке, и контролне инструкције. Опслуживање прекида. Чување контекста и одређивање адресе прекидне рутине.						
Извођење наставе:	30 часова предавања + 30 часова вежби на табли + 15 часова лабораторијских вежби + колоквијум на средини курса.						
Литература:	1. Увод у архитектуру и организацију рачунара, Б. Лазић, Академска мисао, 2006. 2. Основи рачунарске технике 2, Ј. Ђорђевић, материјали за предавања 3. Основи рачунарске технике 2, материјали за вежбе 4. Материјали за лабораторијске вежбе, http://rti7020.etf.bg.ac.yu						
Начин полагања:	**Испит –** Испит се састоји из два дела, од којих сваки носи 40 поена. Лабораторијске вежбе носе 20 поена. Колоквијум се може надокнадити у посебном термину.						
Језик наставе:	Српски	Датум:	20.12.2005.	Потпис:			

<table>
<tr><th colspan="8">Опис курса</th></tr>
<tr><td>Шифра курса:</td><td>ИР2АР</td><td>Ниво студија:</td><td>Основне</td><td>ЕЦТС</td><td>6</td><td>Семестар:</td><td>4</td></tr>
<tr><td>Назив курса:</td><td colspan="3">Архитектура рачунара</td><td colspan="3">Година студија:</td><td>2</td></tr>
<tr><td>Предуслови:</td><td colspan="3">Основи рачунарске технике 2</td><td colspan="2">Тип курса:</td><td colspan="2">Обавезни</td></tr>
<tr><td>Наставник:</td><td colspan="7">Јован Ђорђевић</td></tr>
<tr><td>Ангажовани:</td><td colspan="7">Захарије Радивојевић, Марија Обрадовић</td></tr>
<tr><td>Циљ курса:</td><td colspan="7">✓Увод у елементе CISC и RISC архитектуре, магистрала, улазно/излазни систем, и систем дискова.
✓Разумевање структуре и функције архитектуре рачунара.
✓Развој вештина како би се омогућило даље изучавање ове области.</td></tr>
<tr><td>Садржај курса:</td><td colspan="7">1. Архитектура. 1.1 CISC и RISC архитектуре. 1.2 Елементи архитектуре. Програмски модел. Типови података. Формати инструкција. Начини адресирања. Скуп инструкција. Стандардне инструкције. Условни скокови. Нестандардне инструкције. Инструкције за рад за стринговима и контроле петљи. Механизам прекида. Спољашњи и унутрашњи прекиди. Приоритет прекида. Селективно и потпуно маскирање прекида. Гнежђење. Прекид после сваке инструкције. Инструкција прекида. 2. Магистрала. 2.1 Основи. Газда/слуга конфигурација. Адресне, линије података и контролне линије. Адресирање унутар уређаја. 2.2 Арбитрација. Процесор. Арбитратор са паралелном и серијском арбитрацијом. Политика арбитрације. 2.3 Циклуси. Асинхроне и синхроне магистрале са атомским и подељеним циклусима. 2.4 Системи са више магистрала. 3. Улазно/излазни систем. 3.1 Основи. Делови. Контролери. Уређаји. Регистри унутар контролера. 3.2 Контролер без директног приступа меморији. Структура и рад. Програмирање узлазно/излазних уређаја користећи испитивање бита спремности и прекида. 3.3 Контролери са директном приступом меморији. Структура и рад. Програмирање узлазно/излазних уређаја. 3.4 Прекиди улазно/излазних уређаја. Појединачно испитивање, векторисан и мешовит приступ. 4. Складишта података. 4.1 Структуре дискова. Механичке компоненте. Електронске компоненте. Технолошки аспекти. 4.2 Карактеристике дискова и контролера дискова. Распоред дискова. Приступ дисковима. Обрада захтева. Кеширање. Бафери, кеш и RAID контролери. 4.3 Интерфејси. Паралелни и серијски ATA и SCSI интерфејс. 4.4 Мрежно засновани системи NAS и SAN архитектуре.</td></tr>
<tr><td>Извођење наставе:</td><td colspan="7">30 часова предавања + 30 часова вежби на табли + 15 часова лабораторијских вежби + колоквијум на средини курса.</td></tr>
<tr><td>Литература:</td><td colspan="7">1. Архитектура рачунара, Ј. Ђорђевић, материјали за предавања
2. Материјали за лабораторијске вежбе, http://rti7020.etf.bg.ac.yu</td></tr>
<tr><td>Начин полагања:</td><td colspan="7">Испит – Испит се састоји из два дела, од којих сваки носи 40 поена. Лабораторијске вежбе носе 20 поена. Колоквијум се може надокнадити у посебном термину.</td></tr>
<tr><td>Језик наставе:</td><td>Српски</td><td>Датум:</td><td>20.12.2005.</td><td colspan="2">Потпис:</td><td colspan="2"></td></tr>
</table>

<table>
<tr><td colspan="8">Опис курса</td></tr>
<tr><td>Шифра курса:</td><td>ИР3АР1</td><td>Ниво студија:</td><td>Основне</td><td>ЕЦТС</td><td>6</td><td>Семестар:</td><td>5</td></tr>
<tr><td>Назив курса:</td><td colspan="3">Архитектура и организација рачунара 1</td><td colspan="3">Година студија:</td><td>3</td></tr>
<tr><td>Предуслови:</td><td colspan="3">Архитектура рачунара</td><td colspan="2">Тип курса:</td><td colspan="2">Обавезни</td></tr>
<tr><td>Наставник:</td><td colspan="7">Јован Ђорђевић</td></tr>
<tr><td>Ангажовани:</td><td colspan="7">Захарије Радивојевић, Марија Обрадовић</td></tr>
<tr><td>Циљ курса:</td><td colspan="7">✓Увод у елементе меморијског хијерархијског система и процесори са проточном обрадом.
✓Разумевање структуре и функције архитектуре рачунара.
✓Развој вештина како би се омогућило даље изучавање ове области.</td></tr>
<tr><td>Садржај курса:</td><td colspan="7">1. Меморија. 1.1 Преклапање приступа меморијским модулима. Брзина приступа меморији. Расподела адреса. 1.2 Кеш меморија. Основи. Технике мапирања. Асоцијативно, директно и сет-асоцијативно пресликавање. Замене блокова. Случајан, fifo, lru као и pseudo lru алгоритми. Освежавање главне меморије. Врати назад, упиши скроз, write allocate и no write allocate. 1.3 Виртуелна меморија и јединица за убрзавање пресликавања. Виртуелна меморије страничне, сегментне и сегментно-страничне организације. Јединица за убрзавање пресликавања са асоцијативним, директним и сет-асоцијативним пресликавањем. 1.4 Повезивање кеш меморије са јединицом за убрзавање пресликавања, улазно/излазним уређајима, и главном меморијом. Виртуелни кеш. Реални кеш. Софтверске и хардверске технике за одржавање конзистентности и кохерентности кеш меморије и улазно/излазних уређаја. Главна меморија са широком речи, меморија са преклопљеним приступом и независним модулима. 1.5 Перформанс кеш меморије. Смањење времена приступа, вероватноће проналаска податка као и кашњења да нема сагласности у кеш меморији.
2. Проточна обрада. 2.1 Основи. Статичка, динамичка, асинхрона и синхрона проточна обрада. 2.2 Инструкцијска проточна обрада. Архитектура процесора. Организације процесора без и са проточном обрадом. 2.3 Хазарди у проточној обради. Структурни хазарди. Регистарски фајл. Кеш меморија за инструкције и за податка. Хазарди података. Прослеђивање. Заустављање. Закашњео пуњење. Контролни хазарди. Фазе у проточној обради за скок. Прекиди. Статичка предикција. Скок није прихваћен. Скок прихваћен. Заустављање. Закашњен скок. Динамичка предикција. Бафер за предикцију. Кеш за предикцију. Предикција са једним и два бита. Корелисани предиктори. Динамичко пуштање инструкција. Scoreboard и Tomasulo.</td></tr>
<tr><td>Извођење наставе:</td><td colspan="7">30 часова предавања + 30 часова вежби на табли + 15 часова лабораторијских вежби + колоквијум на средини курса.</td></tr>
<tr><td>Литература:</td><td colspan="7">1. Архитектура и организација рачунара, Ј. Ђорђевић, материјали за предавања
2. Архитектура и организација рачунара, материјали за вежбе
3. Материјали за лабораторијске вежбе, http://rti7020.etf.bg.ac.yu</td></tr>
<tr><td>Начин полагања:</td><td colspan="7">Испит – Испит се састоји из два дела, од којих сваки носи 40 поена. Лабораторијске вежбе носе 20 поена. Колоквијум се може надокнадити у посебном термину.</td></tr>
<tr><td>Језик наставе:</td><td>Српски</td><td>Датум:</td><td>20.12.2005.</td><td colspan="2">Потпис:</td><td colspan="2"></td></tr>
</table>

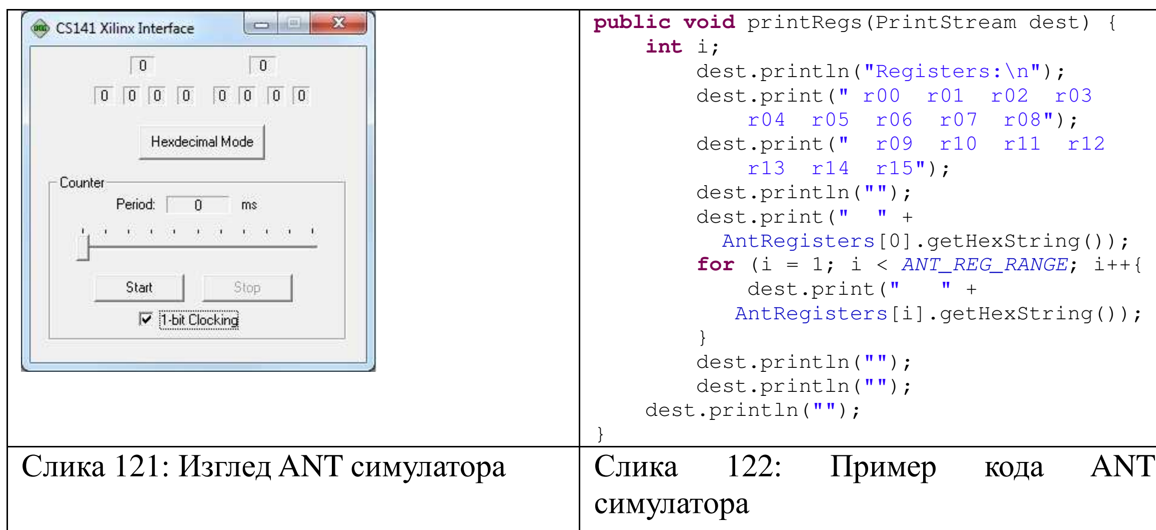
Опис курса							
Шифра курса:	ИР3АР2	**Ниво студија:**	Основне	**ЕЦТС**	**6**	Семестар:	6
Назив курса:	Архитектура и организација рачунара 2			Година студија:			3
Предуслови:	Архитектура рачунара			**Тип курса:**		Изборни	
Наставник:	Јован Ђорђевић, Бошко Николић						
Ангажовани:	Захарије Радивојевић, Марија Обрадовић						
Циљ курса:	✓Увод у технике пројектовања процесора и софтверских анализа за развој графичких симулатора. ✓Разумевање структуре и функције архитектуре рачунара. ✓Развој вештина како би се омогућило развијање архитектура и организација процесора и развој симулатора.						
Садржај курса:	1. **Процесор**. 1.1 Структура процесора. Операциона јединица. Управљачка јединица. 1.2 Архитектура процесора. Програмски модел. Типови података. Формати инструкција. Начини адресирања. Скуп инструкција. Прекиди. Дијаграм тока података, декодовање инструкције читање операнада, извршење операције, смештање резултата и обрада прекида. 1.3 Операциона јединица. Структурна шема са директним везама, са једном, две и три интерне магистрале. Кораци операционе и управљачке јединице, са спојеним и раздвојеном корацима. 1.4 Управљачка јединица. Хардверска реализација. Управљачка јединица са спојеним и раздвојеним корацима. Микро-програмска реализација. Формати микро инструкција, микро програми и управљачка јединица са једним и два типа микро инструкција, и са хоризонталним, вертикалним и мешовитим форматом кодирања сигнала. Нано програмска реализација. Микропрограм, нанопрограм, и управљачка јединица. 2. **Симулатор.** 2.1 Преглед постојећих симулатора и њихових карактеристика. Предности и мане. 2.2 Основне карактеристике симулатора. 2.3 Визуални симулатор пројектованих система. Приказ вредности, сигнала и стања различитих логичких кола. Структурирање и приказ екрана са операционом и управљачком јединицом. Дефиниција понашања основних логичких и комплексних модула. Проблеми приликом симулације пројектованих система. Развој основних функција система као корисничка апликација. Развој асемблерског језика.						
Извођење наставе:	30 часова предавања + 30 часова вежби на табли + 15 часова лабораторијских вежби + колоквијум на средини курса.						
Литература:	1. Архитектура рачунара: Едукациони рачунарски систем, Ј. Ђорђевић, Академска мисао 2004. 2. Архитектура и организација рачунара 2, Ј. Ђорђевић, материјали за предавања 3. Архитектура и организација рачунара 2, материјали за вежбе 4. Материјали за лабораторијске вежбе, http://rti7020.etf.bg.ac.yu						
Начин полагања:	**Испит** – Испит се састоји из два дела, од којих сваки носи 40 поена. Лабораторијске вежбе носе 20 поена. Колоквијум се може надокнадити у посебном термину.						
Језик наставе:	Српски	Датум:	20.12.2005.	Потпис:			

Опис курса							
Шифра курса:	ИР3КДП	**Ниво студија:**	Основне	**ЕЦТС**	**6**	Семестар:	6
Назив курса:	Конкурентно и дистрибуирано програмирање			Година студија:			3
Предуслови:	Програмирање 1, Објектно оријентисано програмирање 2			**Тип курса:**	Обавезан		
Наставник:	Зоран Јовановић						
Ангажовани:	Захарије Радивојевић						
Циљ курса:	✓Упознавање студената са основним концептима конкурентног и дистрибуираног програмирања ✓Увођење појма различитих нивоа апстракције у конкурентном и дистрибуираном програмирању. ✓Оспособљавање студената за писање конкурентних и дистрибуираних програма за најчешће проблеме у различитим програмским језицима						
Садржај курса:	1. **Конкурентно програмирање.** 1.1 Процеси и синхронизација. Locks и баријере, Tie Breaker, Ticket и Bakery алгоритми. Различити начини имплементације баријера. 1.2 Семафори. Расподељени бинарни семафори, технике прослеђивања штафете, алокација ресурса и распоређивање помоћу семафора. 1.3 Монитори. Условне промењиве, дисциплине за сигнал, детаљна анализа дијаграма стања придруженог мониторима. Различити проблеми решени помоћу монитора: Readers/Writers, Interval timer, Sleeping barber, Santa Claus Problem. 1.4 Условни региони. Наредба await. Условни критични региони. Постављање услова блокирања. Различити проблеми решени помоћу региона: One lane Bridge, Dining philosophers, Cigarette Smokers. 2. **Дистрибуирани програмирање.** 2.1 Прослеђивање порука. Асинхроно прослеђивање порука. Филтерске мреже, клијенти и сервери. Синхроно прослеђивање порука. Примене у CSP, Linda, Јава. 2.2 Удаљени позиви процедура – приказ коришћења RMI у Јави. 2.3 Rendezvous – приказ примене у АДА језику.						
Извођење наставе:	30 часова предавања + 30 часова вежби на табли са решавањем задатака + 15 часова практичних вежби у лабораторији, колоквијум на средини семестра. Приближно 70 сати самосталног учења и вежбања.						
Литература:	1. Foundation of Multithreaded, Parallel and Distributed Programming, Gregory Andrews, Addison Wesley, 2000. 2. Конкурентно програмирање: Решени задаци, Игор Икодиновић, Зоран Јовановић, Академска мисао, 2004.						
Начин полагања:	**Испит** –У трајању од 3 сата - програмерски задаци. За полагање испита неопходно је имати 55% поена. **Колоквијум** - замењује 40% комплетног испита. **Пројекат** – замењује 20% комплетног испита.						
Језик наставе:	Српски	Датум:	20.12.2005.	Потпис:			

Прилог Б: Детаљан преглед симулатора из области Архитектуре и организације рачунара

ANT – Симулатор је развијен на Универзитету Харвард ([6] и [7]). ANT симулатор представља једноставну виртуелну машина развијена на основу поједностављене MIPS P2000 архитектуре. ANT развојно окружење садржи три алата: асемблер који конвертује програме писане у асемблеру за ANT симулатор користећи ANT машински језик; симулатор који служи за извршавање програма за ову виртуелну машину; и алата за тражење и праћење грешака унутар ANT програма. Симулатор описују фиксну архитектуру и пружа могућност за писање програма користећи асемблерски језик. Приликом писања програма у асемблерском језику на располагању стоје инструкције, лабеле и макрои. Унутар окружења постоји имплементирано и неколико системских позива који омогућавају програму да приступи деловима система изван виртуелне мешине. Ови системски позиви се могу користити за имплементацију улазно излазних операција. Системски позиви, строго гледано, нису део ANT хардверске архитектуре, већ их треба узети у обзир само као део архитектуре ANT софтверског окружења. Постоје верзије симулатора писаних у програмским језицима C и Јава. Симулатор се може користити на следећим оперативним системима: Windows (95/98/NT/XP), UNIX (SOLARIS, Linux, и OSF/1) и Mac OS-X. Основни циљ због кога је симулатор развијем је педагошка употреба једноставне виртуелне машине. Симулатор је намењен студентима и обезбеђује им основу за савлађивање битних лекција из архитектуре рачунара и асемблерског језика без улажења у конкретне детаље имплементације на неком ниском нивоу. По својој структури ANT симулатор је доста сличан модерним RISC архитектурама тако да се технике обрађене унутар овог симулатора могу пресликати на реалне системе. Студенти којима је првенствено намењен ова симулатор имају могућност да пишу једноставне програме, развијају виртуелну машину и ANT асемблер. Симулатор је дизајниран као основно образовно средство за први курс који обрађује рачунарски хардвер на Универзитету Харвард. На овом курсу се студенти упознају са током података унутар ANT архитектуре. На следећој слици је дат приказ једног прозора ANT симулатора који

служи за повезивање са Xilinx интерфејсом и део изворног кода симулатора.

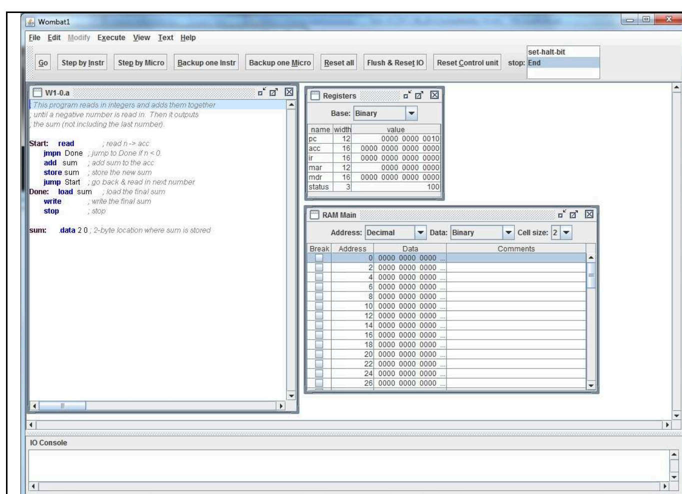



```
public void printRegs(PrintStream dest) {
    int i;
        dest.println("Registers:\n");
        dest.print(" r00  r01  r02  r03
            r04  r05  r06  r07  r08");
        dest.print("  r09  r10  r11  r12
            r13  r14  r15");
        dest.println("");
        dest.print("   " +
          AntRegisters[0].getHexString());
        for (i = 1; i < ANT_REG_RANGE; i++{
            dest.print("    " +
           AntRegisters[i].getHexString());
        }
        dest.println("");
        dest.println("");
    dest.println("");
}
```

Слика 121: Изглед ANT симулатора	Слика 122: Пример кода ANT симулатора

CPU Sim – Симулатор је развијен на Катедри за рачунарство на Колби колеџу у Вотервилу ([8] и [9]). CPU Sim је развијен као интерактивни симулатор процесора развијен користећи Јава програмски језик намењен за употребу на уводним курсевима Организације рачунара. Овде се ради о интерактивном пакету који омогућава симулације у којима корисник задаје детаље везане за симулирани систем који укључује скуп регистара, меморију, скуп миктроинструкција, скуп машинских инструкција, као и скуп асемблерских инструкција. Симулатор CPU Sim је апликација која корисницима омогућава да осмисле једноставан процесор, на нивоу микрокода као и да покрећу симулације програма писаних на машинском или асемблерском језику. Симулатор се може користити за симулацију различитих архитектура, укључујући и једноадресне које користе акумулатор, троадресне налик на RISC архитектуре, или стек машине (налик на Јавину виртуелну машину). Током симулације овај симулатор интерпретира задати модел и његов асемблерски језик. Овај симулатор је развијен користећи Јава програмски језик. Може се покренути на Windows, Linux и Mac OS. Симулатор представља потпуно интегрисано развојно окружење које укључује већи број алата и функција. Доступни алати служе за пројектовање процесора на нивоу трансфера између регистара, едитор текста за писање асемблерских програма. Програми написани у асемблерском језику се након тога конвертују у машински језик за циљни пројектовани процесор. Током симулације, могуће је

користити уграђени алат за праћење и отклањање грешака. Алат за отклањање грешака омогућава кориснику да се током симулације помера напред и назад током извршавања програма, да испита и опционо да промени стање пројектованог процесора након извршења сваког корака. Симулатор је развијен да помогне инструкторима који желе студентима да осмисле такве курсеве на који би студенти сами пројектовали различите архитектуре и да добију прилику да планирају, примене, и пишу програме на машинском језику и асемблерском језику. Симулатор је био коришћење на уводним курсевима Организације рачунара на Колби колецу. На следећој слици је дат приказ једног прозора CPU Sim симулатора код кога се види програм писан на асемблерском језику и стање регистара, као и део изворног кода симулатора.



Слика 123: Изглед CPU Sim симулатора

```
public class Register extends Module {
    private long value;
    private int width;
    public Register(String name, int width){
        super(name);
        value = 0L;
        setWidth(width);
    }
    public int getWidth() {
        return width;
    }
    public long getValue() {
        return value;
    }
    public void setWidth(int w) {
        Assert.That(w > 0,
"Register.setWidth() called with a parameter
<= 0");
        if (w < width) setValue(0L);
        width = w;
    }
```

Слика 124: Пример кода CPU Sim симулатора

DigLC2 – Симулатор развијен је на Ecole Polytechnique, University Paris-Sud у Француској ([10] и [11]). DigLC2 је развијен као симулатор на нивоу дигиталних логичких кола који описује Little Computer 2 (LC-2) архитектуру. Симулатор пружа детаљан опис свих компоненти процесора на нивоу логичких кола, тако да на тај начин демистификације архитектуру процесора. Користећи овај симулатор могуће је изградити и сложене организације процесора које укључују и проточну обраду као и узлазно излазне уређаје како што је DMA контролер. Као основни градивни блокови се могу користити основна логичка кола (И, ИЛИ, НЕ и Тростатички бафер) али и компоненте које је пројектовано сам корисник.

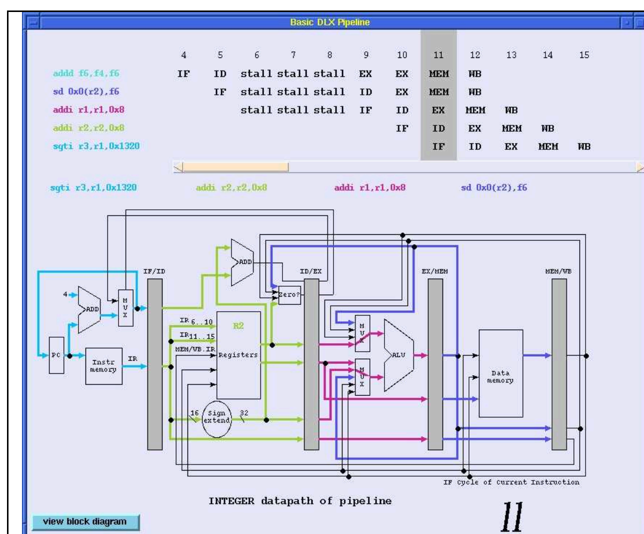
Симулатор је реализован користећи C програмски језик и могуће га је користити на UNIX и Windows оперативним системима. Симулатор DigLC2 представља компромис између високог нивоа моделирање структуре дигиталних кола и скупих хардверских тест окружења. Симулатор је намењен студентима који прате курсеве на којима се представља принцип пројектовања одоздо-навише. Ово студенти би били директно укључени у дизајн сваког од делова процесора истражујући разна питања дизајна хардвера. Симулатор се може користити и за разумевање тока података и контролних структура, са ширим погледом на процесор и пројектовање читавог система. DigLC2 симулатор се користи као помоћно образовно средство на уводном курсу архитектуре рачунара на École Polytechnique у Француској. На следећој слици је дат приказ једног прозора DigLC2 симулатора као и део изворног кода симулатора.

```
Static Void log_16_output(lact, n, v)
log_action *lact;
log_nrec *n;
log_16_value v;
{
  nodeinfo *ni;
  ni = (nodeinfo *)n->info;
  if ((int)v > 2)
    v = log_one;
  if (ni->v0 == log_none)
    ni->v0 = v;
  else if (ni->v0 != v && v != log_none)
    (*lact->hook.nodeconflict)(n);
}
```

Слика 125: Изглед DigLC2 симулатора	Слика 126: Пример кода DigLC2 симулатора

DLXview – Симулатор развијен је на Универзитету Пердју на факултету за Електротехнику и рачунарску технику као компонента CASLE пројекта ([12] и [13]). DLXview је интерактивни симулатора процесора са проточном обрадом који користи DLX скуп инструкција и симулира три верзије процесора са проточном обрадом: основни, систем са распоређивањем (scoreboard) и користећи Томасуло алгоритам. Код овог симулатора су само највиши нивои организације DLX процесора видљиви као и неки делови који имплементирају Томасуло и scoreboard алгоритам. Алгоритам за резервацију се задају као параметар на почетку симулације. Поред алгоритма на почетку се задаје и програм који је потребно симулирати. Овај програм је могуће писати користећи асемблерски језик који

подржава рад са инструкцијама и лабелама. Симулатор интерпретира инструкције које извршава и подржава кретање унапред и уназад за један такт у циљу испитивања понашања симулираног система. Симулатор је развијен коришћењем програмски језик C, и аутори су га тестирали на Solaris, SunOS, HP-UX, и Linux оперативним системима. Главни циљ DLXview је да обезбеди визуелно, интерактивно окружење у коме студенти могу лакше да разумеју рад процесора са проточном обрадом. Симулатор омогућава кориснику да током симулације може да прати: стање и инструкције унутар сваког степена проточне обраде, вредности у табелама распоређивања у сваком такту на завршетку сваке инструкције. Поред овог свака операциона јединица унутар степена проточне обраде која ради са целим и бројевима у покретном зарезу су дати на нивоу шема и сви делови су јасно истакнути. Симулатор се може користити у образовне сврхе за увођење процесора који користе проточну обраду на курсевима архитектуре и организације рачунара. Симулатор се користио на курсевима на основним и постдипломских студија за експериментисање са системом код кога се може мењати броја регистара, кашњење појединих степена протоне обраде, као и коришћење разних оптимизација. На следећој слици је дат приказ једног прозора DLXview симулатора DLX процесора код кога се утврђивање да ли је дошло до скока обавља у другом степену и део изворног кода симулатора.

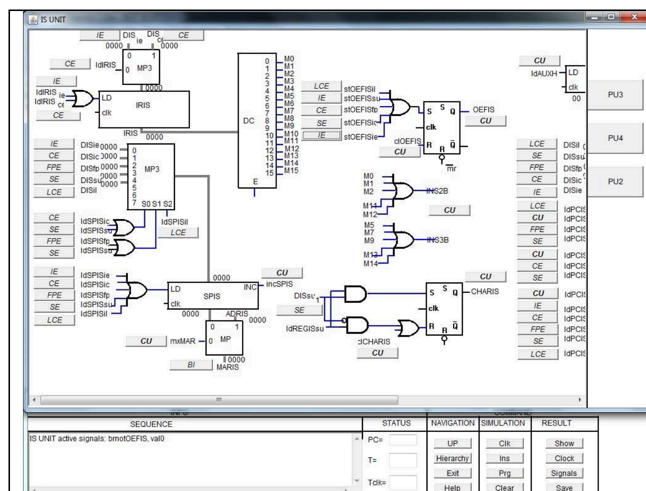



```
void statsReset(machPtr)
      DLX *machPtr; /* machine
description */
{
    int i;
    machPtr->stalls = 0;
    machPtr->branchYes = 0;
    machPtr->branchNo = 0;
    for (i = 0; i <= OP_LAST; i++)
        machPtr->operationCount[i] = 0;
}
```

Слика 127: Изглед DLXview симулатора

Слика 128: Пример кода DLXview симулатора

EDCOMP - Симулатор је развијен на Електротехничком факултету Универзитета у Београду ([14] и [15]). EDCOMP је образовни рачунарски систем заснован на коришћењу Веб технологија који симулира рачунарски систем на RTL нивоу. EDCOMP укључује CISC процесор, главну меморију, улазно/излазни подсистем са контролером за директан приступ меморији (DMA), контролер без директног приступа меморији, шест контролера периферија, и арбитратор. Веб засновани графички симулатор подржава анимацију извршења симулација и омогућава студентима да пишу сопствене програме користећи асемблерски језик; да преводе такве програме, интерактивно подешавају и испитују вредности меморијских локација, регистара, и улазно/излазних уређаја, као и да самостално покрећу симулацију. Симулатор омогућава користи да покренете симулацију такт по такт, инструкцију по инструкцију, као и да омогући кориснику да подеси да се извршавање програма паузира уколико се достигне одрешена вредност. Изворни код симулатор је писан користећи Јава програмски језик. Може се покренути на Windows, Linux и Mac OS, или било који други оперативни систем са Јава виртуелном машином. EDCOMP је флексибилно образовно окружење засновано на коришћењу Веб технологија дизајнирано да помогне у настави и учењу на курсевима архитектуре и организације рачунара. Може се користити у образовне сврхе за увођење CISC сета инструкција, начина адресирања, организације рачунара, рада са целим бројевима, као и са бројевима у покретном зарезу њиховом аритметиком, обрадом прекида, основама улаза и излаза, магистралама и арбитрацији. Симулатор се користи на Београдском универзитету. На следећој слици је дат приказ једног прозора EDCOMP симулатора који представља једну операциону јединицу CISC процесора као и део изворног кода симулатора.



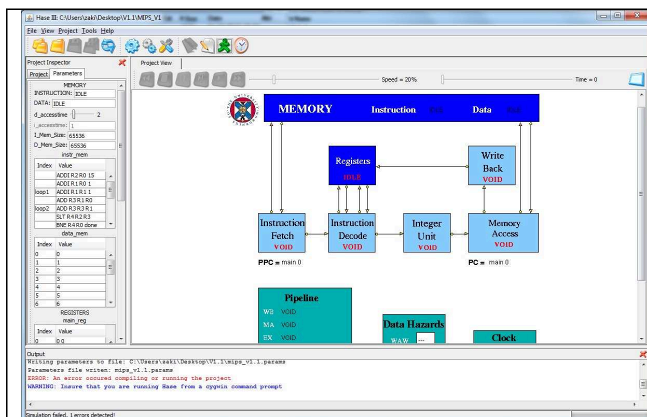
Слика 129: Изглед EDCOMP симулатора

```
void izracunaj_periferija(int rbr){
    if(pstart[rbr]){
        brojregper[rbr]= 0;
    }else if(incCOUNTERper[rbr]){
        brojregper[rbr] ++;
    }
    for (int i = 0; i<16; i++){
        if (stVm[rbr][i]){
REGperposle[rbr][i]=ADRper[rbr];
        }
    }
    for (int i = 0; i<16; i++){
        if (stVm[rbr][i]){
            Vm[rbr][i]=true;
        }else if(clVm[rbr]){
            Vm[rbr][i]=false;
        }
    }
```

Слика 130: Пример кода EDCOMP симулатора

HASE – (Hierarchical Computer Architecture design and Simulation Environment) симулатор је развијен на Групи за рачунарске системе на факултету за информатику на Универзитет у Единбургу ([16] и [17]). HASE обухвата скуп алата који омогућавају да се конфигуришете рачунарски систем повезивањем расположивих модула и да се симулирају добијени системи. Симулатор омогућава дизајнеру да се креће у оквиру хијерархијских нивоа и да изаберу најпогоднији ниво за различите делове система. Скуп пратећих алата омогућава креирање нових, корисничких, модула који се могу користи као грађевински блокови за стварање нових симулатора. Ови модули се могу додатно конфигурисати, што укључује и коришћење шаблона који су идеални градивни блокови за дизајн прототипова и анализу проширивости архитектура. Симулациона анализа креираних модула је могуће захваљујући алатима која омогућава бележење догађаја који би касније бити представљени на кориснички дефинисаним временским графицима. Компоненте и модули се унутар симулатора третирају као објекти. Ово је постигнуто кроз развој посебног објектно оријентисаног симулационог језика HASE++. Ово симулационо окружења онда тумачи HASE++ код. Додати алат JavaHASE служи за претварање HASE пројеката за Јава аплете. Симулатор може користити на следећим оперативног система: Solaris, Linux, и Windows. Алати унутар HASE окружења обухватају и опште наменско графичко окружење које омогућава кориснику да креира модуле за хијерархијски рачунарски систем специфичне архитектуре која онда може бити симулиране и анализирани. Процес симулације се може пратити коришћењем анимираног мода

рада за праћење тока симулације. Симулатор је првенствено намењен студентима, али га могу користи особе које раде развој и тестирање компонената, за образовне сврхе пројектоване архитектуре, за прелиминарно тестирање и анализу перформанси пројектованог система. HASE је коришћен у следеће пројекте: хијерархијска PRAM студијама, језику за паралелно програмирање H-FORK, EMIN пројекту, на моделу за паралелно рачунања на 2-D мрежа, на процени перформанси мултипроцесорских интерконекционих мрежа, као и на модел Стенфорд DASH архитектуре, са циљем провере протокола кохерентности кеш меморије. На следећој слици је дат приказ једног прозора HASE симулатора у кога је учитан процесор са проточном обрадом како развијена компонента, као и део изворног кода симулатора.



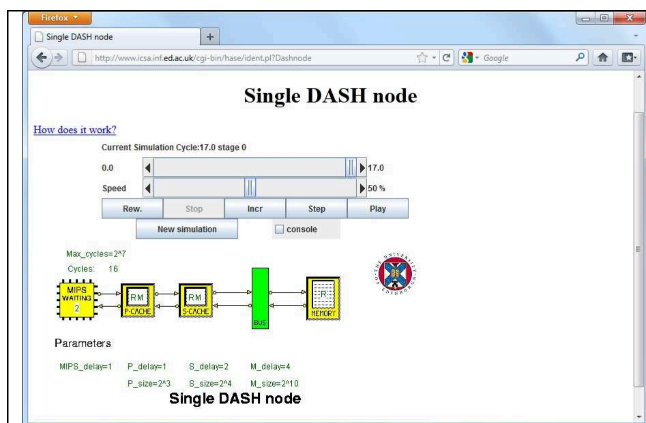
Слика 131: Изглед HASE симулатора

```
public Port(String szPortName, Entity
pEntity,
        int nX, int nY, ImageIcon imgIcon)
{
        m_bLegacy = false;
        m_szPortName = szPortName;
        m_pEntity = pEntity;
        m_nXOffset = nX;
        m_nYOffset = nY;
        m_imgIcon = imgIcon;
        m_pLink = null;
        rePosition();
}
```

Слика 132: Пример кода HASE симулатора

HASE Dinero - Симулатор је развијен на Групи за рачунарске системе на факултету за информатику на Универзитет у Единбургу ([18] и [19]). Симулатор представља графичко окружење за симулацију кеш меморије. Организација симулатора је дата само на највишем нивоу хијерархије. Параметри који је задају на почетку симулације су: величина речи, величина кеш линије, величину сета, број сетова, политику замене, политику ажурирања, политику алокације приликом промашаја код операције уписа, итд. Симулатор омогућава коришћење до 10.000.000 линија унутар датотеке са приступима који ће бити обрађен. Симулатор је развијен користећи HASE симулатор и одговарајући скрипт језик. Симулатор може користити на истим оперативним системима као и HASE симулатор (Solaris, Linux, и Windows) са том разликом што је користећи развијено

JavaHASE окружење развијен и одговарајући аплет који се може користити на свим оперативним системима који подржавају Јаву. Симулатор има два режима рада: анимирани и брзо. У Анимираном моду, све активности током симулације су видљиве и током репродукције симулације. Намера коришћења анимираног режима рада је била да се студентима омогући боље разумевање рада кеш меморије које се може добити посматрањем како се одређује сагласност унутар кеш меморије и како различите конфигурације које се варирају имају на овај процес. Брзи режим је по питању брзине далеко ефикасније него Анимирани режим, али не постоји континуално и визуелно праћење активности током симулације већ се само посматра резултат на крају. Симулатор могу да користе студенти у образовне сврхе. На следећој слици је дат приказ једног прозора HASE Dinero симулатора који је учитан у HASE окружење, као и део изворног кода симулатора.



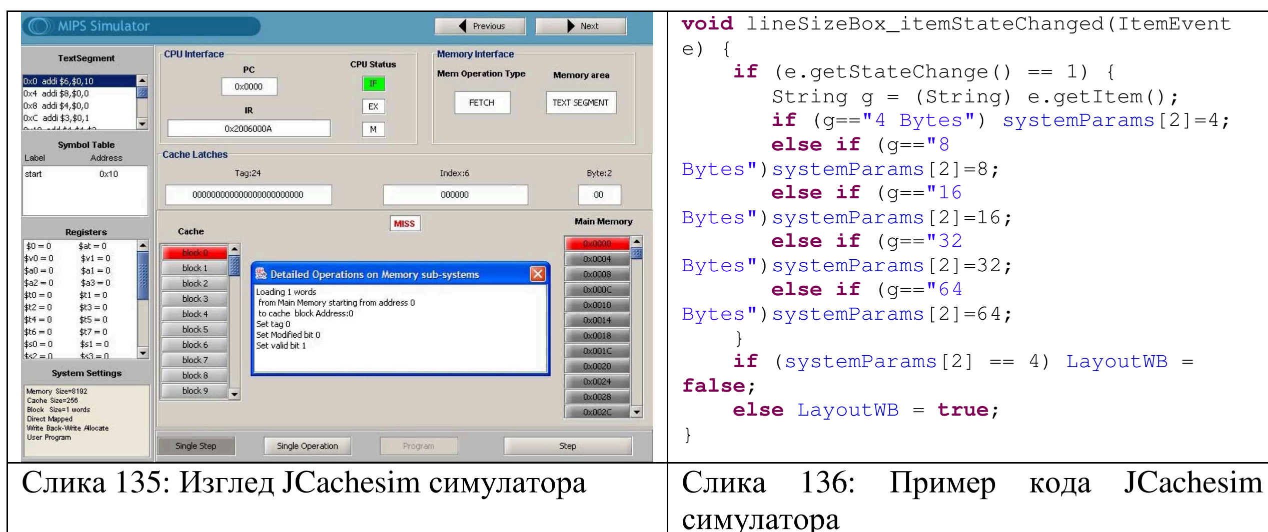
Слика 133: Изглед HASE-Dinero симулатора

```
void
CACHE_CONTROL_UNIT::calculate_params(){
    CONTROL2 *basic;
    basic = (CONTROL2 *) sim.get_entity(
        sim.get_entity_id("control2"));
    unified_cache = false;
    if(basic->Unified_Cache_Size)
        unified_cache = true;
    for(int i=0; i<32; i++){
        if(pow(2, i) == basic-
>Block_Size){
            block_bits = i;
            break;
        }
    }
    block_offset_mask = 0;
    for(int i=0; i<block_bits; i++){
        block_offset_mask += pow(2, i);
    }
```

Слика 134: Пример кода HASE-Dinero симулатора

JCachesim – Симулатор је развијен на Универзитету у Сијени на факултету за рачунарску технику ([20] и [21]). JCachesim је Јава алат, који се користи за експериментисање са понашањем кеш меморије приликом извршавања једноставних програма и варирањем карактеристика дате кеш меморије. Симулатор представља MIPS процесор на највишем нивоу организације. Параметри који се могу поставити у овом симулатору су: ширина магистрале, величина кеш меморије, величину сета, број сетова, политику замене, политику ажурирања, политику алокације приликом промашаја код операције уписа и

величина оперативне меморије. Корисници могу да пишу код користећи асемблерски језик. Могуће је пратити регистре и меморију, али и статистику кеш меморије. Статистика укључују процесор, магистралу, праћење и памћење опште и статистике везане за детаље кеш меморије. Симулатор је развијен користећи Јава програмски језик и може се извршавати на Windows, Linux и Mac OS, или било који други оперативни систем са Јава виртуалном машином. JCachesim је симулационо окружење за процену перформанси рачунарских система заснованих на MIPS архитектури са кеш меморијом. Образовни визуелни алат у Јава има уграђен карактеристике које омогућавају предавачу да прате напредак сваког појединачног студента у току рада са симулатором. Симулатор омогућава студентима да посматрају активности процесора и кеш меморије током извршавања програма, а нарочито за време читања и писања у меморију, да процени перформансе система, да анализирају локалност и расподелу приступа меморије током извршење програма. Симулатор се користи на курсевима архитектуре рачунара за проучавање и анализу перформанси рачунара са кеш меморије. На следећој слици је дат приказ једног прозора JCachesim, као и део изворног кода симулатора.



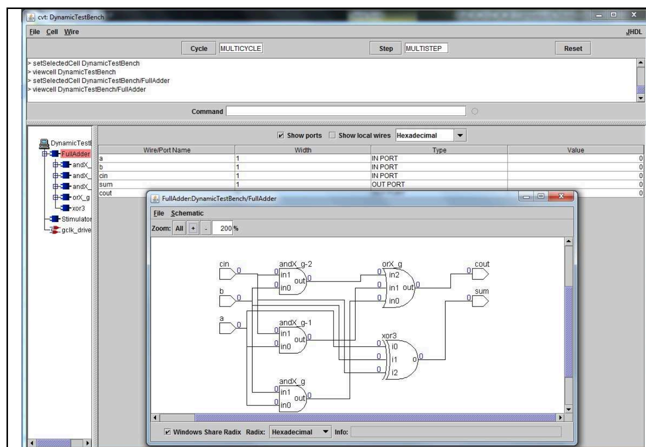

```
void lineSizeBox_itemStateChanged(ItemEvent
e) {
    if (e.getStateChange() == 1) {
        String g = (String) e.getItem();
        if (g=="4 Bytes") systemParams[2]=4;
        else if (g=="8
Bytes")systemParams[2]=8;
        else if (g=="16
Bytes")systemParams[2]=16;
        else if (g=="32
Bytes")systemParams[2]=32;
        else if (g=="64
Bytes")systemParams[2]=64;
    }
    if (systemParams[2] == 4) LayoutWB =
false;
    else LayoutWB = true;
}
```

Слика 135: Изглед JCachesim симулатора

Слика 136: Пример кода JCachesim симулатора

JHDL - (Just-Another Hardware Description Language) симулатор је развијен на Универзитету Бригам Јанг ([22], [23] и [24]). JHDL симулатор је структурално заснован на језику за опис хардвера (HDL) реализован у програмском језику Јава. Симулатор омогућава кориснику да различите нивое симулатора реализује или

користећи алат за моделовање компонената или користећи посебан језик за њихов опис стварајући притом хијерархијску структуру. На овај начин корисник има могућност да креира нове градивне блокове које може даље користити приликом развоја сложенијих система. Овако креирани модули се могу конфигурисати, што укључује и симулацију шаблона које се након тога могу користити као градивни блокови за израду прототипова дизајна сложених компоненти и за анализу скалабилности архитектура. Током симулације шематски приказ резултата симулације може да приказује или симулиране вредности или вредности преузете директно са FPGA платформе реализоване користећи исти интерфејс за опис компонената за време њеног извршења. Системи који се развијају користећи овај алат се могу симулирати или користећи сам симулатор или користећи развијени интерфејс и на наменском програмабилном хардверу FPGA. Сам симулатор је развијен користећи Јава програмски језик и може се извршавати на Windows, Linux и Mac OS, или било који други оперативни систем са Јава виртуалном машином. У оквиру овог симулатора постоје доступни алати који омогућавају отклањање грешака, симулацију, тестирање и израду интерфејса за логичка дигитална кола, како оних који постоје унутар симулатора и тако и оних који се извршавају на FPGA. Симулатор JHDL као алат за рад са FPGA компонентама покрива три главне области: структуре или организације дигиталних логичких кола, распоред кола, а интерфејс према FPGA компонентама Симулатор је намењен студентима, особама које се баве развојем компонентама, тестери, и особе које се баве анализама перформанси. Симулатор се користи као CAD алат на Бригам Јанг универзитету и студенти да интензивно користе за развој дигиталних система високих перформанси у аутоматизованом препознавању циљева, код сонара, као и на другим пројектима. На следећој слици је дат приказ једног прозора JHDL са реализованим потпуним једноразредним сабирачем, као и део изворног кода симулатора.



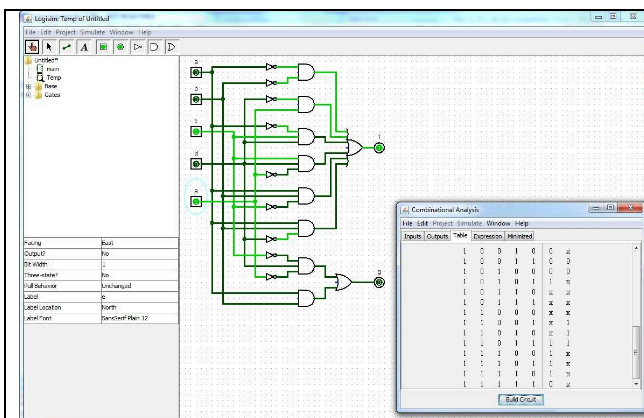
Слика 137: Изглед JHDL симулатора

```
public Wire getDefaultClock() {
    Wire clock = super.getDefaultClock();
    if (clock == null) {
      HWSystem system = getSystem();
      Cell tb = system.getTestBench();
      clock = wire(tb,
"GLOBAL_DEFAULT_CLOCK");
      system.libraryDefaultClock();
      tech_mapper.clockDriver(tb,
          clock, "01", "gclk_driver");
      setDefaultClock(

system.declareExternalClock(clock));
      system.setUsingImplicitClock();
    }
    return clock;
  }
```

Слика 138: Пример кода JHDL симулатора

Logicsim - Симулатор је развио Карл Бурцх са Хендрикса колеџу као образовни алат за пројектовање и симулацију дигиталних логичких кола ([25] и [26]). Алат омогућава креирање нових модула који се могу користи као градивни блокови за стварање нових симулатора. Поред графичког начина пројектовања симулатор омогућава унос дигиталних кола користећи или табелу улаза/излаза или унос једначина које описују улаз излаз. Користећи ове начине могу се добити једначине које се касније могу минимизовати и аутоматски трансформисати у дигитална логичка кола. Симулатор је развијен користећи Јава програмски језик. Симулатор поседује једноставан интерфејс и скуп функционалности тако да омогућава развој и симулацију дигиталних система. Овакво поједностављено решење је намењено да олакша учење најосновнијих појмова у вези са логичким колима. Овај симулатор користе студенти на колеџима и универзитетима широм света на различитим курсевима који покривају области логичког пројектовања, тако да ови курсеви укључују курсеве на којима се само даје кратак преглед области рачунарских наука, на курсевима организације рачунара, па чак и на курсевима који трају читав семестар а посвећени су архитектури рачунара. Logicsim се може користити за пројектовање и симулацију целог процесора у едукативне сврхе. Симулатор се користи као CAD алат на: Curtin International College, Ecole Supérieure d'Informatique - Haute Ecole de Bruxelles, University of Sao Paulo, Augustana U - Alberta, Putian University, Universidad Pontificia Bolivariana - Medellin, Universidad EAFIT- Medellin, University of Grenoble, University of Rennes, University of Iceland, Holy Cross Matriculation Higher Secondary School, Victoria

University of Wellington, University of Lugano, National Chi Nan University, University Akron, Cal Poly Pomona, Christopher Newport University, Colgate University, Coll of St Benedict/St John's University - Minnesota, Denison University, DePaul University, Knox Coll, Mississippi Coll, New York University, University California - Berkeley, University Massachusetts - Amherst, University Massachusetts - Lowell, Virginia Tech. На следећој слици је дат приказ једног прозора Logicsim симулатора који је аутоматски реализовано комбинациону мрежу на основу табеле улаза/узлаза, као и део изворног кода симулатора.



Слика 139: Изглед Logisim симулатора

```
public void propagate(CircuitState state) {
    PinAttributes attrs = (PinAttributes)
        getAttributeSet();
    Location pt = getEndLocation(0);
    Value val = state.getValue(pt);
    State q = getState(state);
    if(attrs.type == EndData.OUTPUT_ONLY) {
        q.sending = val;
        q.receiving = val;
        state.setValue(pt,
Value.createUnknown(
            attrs.width), this, 1);
    } else {
       q.receiving = val;
       if(!val.equals(q.sending)) {
         state.setValue(pt, q.sending, this,
1);
       }
    }
}
```

Слика 140: Пример кода Logisim симулатора

M5 - Симулатор је развијен у оквиру лабораторије за Напредну архитектуру рачунара на департману за Електротехнику и рачунарске науке Универзитета у Мичигену ([27] и [28]). M5 симулатор пружа могућности неопходне за симулацију мреже повезаних рачунара, укључујући могућност комплетне симулације система, детаљног подсистема улаза/излаза, као и способност да детерминистички симулира више умрежених система. Симулатор укључује алате за дефинисање, параметризацију, инстанцирање и проверу стања симулираних објеката. Симулатор такође обезбеђује интерфејс за изградњу система састављених од колекција сложених објеката. Симулатор садржи и доста развијене пакете за прикупљање и обраду статистике као и значајан скуп алата за подршку провери, анализи и отклањању грешака. M5 је имплементиран помоћу два објектно оријентисана језика: Python за конфигурацију објеката високог нивоа и

симулацију користећи скрипте, и C++ за пројектовање објеката ниског нивоа. Све симулирани објекти (процесора, магистрала, кеш итд) су представљене као објекти у оба програмска језика. Симулатор је развије као симулатор читавог система "full-system" који подржава подизање целог оперативног система, као и апликације дате у облику бинарних фајлова са емулацијом системских позива. М5 ради на већини оперативних система (Linux, MacOS X, Solaris, OpenBSD, Cygwin) и архитектуре (x86, x86-64, SPARC, Alpha, и PPC). Иако је овај симулатор намењен за симулацију умрежених система, М5 је структурирано дизајниран и поседује богат скуп функција, као и библиотека са унапред дефинисаним објектима тако да обезбеђује оквир за изградњу комплексних симулације архитектура у различитим доменима. Симулатор могу користити дизајнери хардвера и софтвера, студенти, тестери, и аналитичари перформанси. Симулатор се може користити и за реверзибилна извршења и отклањање грешака, као и за потпуну симулацију система, мреже различитих система које садрже вишепроцесорске/мулти-системске могућности. Симулатор се користи на универзитету у Мичигену, универзитету у Висконсину - Медисон, Нортвестерн универзитета, и Ајова Стате Университу. На следећој слици је дат приказ излаза М5 симулатора, као и део изворног кода симулатора.

```
root@rti29:/storage/exp_soft/m5-2.0b3/gem5
[root@rti29 gem5]# build/ARM_SE/gem5.opt configs/example/se.py -c tests/test-pro
gs/hello/bin/arm/linux/hello
[root@rti29 gem5]#
[root@rti29 gem5]#
[root@rti29 gem5]#
[root@rti29 gem5]#
[root@rti29 gem5]# gem5 Simulator System. http://gem5.org
gem5 is copyrighted software; use the --copyright option for details.

gem5 compiled April 10 2011 20:15:31
gem5 started April 10 2011 20:19:15
gem5 executing on rti29
command line: ./build/ARM_SE/m5.opt configs/example/se.py
Globale frequency set at 1000000000000 ticks per second
0: system.remote_gdb.listener: listening for remote gdb #0 on port 7000
**** REAL SIMULATION ****
info: Entering event queue @ 0.  Starting simulation...
Hello world!
hack: be nice to actually delete the event here
Exiting @ tick 3188500 because target called exit()
[root@rti29 gem5]#
```

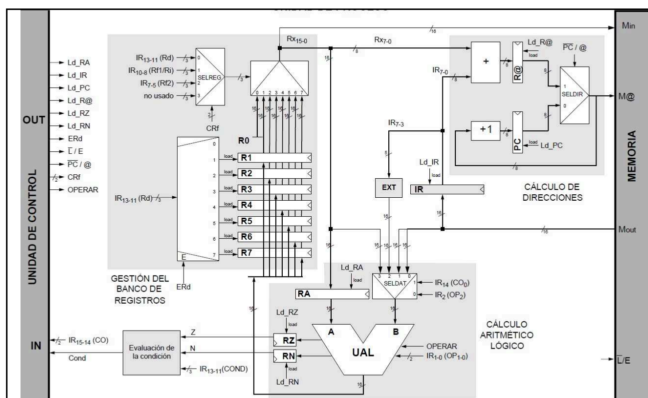
Слика 141: Изглед М5 симулатора

```
SimLoopExitEvent * simulate(Tick num_cycles){
...

    while (1) {
...
        curTick = mainEventQueue.nextTick();
        Event *exit_event =
            mainEventQueue.serviceOne();
        if (exit_event != NULL) {
            SimLoopExitEvent *se_event;
            se_event =
                dynamic_cast<SimLoopExitEvent*>
                  (exit_event);
            if (se_event == NULL)
              panic("Bogus exit event class!");
            if (se_event != limit_event) {
              assert(limit_event->scheduled());
                limit_event->deschedule();
                delete limit_event;
            }
    return se_event;
}
```

Слика 142: Пример кода М5 симулатора

RM – (Рудиментарна машина) симулатор је развијен на Катедри за рачунарску архитектуру на Техничком универзитету Каталоније (UPC) ([29], [30] и [31]). Код RM симулира омогућено је праћење извршавања инструкције код система са

једноставним архитектурном развијеном у RISC стилу. Систем не даје могућности за праћење потпрограма и обраду прекида. Графички интерфејс симулатора је тако развијен да даје само јединице процесора на највишем нивоу организације. Симулатор поседује фиксну архитектуре, и даје могућност корисницима да користећи посебан асемблерски језик пишу код који је касније могуће симулирати. Наменски асемблерски језик за дату симулирану машину је развијен како би се поједноставило писање кода. Овај асемблерски језик садржи следеће компоненте: инструкције, лабеле, изразе и макрое. Стање симулације се може пратити прати такт по такт, инструкцију по инструкцију или коришћењем прекиде извршавања на одређеној линији, вредности променљиве или сигнала. Могуће је пратити и мењати садржаје регистара и меморије у сваком тренутку. Симулација се извршава као стандардни Windows апликација. RM симулатор је намењен да буде помоћно наставно средство које има за циљ да објасни основне принципе архитектуре рачунара студентима. Основни циљ употребе овог симулатора на неком од курсева архитектуре и организације рачунара је једноставност и особина ортогоналности. Симулатор је првенствено намењен студентима да га користе у образовне сврхе у циљу праћења и анализе дијаграма сигнала операционе и управљачке јединице. Овај симулатор се користи на првом курсу логичког дизајна и архитектуре рачунара у рачунарским наукама и рачунарском инжењерству на основним студијама Политехнике у Каталонији (Politecnica de Catalunya), Шпанија. На следећој слици је дат приказ основног екрана RM симулатора, као и део изворног кода симулатора.



Слика 143: Изглед RM симулатора

```
BOOL FAR PASCAL DlgProcDialogoSenyales(HWND hWnd,
UINT messg, WPARAM wParam, LPARAM lParam){
    switch(messg){
            case WM_INITDIALOG:
                InicializarDialogoSenyales(hWnd);
                break;
            case WM_COMMAND:
                switch(LOWORD(wParam)){
                    case IDOK:
                        AceptarSenyal(hWnd);
                        EndDialog(hWnd,IDOK);
                        break;
                    case IDCANCEL:
                        CancelarSenyal(hWnd);
                        EndDialog(hWnd,IDCANCEL);
                        break;
                    case CM_ANYADIR:
                        AnyadirSenyal(hWnd);
                        break;
                    case CM_QUITAR:
                        QuitarSenyal(hWnd);
                        break;
```

Слика 144: Пример кода RM симулатора

RSIM – (Rice Simulator for ILP Multiprocessor) симулатор је развијен на Универзитету Рајс Хјустон Тексас (Rice University Houston Texas), САД ([32],[33] и [34]). RSIM симулатор служи за анализу дељене меморије код вишепроцесорских и једнопроцесорских система са паралелизмом на нивоу инструкција. Симулатор описује напредну архитектуру која поседује паралелизам на нивоу инструкција код које корисник има могућност да пише код у асемблеру. Представљена архитектура је развијена на основу података о карактеристикама тренутно доступних процесора, али је најближа MIPS R10000 архитектури. Поред ових напредних карактеристика проточне обраде у системима са једним процесором и вишепроцесорских система симулатор описује и меморијски систем, укључујући кеш меморију реализовану у два нивоа хијерархије. Приликом писања асемблерског кода корисник има на располагању скуп инструкција који је сличан скупу инструкција код SPARC V9 arhitekture. Током симулације корисник може да прати следеће податке: стање сваког степена проточне обраде и стања сваке операционе јединице процесора у сваком такту и у времену када је нека јединица процесора заустављена. Поред могућности праћења симулатор има могућност генерисања одговарајуће статистике симулације. Статистика може да садржи податке који се односе на различите фазе симулације. Овај симулатор је развијен коришћењем C и C++ програмског језика. Код за меморијски систем је развијен унутар окружења RPPT (the Rice Parallel Processing Testbed). RSIM је реализован као догађајима вођен симулатор дискретних догађаја користећи симулациону библиотеку YACSIM. Симулатор може користити на следећим системима: SUN ULTRASPARC work stations, SUN SparcStations, и SGI PowerChallenge systems, као и Unix компатибилни. RСИМ је првенствено намењен за проучавање дељене меморије вишепроцесорских организација које у великој мери користе паралелизам на нивоу инструкција. Овај симулатор се користи као истраживачки алат и у настави курсева архитектуре рачунара укључујући једно процесорске и мултипроцесорске системе на дипломским и постдипломске студије на Универзитету Рајс Хјустон Тексас, САД. На следећој слици је дат приказ излаза RSIM симулатора, као и део изворног кода симулатора.

```
DUSESIGNAL -DUNELF -DNO_IEEE_FP -I../../incl -DSTORE_ORDERING -O3 -1V3 -c ../../src/Processor/
make: DUSESIGNAL: Command not found
make: [units.o] Error 127 (ignored)
make: Warning: File `act.o' has modification time 49 s in the future
gcc -DUSESIGNAL -DUNELF -DNO_IEEE_FP -I../../incl -DSTORE_ORDERING -O1 -1V3 -c ../../src/MemSy
In file included from ../../src/MemSys/util.c:64:
/usr/lib/gcc/i386-redhat-linux/3.4.6/include/varargs.h:4:2: #error "GCC no longer implements <
/usr/lib/gcc/i386-redhat-linux/3.4.6/include/varargs.h:5:2: #error "Revise your code to use <s
../../src/MemSys/util.c:177: error: syntax error before "va_dcl"
../../src/MemSys/util.c:178: error: syntax error before '{' token
../../src/MemSys/util.c:182: warning: parameter names (without types) in function declaration
../../src/MemSys/util.c:182: warning: data definition has no type or storage class
../../src/MemSys/util.c:183: error: conflicting types for 'fmt'
../../src/MemSys/util.c:180: error: previous declaration of 'fmt' was here
../../src/MemSys/util.c:183: error: `var' undeclared here (not in a function)
../../src/MemSys/util.c:183: error: syntax error before "char"
../../src/MemSys/util.c:185: warning: parameter names (without types) in function declaration
../../src/MemSys/util.c:185: warning: data definition has no type or storage class
../../src/MemSys/util.c:187: warning: parameter names (without types) in function declaration
../../src/MemSys/util.c:187: warning: data definition has no type or storage class
../../src/MemSys/util.c:188: error: syntax error before '}' token
../../src/MemSys/util.c:191: error: syntax error before "va_dcl"
../../src/MemSys/util.c:192: error: syntax error before '{' token
../../src/MemSys/util.c:194: error: conflicting types for 'fmt'
../../src/MemSys/util.c:183: error: previous definition of 'fmt' was here
../../src/MemSys/util.c:194: error: conflicting types for 'fmt'
../../src/MemSys/util.c:183: error: previous definition of 'fmt' was here
../../src/MemSys/util.c:196: warning: parameter names (without types) in function declaration
```

Слика 145: Изглед RSIM симулатора

```
int maindecode(state *proc){
    update_cycle(proc);
    graduate_cycle(proc);
    proc->intregbusy[ZEROREG] = 0;
    if (proc->exit{
        return 1;
    }
    ComputeAvail(proc);
    if (proc->in_exception == NULL{
        decode_cycle(proc);
    }
    return (0);
}
```

Слика 146: Пример кода RSIM симулатора

SIMCA - (The Simulator for Multi-threaded Computer Architecture) симулатор је развијен у Лабораторији за напредне истраживања у рачунарској техници и програмским преводиоцима на Катедри за електро и рачунарску технику Универзитета у Минесоти, САД ([35],[36] и [37]). Симулатор SIMCA је развијен са циљем да омогући процену перформанси super-threaded архитектура, као и приликом истраживања на различитим могућностима представљене архитектуре. Овај симулатор је развијен на основу SimpleScalar симулатора, а изворни код симулатора је написан у програмским језицима C и FORTRAH и може се превести у извршни програм за циљну super-threaded архитектуру. Перформансе SIMCA симулатора коришћеног без оптимизације које прави преводилац на SGI Challenge Cluster са R1000 процесорима је око 15 до 20 хиљада инструкција у секунди када се програм користи са великим бројем нити и око 15 хиљада инструкција у секунди када постоји само једна нит унутар симулатора која је активна. SIMCA ради само са процесорима ког којих се вишеречних операнда и адреса на нижу адресу смешта виши бајт податка (big-endian) и аутори су систем тестирали на SunOS 5.6 и Silicon Graphics IRIX 6.2 оперативним системима. SIMCA покушава да симулира микро-архитектуру за сваку елементарну нит обраде. Свака елементарна нит обраде сам по себи представља процесор ког кога више инструкција може започињати извршавање истовремено (superscalar processor). Главни допринос овог симулатора је да решава многа питања везана за детаље дизајна хардвера, а може да служи и као водич кроз актуелне имплементације хардверске. Овај симулатор је коришћење пројекту Superthreaded Computer Processor Design на Универзитету у Минесоти. На следећој слици је дат приказ

излаза SIMCA симулатора, као и део изворног кода симулатора.

```
int eventq_execute(EVENTQ_ID_TYPE id) {
    struct eventq_desc *prev, *ev;
    for (prev=NULL,ev=eventq_pending; ev;
        prev=ev,ev=ev->next){
        if (ev->id == id){
            if (prev){
                prev->next = ev->next;
            }
            else {
                eventq_pending = ev->next;
            }
            EXECUTE_ACTION(ev, 0);
            ev->next = eventq_free;
            eventq_free = ev;
            return TRUE;
        }
    }
    return FALSE;
}
```

Слика 147: Изглед SIMCA симулатора	Слика 148: Пример кода SIMCA симулатора

SimFlex – Симулатор је развијена на одсеку за Електротехнику и рачунарску технику на универзитету Карнеги Мелон ([38],[39] и [40]). Симулационо окружење SimFlex се састоји из два дела Flexus и SMARTS који треба да омогуће бржу и детаљнију симулацију. Flexus је богат и флексибилан симулациони оквир који омогућава симулацију целог система (full-system simulation) која се ослања на добро дефинисани модел интерфејса компоненте који омогућава да се олакша интеграција и модела и оптимизација симулатора коју може да начини програмски преводилац. SMARTS компонента симулатора примењује ригорозну теорију статистичког узорковања да би се смањила успорења симулације за неколико редова величине, у исто време омогућавајући постизање високе тачности и поверења у добијене статистичке резултате. Симулатор омогућава пројектовање засновано на коришћењу готових компонената које омогућавају пројектовање процесора код којих се инструкције извршавају и у редоследу у коме пристижу (in order) и ван тог редоследа (out-of-order) на једнопроцесорским и више процесорским моделима. Симулатор има уграђен модул за управљање статистиком и проверу исправности стања симулације. Flexus припада породици симулатора архитектуре рачунара заснованих на коришћењу компонената развијених користећи C++ које су изграђене користећи интерфејс Virtutech Simics' Micro-Architecture Interface (MAI). Овај приступ користе да би се омогућила

временски прецизна симулација рада читавог система на којима се извршава не модификовани изворни код оперативног система и комерцијалне апликације. Уколико се жели детаљна симулације предложеног хардвера, а не само статистички прецизна симулација Flexus симулира понашање процесора се перформансама које су чак 100.000 пута спорије од стварног хардвера. Уколико се користи статистички прецизна симулација која је настала након функционалног загревања система онда је симулација само 100 до 1.000 пута спорије од хардвера. SimFlex је коришћен за моделовање x86 и SPARC система заснованих на једнопроцесорским и вишепроцесорским системима. Симулатор је коришћен на курсевима архитектуре рачунара да би омогућио студентима да савладају рад са једнопроцесорским и вишепроцесорским системима и различитим редоследима извршавања инструкција, на Карнеги Мелон универзитету. На следећој слици је дат приказ излаза SimFlex симулатора, као и део изворног кода симулатора.

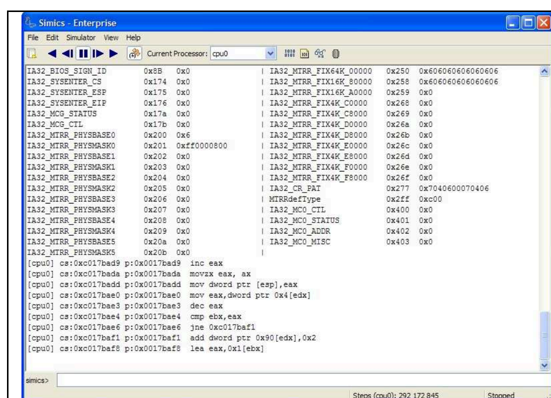
```
public class EventQueue {
    void post(Detail::EventImpl & anEvent) {
        if (anEvent.theTiming ==
            Detail::EventImpl::eSeconds) {
            API::SIM_time_post(theQueueOwner,
                anEvent.theDelta.Seconds,

static_cast<API::sync_t>(anEvent.theSync),
                anEvent.theHandler,
                anEvent.theParameter);
        } else if (anEvent.theTiming ==
...
        } else {
            API::SIM_stacked_post(

theQueueOwner,anEvent.theHandler,
                anEvent.theParameter);
        }
    }
```

Слика 149: Изглед SimFlex симулатора

Слика 150: Пример кода SimFlex симулатора

Simics – Симулатор је развијен на Virtutech AB Stockholm ([41] и [42]). Симулатор Simics представља платформу за потпуну симулацију система на којој се може извршавати стварни уграђени програм (firmware) и потпуно неизмењено језгро оперативног система као и везне програме. Потпуна симулација система подразумева стварање модела сваког дигиталног система који тачно описује границу између хардвера и софтвера (обично на нивоу регистара и прекида). Овакав приступ омогућава се бинарни код не модификован извршава у симулатору јер код не може да открије разлику од стварног хардвера. Simics симулатор је тако развијен да може да симулира сваки електронски систем, а не

само једноставан појединачне рачунаре. Скуп развијених компонената које симулатор може да симулира обухвата вишепроцесорске сервере, телекомуникационе свичеве и рутере, другу мрежну опрему, авионику, аутомобилску електронике и тако даље. Због своје брзине у Simics симулатор се могу лако додати нове компоненте и искористити старе компоненте додавањем слоја апстракције. Овај симулатор је развијен коришћењем C програмски језик, али је развијен и механизам који омогућава аутоматско превођење у Python. Симулатор може користити на следећим Оперативни систем: Linux (x86, PowerPC, и Alpha), Solaris/UltraSparc, True64/Alpha, и Windows 2000/x86. Овај симулатор је развије тако да опонаша систем на нивоу понашања, што даје га чини довољно апстрактним да оствари подношљив ниво квалитета, а да пружи и функционалну тачност приликом покретања комерцијалних апликација, али и довољну временску тачност приликом симулације појединих модела хардвер. Simics симулатор такође може омогућити симулацију система хетерогене мреже састављене од компонената различитих произвођача у истом оквиру. Поред ових могућности симулатор нуди платформу за приступ користећи развијени API као и посебно скрипт окружење за коришћење у широком опсегу примена. Овај програм могу користити дизајнери, студенти, тестери, аналитичари перформанси, они који се баве реверзибилним извршење програма као и отклањањем грешака. У настави се овај симулатор користио на курсу Рачунарских система на Упсала универзитету. На следећој слици је дат приказ излаза Simics симулатора, као и део изворног кода симулатора.



Слика 151: Изглед Simcs симулатора

```
/*
 * the init_local() function is called after
loading this file with load_object()
 */
void init_local() {
  /* the third parameter is for new
processors, but this machine is unipro */
  set_mmu(mmu_access, mmu_access_inq, NULL);
  device_define(tty_init, tty_operation,
                tty_describe, "tty");
  memory_define(fix_address,outside_memory);
  printf("Loaded machine specification.\n");
}
```

Слика 152: Пример кода Simcs симулатора

SimOS - Симулатор је развијен на универзитету Стенфорд као комплетно симулационо окружење предвиђено за извођење ефикасних и тачних студија једнопроцесорских и вишепроцесорских рачунарских система ([43] и [44]). Симулатор SimOS омогућава симулацију рачунарског хардвер (процесор, кеш меморија, вишепроцесорске магистрале, дискови, ...) са довољним нивоом детаља да се у оквиру симулатора може покренути комерцијално доступни оперативни систем. SimOS као развијене компоненте нуди више различитих модела процесора што укључује MIPS R4000 и R10000, као и Digital Alpha фамилије процесор, као и моделе хардвера сличног машинама које производе Silicon Graphics Inc. и Digital Equipment Corporation. SimOS има могућност извршавања у два режима рада: непосредно извршење и детаљна симулација. Код непосредног извршавања комплетан програм се извршава на процесору у оквиру неког процеса у корисничком адресном простору. Овом приликом је могуће пратити извршавање инструкција без залажења у детаље. Код детаљне симулације цео систем као и сет инструкција се симулира. На овај начин је могуће визуелно представити извршавање програма као и сачувати комплетно стање симулиране машине на диску ради каснијег настављања симулације и лакше накнадне анализе. MIPS верзија је развијена користећи C програмски језик, и донекле је преносива и може се користити под Irix, Solaris и Linux оперативним системима. SimOS симулатор представља окружење за учење хардвера и софтвера рачунарских система. Приликом симулација SimOS балансира између брзине обављања симулације и детаља које та симулација пружа. Прелазак између ових режима рада могуће је и током саме симулације, све у циљу смањења времена потребног за обављање комплетне симулације, а да се при томе не жртвују детаљи у траженом временском оквиру. Овај програм могу да користи студенти, особе које се баве развојем хардвера, тестери, и аналитичари перформанси, за обављање студија дизајна новог рачунарског хардвера, за анализу перформанси апликација, као и за проучавање рада оперативних система. Симулатор је коришћен на многим пројектима, укључујући испитивање нових решења у области архитектуре рачунара, приликом развоја Hive оперативног система, као и за разне студије перформанси оперативних система и апликација. На следећој слици је дат приказ излаза SimOS симулатора приликом његове инсталације, као и део изворног кода

симулатора.

```
root@rti30:/storage/exp_soft/simos
#!/bin/csh -f

set  SCRIPTDIR=$SIMOS_DIR/bin  set BINDIR=$SIMOS_DIR/bin/$CPU set
BUILDFILE=root-cd.bld set DISKFILE=root.disk

# Extra space to put on the root disk (in 512  byte  blocks)  set
FREESPACE=40000

# Size of swap partition (in Megabytes) set SWAPSIZE=20

echo "Creating mkfs input from ${BUILDFILE}" $SCRIPTDIR/builddisk
${BUILDFILE}  >  root-cd.mkfs1  $SCRIPTDIR/adjsize  root-cd.mkfs1
root-cd.mkfs2 ${FREESPACE} rm root-cd.mkfs1 rm -f ${DISKFILE}

# Remove any occurrences of /dev/null from file sed â/devnull/ dâ
root-cd.mkfs2 > root-cd.mkfs

echo "Running mkfs to create ${DISKFILE}" if ( { $BINDIR/mkfs  -q
$DISKFILE root-cd.mkfs } != 1) then
  echo "MKFS failed... aborting"
  exit 1; endif

echo "Fixing superblock of ${DISKFILE}" $BINDIR/vh -r ${SWAPSIZE}
:
```

Слика 153: Изглед SimOS симулатора

```
INLINE_EVENTS (void)
event_queue_process(event_queue *events){
    if (events->time_from_event == 0) {
        do {
            event_entry *to_do = events->queue;
            events->queue = to_do->next;
            to_do->handler(events,to_do->data);
            zfree(to_do);
        } while (events->queue != NULL
             && events->queue->time_of_event
                <= events->time_of_event);
        if (events->queue != NULL) {
            events->time_from_event =
                 (events->queue->time_of_event
                 - events->time_of_event);
            events->time_of_event =
                events->queue->time_of_event;
        }
        else {
        }
    }
}
```

Слика 154: Пример кода SimOS симулатора

SimpleScalar - Симулатор је развијен на Департману за рачунарске науке, Универзитета Wisconsin-Madison у САД ([45],[46] и [47]). Симулатор SimpleScalar представља окружење које се састоје од програмског преводиоца, линкер, симулатора и алата за визуелизацију. Приликом рада са овим симулатором на располагању стоје и скуп уграђених компонената које је могуће модификовати користећи доступне алате. Уграђене компоненте обухватају процесоре са статичким и динамичким извршавањем, са спекулативним извршавањем инструкција, као и најсавременије процесоре у области архитектуре и организације рачунара, као и неблокирајуће кеш меморије. Овај симулатор омогућава кориснику да обави модификације компонената, што укључујуће и промену платформе као и могућност проширења. Овај симулатор приликом покретања симулације узима бинарне датотеке програма преведених за SimpleScalar архитектуру и симулира њиховог извршења на једној од неколико варијанти процесора које стоје на располагању. Ради тестирања перформанси циљних система симулатор нуди већ припремљене и преведене тест програме (укључујући SPEC), као и могућност да корисник преводи своје програме писане користећи FORTRAN и C помоћу модификоване верзија GNU GCC преводиоца. Овај симулатор је развијен коришћењем C програмски језик и предвиђен је за коришћење под UNIX оперативним системом, али се може пребацити за било коју варијанту која подржава POSIX, па чак и за Windows оперативни систем уколико

се користе cygwin32 GNU алати. Овај симулатор је првенствено намењен за брзо обављање симулација на функционалном нивоу али је могуће на њему извршавати и детаљне симулације у зависности од брзине која се жели постићи. Као уграђене компоненте постоје функционални описи за процесоре са Alpha, PISA, ARM, и x86 сетовима инструкција. Симулатор SimpleScalar се може користити за изградњу симулација за анализу перформанси програм, анализу перформанси код моделовања микро архитектуре и у случају да је потребно урадити хардверско-софтверску коверификацију. Симулатор SimpleScalar је коришћен на извесном броју пројеката који обухватају MASE пројекта на Универзитету Мичиген, PowerAnalyzer пројекта на Универзитету Мичиген, SIMCA пројекту на универзитету у Минесоти, Wattch пројекат на Универзитету Принстон, HydraScalar пројекат на универзитету Вирџиније. На следећој слици је дат приказ излаза SimpleScalar симулатора приликом његове инсталације, као и део изворног кода симулатора.

```
root@rti30:/storage/exp_soft/SimpleScalar/simplesim-3.0
gcc `./sysprobe -flags` -DDEBUG -O0 -g -Wall   -c sim-safe.c
gcc -o sim-safe `./sysprobe -flags` -DDEBUG -O0 -g -Wall   sim-safe.o main.o syscall.o memory.o regs.o loader.o endian.o
 dlite.o symbol.o eval.o options.o stats.o eio.o range.o misc.o machine.o libexo/libexo.a `./sysprobe -libs` -lm
gcc `./sysprobe -flags` -DDEBUG -O0 -g -Wall   -c sim-eio.c
gcc -o sim-eio `./sysprobe -flags` -DDEBUG -O0 -g -Wall   sim-eio.o main.o syscall.o memory.o regs.o loader.o endian.o d
lite.o symbol.o eval.o options.o stats.o eio.o range.o misc.o machine.o libexo/libexo.a `./sysprobe -libs` -lm
gcc `./sysprobe -flags` -DDEBUG -O0 -g -Wall   -c sim-bpred.c
gcc `./sysprobe -flags` -DDEBUG -O0 -g -Wall   -c bpred.c
gcc -o sim-bpred `./sysprobe -flags` -DDEBUG -O0 -g -Wall   sim-bpred.o bpred.o main.o syscall.o memory.o regs.o loader.
o endian.o dlite.o symbol.o eval.o options.o stats.o eio.o range.o misc.o machine.o libexo/libexo.a `./sysprobe -libs` -
lm
gcc `./sysprobe -flags` -DDEBUG -O0 -g -Wall   -c sim-profile.c
gcc -o sim-profile `./sysprobe -flags` -DDEBUG -O0 -g -Wall   sim-profile.o main.o syscall.o memory.o regs.o loader.o en
dian.o dlite.o symbol.o eval.o options.o stats.o eio.o range.o misc.o machine.o libexo/libexo.a `./sysprobe -libs` -lm
gcc `./sysprobe -flags` -DDEBUG -O0 -g -Wall   -c sim-cache.c
gcc `./sysprobe -flags` -DDEBUG -O0 -g -Wall   -c cache.c
gcc -o sim-cache `./sysprobe -flags` -DDEBUG -O0 -g -Wall   sim-cache.o cache.o main.o syscall.o memory.o regs.o loader.
o endian.o dlite.o symbol.o eval.o options.o stats.o eio.o range.o misc.o machine.o libexo/libexo.a `./sysprobe -libs` -
lm
gcc `./sysprobe -flags` -DDEBUG -O0 -g -Wall   -c sim-outorder.c
gcc `./sysprobe -flags` -DDEBUG -O0 -g -Wall   -c resource.c
gcc `./sysprobe -flags` -DDEBUG -O0 -g -Wall   -c ptrace.c
gcc -o sim-outorder `./sysprobe -flags` -DDEBUG -O0 -g -Wall   sim-outorder.o cache.o bpred.o resource.o ptrace.o main.o
 syscall.o memory.o regs.o loader.o endian.o dlite.o symbol.o eval.o options.o stats.o eio.o range.o misc.o machine.o li
bexo/libexo.a `./sysprobe -libs` -lm
my work is done here...
[root@rti30 simplesim-3.0]#
```

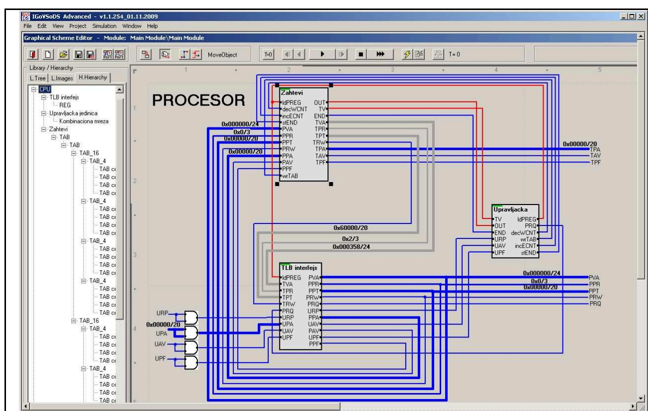
Слика 155: Изглед SimpleScalar симулатора

```
void eventq_service_events(SS_TIME_TYPE
now){
    while (eventq_pending
         && eventq_pending->when <= now){
         struct eventq_desc *ev =
eventq_pending;
         /* handle action */
         EXECUTE_ACTION(ev, now);
 /* return the event record to the free list
*/
         eventq_pending = ev->next;
         ev->next = eventq_free;
         eventq_free = ev;
    }
}
```

Слика 156: Пример кода SimpleScalar симулатора

VSDS - Овај симулатор је развијен на Електротехничком факултету Универзитета у Београду ([48],[49] и [50]). Овај симулатор омогућава корисницима креирање нових, сложенијих хијерархијских модула који се могу користи као градивни блокови за стварање нових симулатора. Овај симулатор поседује визуелни алат за креирање сложених дигиталних система повезивањем расположивих модула и као и могућност симулације добијени система на нивоу трансфера између регистара. Скуп пратећих алата омогућава креирање нових, корисничких, модула који се могу користи као грађевински блокови за стварање нових симулатора. Ови модули се могу додатно конфигурисати, што укључује и коришћење именованих и неименованих шаблона који су идеални градивни блокови за дизајн

прототипова и анализу проширивости архитектура. Захваљујући алатима за визуелизацију временских облика сигнала овај симулатор се може користити за симулациону анализу створених модула. Симулатор је развијен користећи Visual BASIC програмски језик и не модификован се може извршавати на Windows оперативном систему. У оквиру овог симулатора постоје доступни алати који олакшавају праћење и отклањање грешака. Алат је посебно прилагођен брзом развоју прототипова организација рачунара. Симулатор је првенствено намењен студентима, али га могу користити особе које се баве развојем и тестирање понашања дигиталних компонената. VSDS је коришћен за развој едукативних симулатора кеш меморије са више начина пресликавања, виртуелне меморије различитих организација, меморије са преклопљеним приступом, процесора CISC архитектуре, процесора са проточном обрадом RISC архитектуре, као и симулациону верификацију збирки задатака из наведене области. Симулатор се користи на Универзитету у Београду на курсевима Основа рачунарске технике, и Архитектуре и организације рачунара 1. На следећој слици је дат приказ једног прозора VSDS са реализованим процесором и кеш меморијом, као и део изворног кода симулатора.



Слика 157: Изглед VSDS симулатора

```
Public Function Tf_FunDetails_Delete(
        argID As Long) As Boolean
    gSQLxFalseExitCode = 0
    If Tf_FunDetails_cur<=Tf_FunDetails_max Then
        If Tf_FunDetails(argID).VEntry=True Then
            Tf_FunDetails(argID).VEntry = False
            Tf_FunDetails_rm_cur =
                Tf_FunDetails_rm_cur + 1
            Tf_FunDetails_rm(
                Tf_FunDetails_rm_cur) = argID
            Tf_FunDetails_Delete = True
        Else
            Tf_FunDetails_Delete = False:
            gSQLxFalseExitCode = 11001
        End If
    Else
        Tf_FunDetails_Delete = False:
        gSQLxFalseExitCode = 21001
    End If
End Function
```

Слика 158: Пример кода VSDS симулатора

Прилог В: Преглед наставе из конкурентног и дистрибуираног програмирања на првих 100 универзитета Шангајске листе

Ранг	Универзитет	Име курса	Часова предавања	Часова вежби	Часова лабораторије	Укупно часова	Укупно часова	Укупно лабораторија	Предавања за КДП	Вежбе за КДП	Лабораторија за КДП	Укупно часова за КДП	Процеси и нити	Катанци и баријере	Синхронизација	Алгоритми	Семафори	Монитори	Условни региони	Синхронизација	Алгоритми	Размена порука	Удаљени позиви процедура	Мрежно програмирање	Rendezvous	Grid Computing	Веб сервиси	P2P	Програмски језик	Пројекат	Пројекат	Лабораторијске вежбе	Квизови	Учешће у настави	Колоквијум	Испит	Литература
1	Harvard University	Principles of Operating Systems	1	0	0	15	0	0	4	0	0	4	+	+	+		+	+		+	+	+	+	+					C	23	65	0	0	0	12	0	Modern Operating Systems, Third Edition; Andrew S. Tanenbaum; Prentice-Hall, 2008
2	Stanford University	Distributed Systems	2	0	0	16	0	0	8	0	0	8								+	+	+	+	+					C/C++	0	40	0	0	0	15	45	Distributed Systems: Concepts and Design by George Coulouris, Jean Dollimore, and Tim Kindberg, Addison Wesley, August 2000 (3rd edition)
		Operating Systems	3	0	0	28	0	0	9	0	0	9	+	+	+	+	+					+	+						C	50	0	0	0	0	15	35	Operating System Concepts, 8th Edition, by Silberschatz, Galvin, and Gagne
3	University of California, Berkeley	Concurrent Models of Computation	3	0	0	48	0	0	18	0	0	18	+		+	+				+	+	+							Java								B. A. Davey and H. A. Priestley, *Introduction to Lattices and Order,* Cambridge University Press, second edition, 2002.
		Operating Systems and Systems Programming	4	0	0	52	0	0	18	0	0	18	+	+	+	+	+	+	+				+	+					Java	50	0	0	0	5	20	25	Operating System Concepts, 8th Edition, by Silberschatz, Galvin, and Gagne
5	MIT	Distributed Algorithms	4	0	0	52	0	0	24	0	0	24		+			+			+	+	+	+	+						0	70	0	0	20	0	0	Lynch, Nancy A. Distributed Algorithms. San Francisco, CA: Morgan Kaufmann, 1997. ISBN: 1558603484.
		Operating System Engineering	4	0	1	46	0	6	6	0	2	8	+	+	+							+	+						C	0	20	50	30	0	0	0	Lions, John. *Lions' Commentary on UNIX® 6th Edition with Source Code*. San Jose, CA: Peer-to-Peer Communications, 1996. ISBN: 1573980137.
		Distributed Computer Systems Engineering	4	0	1	48	0	10	14	0	6	20								+	+	+	+	+					C	40	0	25	25	10	0	0	Tanenbaum, Andrew. *Modern Operating Systems*. 2nd ed. Upper Saddle River, NJ: Prentice Hall, 2001. ISBN: 9780130313584.

Ранг	Универзитет	Име курса	Часова предавања	Часова вежби	Часова лабораторије	Укупно часова	Укупно часова	Укупно лабораторија	Предавања за КДП	Вежбе за КДП	Лабораторија за КДП	Укупно часова за КДП	Процеси и нити	Катанци и баријере	Синхронизација	Алгоритми	Семафори	Монитори	Условни региони	Синхронизација	Алгоритми	Размена порука	Удаљени позиви процедура	Мрежно програмирање	Rendezvous	Grid Computing	Веб сервиси	P2P	Програмски језик	Пројекат	Пројекат	Лабораторијске вежбе	Квизови	Учешће у настави	Колоквијум	Испит	Литература
7	Columbia University	Operating Systems I	4	0	0	52	0	0	14	0	0	14	+		+	+													C	0	50	0	0	0	20	30	Operating System Concepts (8th Edition), Avi Silberschatz, Peter Galvin, and Greg Gagne, John Wiley and Sons, New York, NY, 2009
8	Princeton University	Operating Systems	4	0	0	48	0	0	20	0	0	20	+	+	+	+	+	+	+			+	+						C	60	0	0	15	10	15	0	Protected Mode Software Architecture, by Tom Shanley, MindShare, Inc. 1996.
9	University of Chicago	Networks and Distributed Systems	2	0	0	36	0	0	14	0	0	14	+				+		+	+	+		+				+	+	C	40	20	0	0	0	20	20	Computer Networks: A Systems Approach, 4th edition L. Peterson and B. Davie
10	University of Oxford	Concurrent Programming										16	+		+	+	+	+		+	+	+							CSP, Java							50	Foundations of Multithreaded, Parallel, and Distributed Programming, Gregory R. Andrews, Addison-Wesley, 2000.Principle of Concurrent and Distributed Programming, M. Ben-Ari, Prentice Hall, 1990.
11	Yale University	Operating Systems	2	0	0	25	0	6	6	0	2	8	+	+	+	+	+			+	+	+	+						C	0	0	60	0	20	20	0	Operating System Concepts, 8th Edition, by Silberschatz, Galvin, and Gagne
		Theory of Distributed Computing	3	0	0	39	0	0	39	0	0	39					+			+	+	+	+	+	+			+									Hagit Attiya and Jennifer Welch, Distributed computing : fundamentals, simulations, and advanced topics, second edition. Wiley, 2004.
12	Cornell University	Systems Programming and Operating Systems	4	0	0	48	0	0	32	0	0	32	+	+	+	+	+	+			+		+	+					C	0	10	0	0	0	30	50	Operating System Concepts, 8th Edition, by Silberschatz, Galvin, and Gagne
13	University of California, Los Angeles	Operating Systems Principles	2	1	0	32	0	4	14	0	1	15	+		+	+				+	+		+	+				+		0	6	54	0	0	16	24	Jerry Saltzer and Frans Kaashoek, Principles of Computer System Design, MIT
14	University of California, San Diego	Principles of Computer Operating Systems	4	0	0	36	0	0	12	0	0	12	+	+	+	+	+	+											Java	30	16	0	0	0	24	30	Operating System Concepts, 8th Edition, by Silberschatz, Galvin, and Gagne
		Operating Systems	4	0	0	38	0	0	4	0	0	4	+																Java	40	0	0	0	30	0	30	There is no required text for this course.
15	University of Pennsylvania	Software Systems	3	0	0	40	0	0	24	0	0	24	+	+	+	+	+	+		+	+	+	+	+					Java	50	0	0	0	0	20	30	Distributed Systems: Principles and Paradigms(2nd Edition). Andrew S. Tanenbaum and Maarten Van Steen, Prentice Hall, 2007.

Ранг	Универзитет	Име курса	Часова предавања	Часова вежби	Часова лабораторије	Укупно часова	Укупно часова	Укупно лабораторија	Предавања за КДП	Вежбе за КДП	Лабораторија за КДП	Укупно часова за КДП	Процеси и нити	Катанци и баријере	Синхронизација	Алгоритми	Семафори	Монитори	Условни региони	Синхронизација	Алгоритми	Размена порука	Удаљени позиви процедура	Мрежно програмирање	Rendezvous	Grid Computing	Веб сервиси	P2P	Програмски језик	Пројекат	Пројекат	Лабораторијске вежбе	Квизови	Учешће у настави	Колоквијум	Испит	Литература
		Operating Systems	4	0	0	48	0	0	22	0	0	22	+	+	+	+	+			+	+		+						Java	35	0	0	0	40	0	25	A.S. Tanenbaum. Modern Operating Systems (3/e). Prentice Hall.
16	University of Washington	Operating Systems	3	0	0	28	0	0	15	0	0	15	+	+	+	+	+	+											C/C++	35	10	0	0	5	20	30	Operating System Concepts, 8th Edition, by Silberschatz, Galvin, and Gagne
17	University of Madison - Wisconsin	Introduction to Operating Systems	2	0	0	32	0	0	7	0	0	7	+	+	+	+	+												C	50	0	0	0	0	25	25	There is no required text for this course.
18	University of California, San Francisco	Operating Systems	2	0	1	23	0	11	4	0	0	4	+	+	+	+	+												C, C++, Java	30	0	0	0	20	30	20	William Stallings, Operating Systems: Internals and Design Principles (5th Edition), Prentice-Hall/Pearson Education, 2005
19	The Johns Jopkins University	Distributed Systems	2	0	0	24	0	0	6	0	0	6								+	+			+				+	C/C++	30	60	0	0	10	0	0	Reliable Distributed Systems, Kenneth P. Birman, Springer Verlag 2005
21	University College London	Concurrent Programming	3	0	0	30	0	0	30	0	0	30	+		+	+	+	+			+	+	+	+	+				Java	0	15	0	0	0	0	85	Concurrency: State Models and Java Programs by J. Magee and J. Kramer; Hardcover - 355 pages.
22	University of Michigan - Ann Arbor	Introduction to Operating Systems	2	0	0	26	0	0	10	0	0	10	+	+	+	+	+	+		+	+		+	+					C/C++	45	0	0	0	0	25	30	A.S. Tanenbaum. Modern Operating Systems (3/e). Prentice Hall.
23	Swiss Federal Institute of Technology, Zurich	Concepts of Concurrent Computation	2	1	0	26	9	0	24	9	0	33	+	+	+	+	+	+											C++, Java	50	0	0	0	0	0	50	Bertrand Meyer, Sebastian Nanz: Concepts of Concurrent Computation
24	Kyoto University	Operating System	2	0	0	26	0	0	8	0	0	8	+	+	+	+	+													0	0	0	0	0	0	100	Nagai Masatake, Tsutomu Sawada Ayako: Linux and Windows OS
25	University of Illinois at Urbana-Champaign	Distributed Systems	2	0	0	23	0	0	8	0	0	8			+		+			+	+			+				+		35	20	0	0	5	10	30	Couloris, G., Dollimore, J., Kindberg, T. Distributed Systems: Concepts and Design, Addison-Wesley, Fourth Edition
26	The Imperial College of Science, Technology and Medicine	Operating Systems	2	1	0	22	11	0	12	2	0	14	+	+	+	+	+					+															A.S. Tanenbaum. Modern Operating Systems (3/e). Prentice Hall.
27	University of Toronto	Operating Systems	2	0	0	26	0	0	10	0	0	10	+	+	+	+	+	+											C	0	45	0	0	0	15	40	Operating System Concepts, 8th Edition, by Silberschatz, Galvin, and Gagne
28	University of Minnesota, Twin Cities	Introduction to Operating Systems	2	0	0	28	0	0	8	0	0	8	+	+	+		+							+					C, C++, Java	0	50	0	0	0	30	20	Unix Systems Programming: Communication, Concurrency, and Threads, Robbins and

Ранг	Универзитет	Име курса	Часова предавања	Часова вежби	Часова лабораторије	Укупно часова	Укупно часова	Укупно лабораторија	Предавања за КДП	Вежбе за КДП	Лабораторија за КДП	Укупно часова за КДП	Процеси и нити	Катанци и баријере	Синхронизација	Алгоритми	Семафори	Монитори	Условни региони	Синхронизација	Алгоритми	Размена порука	Удаљени позиви процедура	Мрежно програмирање	Rendezvous	Grid Computing	Веб сервиси	P2P	Програмски језик	Пројекат	Пројекат	Лабораторијске вежбе	Квизови	Учешће у настави	Колоквијум	Испит	Литература
																																					Robbins (R&R), Prentice-Hall, 2003
		Foundations of Modern Operating Systems	2	0	0	30	0	0	30	0	0	30								+	+	+	+	+	+	+			C, C++, Java	45	15	0	0	0	15	25	Distributed Systems: Principles and Paradigms(2nd Edition). Andrew S. Tanenbaum and Maarten Van Steen, Prentice Hall, 2007.
29	Washington University in St. Louis	Operating Systems Organization	4	0	0	58	0	0	18	0	0	18	+	+	+	+	+												C/C++	40	20	0	0	0	0	40	A.S. Tanenbaum. Modern Operating Systems (3/e). Prentice Hall.
30	Northwestern University	Operating Systems	4	0	0	34	0	0	14	0	0	14	+	+	+	+	+	+		+						+	+	+	C	40	20	0	0	0	20	20	Operating System Concepts, 8th Edition, by Silberschatz, Galvin, and Gagne
		Distributed Systems	4	0	0	34	0	0	34	0	0	34								+	+	+	+	+	+	+			C	25	45	0	0	10	0	20	Distributed Systems: Principles and Paradigms(2nd Edition). Andrew S. Tanenbaum and Maarten Van Steen, Prentice Hall, 2007.
		Distributed Computing Systems	2	0	0	20	0	0	20	0	0	20								+	+	+	+	+	+	+	+	+	C	35	25	0	0	0	0	40	Distributed Systems: Principles and Paradigms(2nd Edition). Andrew S. Tanenbaum and Maarten Van Steen, Prentice Hall, 2007.
31	Duke University	Computer Networks and Distributed Systems	2	0	0	23	0	2	6	0	0	6												+	+	+	+	+	C++, Java	50	10	10	25	5	0	0	Larry L. Peterson, Bruce S. Davie, Computer Networks: A Systems Approach, Morgan Kaufmann.
		Operating Systems	3	0	0	21	0	0	9	0	0	9	+	+	+	+	+	+											C/C++	35	0	0	0	5	30	30	There is no required text for this course.
32	New York University	Operating Systems	2	0	0	27	0	4	8	0	1	9	+	+	+	+	+												C, C++, Java	0	0	40	0	0	0	60	A.S. Tanenbaum. Modern Operating Systems (3/e). Prentice Hall.
34	University of Colorado at Boulder	Operating Systems	2	0	0	28						0																		15	45	0	0	0	15	25	Operating Systems, Third Edition, Gary Nutt, Addison Wesley, 2004.
		Distributed Systems	2	0	0	28						0								+	+	+	+	+		+	+			45	45	0	0	0	0	10	There is no required text for this course.
36	University of British Columbia	Operating and File Systems	3	0	0	39						0	+	+	+	+	+	+											C, C++, Java								Operating System Concepts with Java 6th Edition by A.Silberschatz, P.B.Galvin, and G.Gagne
37	University of Maryland, College Park	Operating Systems	2	0	0	25	0	0	8	0	0	8	+	+	+	+	+												C	0	40	0	0	0	30	30	Operating System Concepts 6th Edition (or newer) by

Ранг	Универзитет	Име курса	Часова предавања	Часова вежби	Часова лабораторије	Укупно часова	Укупно часова	Укупно лабораторија	Предавања за КДП	Вежбе за КДП	Лабораторија за КДП	Укупно часова за КДП	Процеси и нити	Катанци и баријере	Синхронизација	Алгоритми	Семафори	Монитори	Условни региони	Синхронизација	Алгоритми	Размена порука	Удаљени позиви процедура	Мрежно програмирање	Rendezvous	Grid Computing	Веб сервиси	P2P	Програмски језик	Пројекат	Пројекат	Лабораторијске вежбе	Квизови	Учешће у настави	Колоквијум	Испит	Литература
																																					A.Silberschatz, P.B.Galvin, and G.Gagne
38	The University of Texas, Austin	Introduction to Operating Systems	2	0	0	30	0	0	13	0	0	13	+	+	+	+	+	+		+	+		+	+					C	40	0	0	0	0	60	0	Operating System Concepts, 8th Edition, by Silberschatz, Galvin, and Gagne
39	University of North Carolina at Chapel Hill	Distributed and Concurrent Algorithms	3	0	0	42	0	0	42	0	0	42	+	+	+	+	+	+		+	+	+	+	+					C	30	65	0	0	5	0	0	Distributed Systems, second edition, Addison-Wesley, 1993
41	The University of Manchester	Concurrency and Process Algebra	2	0	0	22	0	0	22	0	0	22	+	+	+	+	+	+			+	+							Java	0	0	0	0	0	0	100	Jeff Magee and Jeff Kramer, Concurrency: State Models & Java Programming, 2nd edition, Wiley, 2006
45	Pennsylvania State University - University Park	Operating Systems	2	0	0	28	0	0	11	0	0	11	+	+	+	+	+	+											C	30	15	0	0	0	30	25	Operating System Concepts, 8th Edition, by Silberschatz, Galvin, and Gagne
46	University of California, Irvine	Principles of Operating Systems	2	0	0	20	0	0	10	0	0	10	+	+	+	+	+												C	0	0	0	42	0	0	52	Operating Systems Principles, L. F. Bic and A. C. Shaw, Prentice Hall, 2003
47	University of Southern California	Operating Systems	4	0	0	40	0	0	24	0	0	24	+	+	+	+	+	+		+	+		+						C/C++	40	0	0	0	0	30	30	Operating System: A Concept Based Approach , D. J. Dhamdere, MCGRAW-HILL COLLEGE
49	University of California, Davis	Operating Systems and Systems Programming	3	0	0	28	0	0	9	0	0	9	+	+	+	+	+	+											C	0	50	0	0	2	16	32	The Design and Implementation of the FreeBSD Operating Systems, Marshall Kirk McKusick and George V. Neville-Neil Addison Wesley Professional, 2005
		Principles of Concurrent Programming	3	0	3	28	0	28	28	0	3	31	+	+	+	+	+	+	+	+	+	+	+	+	+				Java	0	0	70	0	0	0	30	G.R. Andrews, Concurrent Programming: Principles and Practice, Benjamin/Cummings, 1991.
51	University of Pittsburgh	Introduction to Operating Systems	1	2	0	12	24	0	6	12	0	18	+	+	+	+													C	30	0	0	10	10	25	25	A.S. Tanenbaum. Modern Operating Systems (3/e). Prentice Hall.
55	Rutgers, The State University of New Jersey- New Brunswick	Operating System Design	4	0	0	42	0	0	0	20	0	20	+	+	+	+	+	+		+	+	+	+						C	0	45	0	0	0	20	35	Operating System Concepts, 8th Edition, by Silberschatz, Galvin, and Gagne
		Distributed Systems	3	0	0	42	0	0	0	0	0	0																		30	50	0	0	20	0	20	There is no required text for this course.
		Distributed Systems: Concept and Design	3	0	0	42	0	0	0	42	0	42								+	+	+	+	+					C, C++, Java								There is no required text for this course.

Ранг	Универзитет	Име курса	Часова предавања	Часова вежби	Часова лабораторије	Укупно часова	Укупно часова	Укупно лабораторија	Предавања за КДП	Вежбе за КДП	Лабораторија за КДП	Укупно часова за КДП	Процеси и нити	Катанци и баријере	Синхронизација	Алгоритми	Семафори	Монитори	Условни региони	Синхронизација	Алгоритми	Размена порука	Удаљени позиви процедура	Мрежно програмирање	Rendezvous	Grid Computing	Веб сервиси	P2P	Програмски језик	Пројекат	Пројекат	Лабораторијске вежбе	Квизови	Учешће у настави	Колоквијум	Испит	Литература
58	University of Florida	Operating Systems Principles	3	0	0	42	0	3	21	0	3	24	+	+	+	+	+	+											C	20	25	15	0	0	20	20	Operating System Concepts, 8th Edition, by Silberschatz, Galvin, and Gagne
59	Carnegie Mellon University	Distributed Systems	2	0	0	30	0	0	30	0	0	30								+	+	+	+	+	+			+	C, C++, Java	50	15	0	0	0	15	20	There is no required text for this course.
		Operating System Design and Implementation	3	0	0	42	0	0	12	0	0	12	+	+	+	+	+	+		+	+	+	+						C	55	10	0	0	0	15	20	Operating System Concepts, 8th Edition, by Silberschatz, Galvin, and Gagne
60	The Australian National University	Operating Systems Implementation	3	2	0	30	12	0	15	4	0	19	+	+	+	+	+			+	+	+	+	+					C	0	20	10	0	0	0	70	Operating Systems, William Stallings, Prentice-Hall, sixth edition, 2008
		Concurrent and Distributed Systems	3	0	0	33	0	5	0	33	5	38	+	+	+	+	+	+	+	+	+	+							Java	0	30	0	0	0	0	70	Principles of Concurrent and Distributed Programming by M. Ben-Ari, 2nd Edition 2006, Prentice-Hall
61	University of Bristol	Concurrency and Communications	3	0	0	36	0	0	0	18	0	18	+	+	+	+	+												CSP	0	50	0	0	0	0	50	Concurrent and Real-time Systems: The CSP approach, Steve Schneider; John Wiley & Sons, 2000
62	The Ohio State University - Columbus	Introduction to Operating Systems	3	0	0	30	0	4	0	10	2	12	+	+	+	+	+													0	10	25	0	0	25	40	Operating System Concepts, 8th Edition, by Silberschatz, Galvin, and Gagne
		Operating Systems	3	0	0	30	0	0	0	21	0	21	+	+	+	+	+			+	+	+								0	20	0	0	0	40	40	Distributed Systems: Principles and Paradigms(2nd Edition). Andrew S. Tanenbaum and Maarten Van Steen, Prentice Hall, 2007.
		Introduction to Distributed Computing	3	0	0	30	0	0	0	30	0	30								+	+	+	+	+	+					0	20	0	10	0	30	40	Elements of Distributed Computing - Vijay Garg
64	The Hebrew University of Jerusalem	Operating Systems	2	4	0	28	46	5	14	20	2	36	+	+	+	+	+			+	+	+	+	+					C/C++	0	40	0	0	0	0	60	Operating System Concepts, 8th Edition, by Silberschatz, Galvin, and Gagne; Operating Systems, William Stallings, Prentice-Hall, sixth edition, 2008
66	McGill University	Distributed Systems	3	0	0	42	0	0	42	0	0	42								+	+	+	+	+	+				Java	34	21	0	0	5	10	30	There is no required text for this course.
67	Purdue University - West Lafayette	Operating Systems	3	0	0	36	0	5	12	0	1	13	+	+	+	+	+												C	45	0	0	3	0	25	30	Operating System Concepts, 8th Edition, by Silberschatz, Galvin, and Gagne; Operating Systems, William Stallings, Prentice-Hall, sixth edition, 2008

Ранг	Универзитет	Име курса	Часова предавања	Часова вежби	Часова лабораторије	Укупно часова	Укупно часова	Укупно лабораторија	Предавања за КДП	Вежбе за КДП	Лабораторија за КДП	Укупно часова за КДП	Процеси и нити	Катанци и баријере	Синхронизација	Алгоритми	Семафори	Монитори	Условни региони	Синхронизација	Алгоритми	Размена порука	Удаљени позиви процедура	Мрежно програмирање	Rendezvous	Grid Computing	Веб сервиси	P2P	Програмски језик	Пројекат	Пројекат	Лабораторијске вежбе	Квизови	Учешће у настави	Колоквијум	Испит	Литература
		Distributed Systems	3	0	0	45	0	0	45	0	0	45								+	+	+	+	+					C/C++	20	33	0	0	17	10	20	There is no required text for this course.
68	University of Oslo	Operating Systems	4	4	0	48	48					0	+	+	+															100	0	0	0	0	0	0	There is no required text for this course.
69	Brown University	Operating Systems	3	0	0	34	0	0	12	0	0	12	+	+						+	+	+	+						C	40	20	0	0	0	10	30	There is no required text for this course.
		Distributed Computer Systems	2	0	0	42	0	0	42	0	0	42								+	+		+	+	+		+	+	C	45	20	0	0	0	10	25	Distributed Systems: Concepts and Design, 4th Edition, by G. Coulouris, J. Dollimore, and T. Kindberg, Addison Wesley, 2005
72	Leiden University	Operating Systems	2	0	0	26	0	6	0	0	0	0																	C	0	0	50	0	0	0	50	A.S. Tanenbaum. Modern Operating Systems (3/e). Prentice Hall.
		Theory of Concurrency	3	0	0	30	0	0	6	0	0	6								+	+																There is no required text for this course.
74	Boston University	Operating Systems	4	0	0	42	0	0	20	0	0	20	+	+	+	+	+	+		+	+								C/C++	0	50	0	0	5	20	25	Operating System Concepts, 8th Edition, by Silberschatz, Galvin, and Gagne; Operating Systems, William Stallings, Prentice-Hall, sixth edition, 2008
76	Uppsala University	Operating Systems	2	0	0	16	0	3	6	0	1	7	+	+	+	+	+	+	+										C, Java	0	0	50	0	0	0	50	Operating System Concepts, 8th Edition, by Silberschatz, Galvin, and Gagne; Operating Systems, William Stallings, Prentice-Hall, sixth edition, 2008
		Distributed Systems	1	0	0	9	0	0	9	0	0	9								+	+	+	+	+	+	+	+	+	Java	0	0	0	0	0	0	100	Distributed Systems Concepts and Design by Coulouris, Dollimore and Kindberg
73	University of Helsinky	Distributed Systems	4	0	0	26	0	0	26	0	0	26								+	+	+	+	+				+	Java	0	0	0	0	0	0	100	Distributed Systems Concepts and Design by Coulouris, Dollimore and Kindberg
		Concurrent Programming	3	0	0	40	0	0	40	0	0	40	+		+	+	+	+		+	+	+							Java	0	40	0	0	0	0	60	Principles of Concurrent and Distributed Programming by M. Ben-Ari, 2nd Edition 2006, Prentice-Hall
78	University of Arizona	Introduction to Parallel and Distributed Programming	2	0	0	24	0	0	24	0	0	24	+	+	+	+	+	+		+	+	+	+	+	+	+		+	Java	50	0	0	0	0	25	25	Foundations of Multithreaded, Parallel, and Distributed Programming, Gregory R. Andrews ; Addison-Wesley, 2000.Programming by

Ранг	Универзитет	Име курса	Часова предавања	Часова вежби	Часова лабораторије	Укупно часова	Укупно часова	Укупно лабораторија	Предавања за КДП	Вежбе за КДП	Лабораторија за КДП	Укупно часова за КДП	Процеси и нити	Катанци и баријере	Синхронизација	Алгоритми	Семафори	Монитори	Условни региони	Синхронизација	Алгоритми	Размена порука	Удаљени позиви процедура	Мрежно програмирање	Rendezvous	Grid Computing	Веб сервиси	P2P	Програмски језик	Пројекат	Пројекат	Лабораторијске вежбе	Квизови	Учешће у настави	Колоквијум	Испит	Литература
																																					Andrews
		Operating Systems	3	0	1	36	0	4	12	0	0	12	+	+	+	+	+					+							C	0	10	45	0	0	20	25	A.S. Tanenbaum. Modern Operating Systems (3/e). Prentice Hall.
79	University of Rochester	Operating Systems	3	0	0	42	0	0	8	0	0	8	+	+	+	+	+												C/C++	0	60	0	0	0	20	20	A.S. Tanenbaum. Modern Operating Systems (3/e). Prentice Hall.
		Parallel and Distributed Systems	3	0	0	42	0	0	15	0	0	15	+							+	+	+	+	+			+		C/C++	50	20	0	0	30	0	0	Distributed Systems: Principles and Paradigms(2nd Edition). Andrew S. Tanenbaum and Maarten Van Steen, Prentice Hall, 2007.
80	University of Utah	Operating Systems	4	0	0	46	0	0	16	0	0	16	+	+	+	+	+	+		+	+		+						C	0	70	0	0	0	30	0	Operating System Concepts, 8th Edition, by Silberschatz, Galvin, and Gagne; Operating Systems, William Stallings, Prentice-Hall, sixth edition, 2008
81	The University of Sheffield	Theory of Distributed Systems	2	0	0	38	0	0	14	0	0	14	+	+	+	+				+									CSP	0	0	0	0	0	0	100	Concurrent and Real-time Systems: The CSP approach, Steve Schneider; John Wiley & Sons, 2000
83	University of Nottingham	Operating Systems	2	0	0	28	0	0	8	0	0	8	+	+			+												C	0	0	0	0	0	0	100	A.S. Tanenbaum. Modern Operating Systems (3/e). Prentice Hall.
		Concepts of Concurrency	2	1	0	28	14	0	28	0	0	28	+	+	+	+	+	+		+	+	+	+						Java	0	25	0	0	0	0	75	Foundations of Multithreaded, Parallel, and Distributed Programming, Gregory R. Andrews ; Addison-Wesley, 2000.Programming by Andrews
		Parallel and Distributed Computing	2	1	0	28	14	0	28	0	0	28								+	+	+	+			+			Java	0	0	0	0	0	0	100	There is no required text for this course.
85	University of Basel	Distributed Information Systems	4	2	0	36	18	0	36	16	0	52								+	+					+	+	+	Java	33	0	0	0	0	0	67	There is no required text for this course.
86	Michigan State University	Operating Systems	3	0	0	42	0	0	12	0	0	12	+	+	+	+	+												Java	25	10	0	0	15	25	25	Operating Systems, William Stallings, Prentice-Hall, sixth edition, 2008
87	Case Western Reserve University	Introduction to Operating Systems	3	0	0	42	0	0	16	0	0	16	+	+	+	+	+	+											C	0	40	0	0	0	25	35	Silberschatz, Galvin and Gagne, Operating System Concepts (6th edition), 2003, John Wiley Sons

Ранг	Универзитет	Име курса	Часова предавања	Часова вежби	Часова лабораторије	Укупно часова	Укупно часова	Укупно лабораторија	Предавања за КДП	Вежбе за КДП	Лабораторија за КДП	Укупно часова за КДП	Процеси и нити	Катанци и баријере	Синхронизација	Алгоритми	Семафори	Монитори	Условни региони	Синхронизација	Алгоритми	Размена порука	Удаљени позиви процедура	Мрежно програмирање	Rendezvous	Grid Computing	Веб сервиси	P2P	Програмски језик	Пројекат	Пројекат	Лабораторијске вежбе	Квизови	Учешће у настави	Колоквијум	Испит	Литература
		Distributed Systems	3	0	0	43	0	0	43	0	0	43								+	+	+	+	+		+	+	+	C	20	30	0	0	10	20	20	Distributed Systems: Principles and Paradigms(2nd Edition). Andrew S. Tanenbaum and Maarten Van Steen, Prentice Hall, 2007.
89	Texas A&M University - College Station	Operating Systems	2	0	0	27	0	0	14	0	0	14	+	+	+	+	+	+											C/C++	40	15	0	0	0	20	25	Operating System Concepts, 8th Edition, by Silberschatz, Galvin, and Gagne; Operating Systems, William Stallings, Prentice-Hall, sixth edition, 2008
		Networks and Distributed Processing	3	0	0	33	0	0	33	0	0	33												+			+	+	C++	0	40	0	15	0	45	0	J.F. Kurose and K.W. Ross, "Computer Networking: A Top-Down Approach," Addison-Wesley, 5th edition, 2009
91	McMaster University	Operating System Concepts	3	0	0	36	0	0	18	0	0	18	+	+	+	+	+												C	25	0	0	30	0	0	45	Operating System Concepts, 8th Edition, by Silberschatz, Galvin, and Gagne; Operating Systems, William Stallings, Prentice-Hall, sixth edition, 2008
		Concurrent System Design	3	1	0	42	8	0	42	0	0	42	+	+	+	+	+	+		+		+	+						Java	0	30	0	0	0	20	50	Jeff Magee and Jeff Kramer, Concurrency: State Models & Java Programming, 2nd edition, Wiley, 2006
93	Indiana University Blomington	Distributed Systems	3	0	0	30	0	0	30	0	0	30								+	+	+	+	+	+	+	+	+	Java	50	25	0	0	0	10	15	Distributed Systems: Principles and Paradigms(2nd Edition). Andrew S. Tanenbaum and Maarten Van Steen, Prentice Hall, 2007.
94	Arizona State University - Tempe	Operating Systems	3	0	0	30	0	0	16	0	0	16	+	+	+	+	+	+											C, Java	40	0	0	0	0	22	38	Operating System Concepts, 8th Edition, by Silberschatz, Galvin, and Gagne; Operating Systems, William Stallings, Prentice-Hall, sixth edition, 2008
95	University of Birmingham	Operating Systems	2	1	0	29	7	0	16	0	0	16	+	+	+	+	+	+											C, Java	0	0	0	0	0	0	100	Operating System Concepts, 8th Edition, by Silberschatz, Galvin, and Gagne; Operating Systems, William Stallings, Prentice-Hall, sixth edition, 2008

Ранг	Универзитет	Име курса	Часова предавања	Часова вежби	Часова лабораторије	Укупно часова	Укупно часова	Укупно лабораторија	Предавања за КДП	Вежбе за КДП	Лабораторија за КДП	Укупно часова за КДП	Процеси и нити	Катанци и баријере	Синхронизација	Алгоритми	Семафори	Монитори	Условни региони	Синхронизација	Алгоритми	Размена порука	Удаљени позиви процедура	Мрежно програмирање	Rendezvous	Grid Computing	Веб сервиси	P2P	Програмски језик	Пројекат	Пројекат	Лабораторијске вежбе	Квизови	Учешће у настави	Колоквијум	Испит	Литература
		Distributed Systems	2	0	0	24	0	0	24	0	0	24								+	+	+	+	+	+				C, Java	0	0	0	0	0	0	100	Distributed Systems Concepts and Design by Coulouris, Dollimore and Kindberg
96	University of Sydney	Operating System Internals	2	4	0	26	48	0	12	16	0	28	+	+	+	+	+	+											C	0	40	0	0	0	0	60	Operating Systems, William Stallings, Prentice-Hall, sixth edition, 2008
97	University of Aarhus	Concurrency	3	3	0	21	21	0	21	21	0	42	+	+															Java	0	0	0	0	0	0	100	Concurrency: State Models and Java Programs by J. Magee and J. Kramer; Hardcover - 355 pages.
		Operating Systems	4	4	0	28	28	0	0	0	0	0	+	+															Java	0	0	0	0	0	0	100	Lubomir F. Bic and Alan C. Shaw: Operating Systems Principles, Pearson Education, 2003
99	Rice University	Operating Systems and Concurrent Programming	3	0	0	36	0	0	0	36	0	36	+	+	+	+	+	+		+	+	+	+						C/C++	50	0	0	0	0	25	25	Operating System Concepts, 8th Edition, by Silberschatz, Galvin, and Gagne; Operating Systems, William Stallings, Prentice-Hall, sixth edition, 2008
		Distributed Systems	3	0	0	33	0	0	0	33	0	33								+	+	+	+	+	+	+	+	+	C, C++, Java	50	0	0	0	50	0	0	There is no required text for this course.
101	University of Massachusetts Dartmouth	Parallel and Distributed Software Systems	3	0	0	42	0	0	42	0	0	42	+	+	+	+	+	+		+	+	+	+	+	+				C, Java	0	30	0	0	5	25	40	Andrews G.R., "Foundations of Multithreaded, Parallel, and Distributed Programming", Addison-Wesley, 2000
101	University of Western Australia	Concurrent Systems	2	0	3	26	0	33	26	0	33	59	+	+	+	+	+	+					+	+					Java	0	20	20	0	0	0	60	Principles of Concurrent and Distributed Programming, by M. Ben-Ari, Prentice-Hall, 1990 or Addison-Wesley 2006 (Second edition).

Прилог Г: Листа слика:

Слика 1: Основни прозор симулатора намењених пројектовању 10
Слика 2: Основни прозор симулатора намењених анализи већ развијених система 11
Слика 3: Вероватноћа појављивања појединих области конкурентног и дистрибуираног програмирања на анализираним универзитетима који имају барем један курс повезан са наведеном области. Праг је постављен на 2/3 26
Слика 4: Структура предложеног решења и зависност између слојева 30
Слика 5: Принципска шема компоненте логичког слоја 37
Слика 6: Приказ повезаних логичких компонената користећи логичке везе 39
Слика 7: Раздвајање интерфејса понашања од реализације 40
Слика 8: Алгоритам временски вођене симулације код јединичног кашњења 48
Слика 9: Алгоритам временски вођене симулације заснован на кашњењу кроз компоненте 49
Слика 10: Алгоритам догађајима вођене симулације 51
Слика 11: Алгоритам паралелне временски вођене симулације 53
Слика 12: Паралелна догађајима вођена симулација 55
Слика 13: Алгоритам конзервативне паралелне догађајима вођене симулације ... 57
Слика 14: Алгоритам конзервативне паралелне догађајима вођене симулације ... 58
Слика 15: Поступак интеграције 69
Слика 16: Шематски приказ панела са придруженим компонентама 76
Слика 17: Пример визуелног повезивања елемената и генерисања листе повезаних компонената 79
Слика 18: Шематски приказ архитектуре за праћење стања симулације користећи графичке репрезентације логичких компонената 81
Слика 19: Шематски приказ архитектуре за праћење стања симулације користећи временске облике сигнала 82
Слика 20: Шематски приказ архитектуре за праћење стања симулације користећи табеларни приказ стања система у појединим тренуцима 84
Слика 21: Шематски приказ архитектуре за праћење стања симулације користећи текстуални приказ стања система у појединим тренуцима 85
Слика 22: Комуникација и синхронизација између слојева SLEEP симулатора. Пуна линија представља комуникацију, а испрекидана линија представља синхронизацију. (а) Варијанта са једним током контроле; (б) Конкурентна варијанта са више нити; (в) Дистрибуирана варијанта са више нити. 111
Слика 23: Хијерархија класа логичког слоја симулатора 112
Слика 24: Хијерархија класа дигиталних вредности 116
Слика 25: Хијерархија класа предефинисане библиотеке дигиталних компонената писаних користећи програмски језик Јава 121
Слика 26: Пример двоулазног И кола дефинисаног користећи језик Јава 122
Слика 27: Хијерархија класа са превођење VHDL кода 125
Слика 28: Хијерархија класа са чување међукода и конверзију кога из VHDL у Јава програмски језик 127
Слика 29: Пример компоненте дефинисане користећи VHDL 130
Слика 30: Пример Јава кода добијеног након превођења VHDL примера 131
Слика 31: Обрада догађаја 134
Слика 32: Основни алгоритам симулације 135

Слика 33: Конзервативни алгоритам конкурентне симулације 136
Слика 34: Оптимистични алгоритам конкурентне симулације 137
Слика 35: Архитектура дистрибуираног система 138
Слика 36: Дијаграм интеракције клијента и сервера приликом започињања комуникације... 143
Слика 37: Дијаграм интеракције клијента и сервера приликом слања посла ради започињања симулације 144
Слика 38: Дијаграм интеракције сервера и радне станице приликом прослеђивања догађаја са једне радне станице на другу радно станицу 147
Слика 39: Дијаграм интеракције радних станица приликом слања догађаја на обраду 150
Слика 40: Пример дела компоненте дефинисане развијени скрипт језик 154
Слика 41: Хијерархија класа са превођење скрипт језика................................... 155
Слика 42: Блок шема система при раду са мобилним уређајима 157
Слика 43: Дијаграм интеракције клијента и сервера приликом слања скрипте... 158
Слика 44: Дијаграм интеракције сервера и радне станице приликом прослеђивања догађаја са једног мобилног уређаја на други мобилни уређај 160
Слика 45: Мерење зависности укупног времена обраде од укупног броја мобилних уређаја на којима се обавља обрада.. 162
Слика 46: Просечно време обраде једног догађаја на серверу 162
Слика 47: Мерење зависности просечног времена обраде једног задатка од броја елементарних задатака у групи 163
Слика 48: Мерење потрошње батерије при континуалној обради јединичних послова.. 164
Слика 49: Хијерархија класа презентационог слоја симулатора 165
Слика 50: Хијерархија класа слоја физике симулатора.. 167
Слика 51: Типични изглед SLEEP симулатора.. 169
Слика 52: Основни мени приликом креирања пројекта и компоненти SLEEP симулатор 171
Слика 53: Креирање интерфејса компоненте користећи Графички едитор SLEEP симулатора 172
Слика 54: Развој компонената користећи текстуални едитор и Јава програмски језик... 174
Слика 55: Развој компонената користећи текстуални едитор и VHDL програмски језик... 176
Слика 56: Додавање нове библиотеке у постојећи пројекат 177
Слика 57: Библиотеке за избор компонената симулације. Компоненте са екстензијом .sleep представљају логичке компоненте а са екстензијом .go графички интерфејс логичке компоненте 177
Слика 58: Поступак креирања логичке компоненте користећи SLEEP симулатор 179
Слика 59: Радна површ алата Symbol Graphical Editor 180
Слика 60: Доступне графичке компоненте 180
Слика 61: Повезивање приступне тачке компоненте са њеном графичком репрезентацијом.. 182
Слика 62: Развој корисничког интерфејса компоненте у Јава програмском језику 183
Слика 63: Креирање параметара компоненте 184

Слика 64: Типични начини коришћења шаблонских компонената...................... 186
Слика 65: Креирање шаблонске компоненте.. 186
Слика 66: Постављање параметара компоненте.. 187
Слика 67: Праћење стања система директним посматрањем компоненти.......... 188
Слика 68: Интерпретација боје коју линија узима током симулације и дебљине коју линија има у зависности од вредности атрибута.. 189
Слика 69: Додавање атрибута за праћење... 189
Слика 70: Временски дијаграми праћених сигнала 190
Слика 71: Табеларно праћење стања система... 190
Слика 72: Конзола симулатора у којој је могуће текстуално праћење исписа стања система у појединим тренуцима 191
Слика 73: Мени за постављање параметара симулационог слоја......................... 192
Слика 74: Одабир компонената за симулацију на специфицираном рачунару.... 193
Слика 75: Постављање параметара физике компоненте 194
Слика 76: Имплементација методе getMsg за реализацију бафера коначног капацитета. (а) Користећи кључну реч synchronized; (б) Користећи браве Locks. 199
Слика 77: Имплементација методе за синхронизацију нити на баријери. (а) Исправно; (б) Погрешно. 200
Слика 78: Основна петља серверске нити задужена за пријем клијентских захтева 201
Слика 79: Протокол клијентске стране за прихватање клијентских захтева........ 202
Слика 80: Имплементације размене објеката. (а) Исправна; (б) Погрешна. 203
Слика 81: Проширење протокола серверске стране ради дистрибуиране синхронизације. Ово проширење је потребно уметнути непосредно пре последње else гране на слици 79. 203
Слика 82: Део кода који представља имплементацију сандучића писаног користећи RMI. (а) Клијентска страна; (б) Серверска страна.............................. 204
Слика 83: Пример jdl датотеке која омогућава започињање независног посла на грид инфраструктури 206
Слика 84: Време извршавања симулације у зависности од броја елемената које се симулирају и од броја доступних радних станица.. 209
Слика 85: Конфигурација компонената које омогућавају примену “Hearth Beat” и конзервативног алгоритма симулације.. 210
Слика 86: Мрежно оптерећење приликом коришћења конфигурације која одговара hearth beat алгоритму и конзервативном алгоритму симулације код коришћења SLEEP библиотека за симулацију проблема кретања n тела у гравитационом пољу 211
Слика 87: Успех контролне групе током посматраног периода 2005-2008.......... 212
Слика 88: Успех експерименталне група током посматраног периода 2006-2008 213
Слика 89: Расподела броја компонената унутар симулатора у области Архитектуре и организације рачунара посматраних у интервалу од 200 до 6000215
Слика 90: Расподела броја веза унутар симулатора у области Архитектуре и организације рачунара посматраних у интервалу од 400 до 25000...................... 216
Слика 91: Расподела односа броја веза и броја компоненти унутар симулатора у области Архитектуре и организације рачунара... 216
Слика 92: Расподела броја улазних портова повезаних на излазни порт унутар

симулатора у области Архитектуре и организације рачунара 217
Слика 93: Расподела корака симулације и јединичног времена обраде унутар симулатора у области Архитектуре и организације рачунара 218
Слика 94: Расподела физичког време обраде једног догађаја код система састављених искључиво од логичких кола 219
Слика 95: Расподела физичког време обраде једног догађаја код система састављених од функционалних јединица 219
Слика 96: Расподела физичког време обраде једног догађаја приликом симулације кретања тела у гравитационом пољу у SLEEP симулатору током симулација.... 220
Слика 97: Расподела вероватноћа да се по иницијалном догађају генерише нови догађај (а), број излаза на које је утицао један примљени догађај (б) 221
Слика 98: Проценат паралелно активних компоненти код симулатора у области Архитектура и организација рачунара 222
Слика 99: Расподела искоришћења мреже приликом коришћења SLEEP симулатора током тестирања.. 223
Слика 100: Расподела вероватноће чувања контекста код симулатора архитектуре и организације рачунара 223
Слика 101: Расподела времена потребног за обраду контекста једне компоненте 224
Слика 102: Расподела времена потребног за графички приказ компоненте која је обрадила догађај током симулација.. 225
Слика 103: Расподела вероватноће приказа компоената и веза унутар симулатора у области Архитектуре и организације рачунара 225
Слика 104: Релативни добитак код симулатора који имају инкрементално чување контекста и симулатора који се увек враћају на почетак симулације.................. 228
Слика 105: Релативни добитак код симулатора који имају инкрементално чување контекста и симулатора који се увек враћају на почетак симулације у ситуацији да се разликује вероватноћа рестарта симулације 228
Слика 106: Релативни добитак код симулатора који имају периодично чување контекста и симулатора који се увек враћају на почетак симулације.................. 229
Слика 107: Релативни добитак код симулатора који имају инкрементално чување контекста и симулатора који се увек враћају на почетак симулације у ситуацији да се разликује вероватноћа рестарта симулације 230
Слика 108: Вероватноћа рестарта симулације у зависности од броја радних станица и логичког времена симулације... 231
Слика 109: Вероватноћа рестарта симулације у зависности од броја радних станица и броја компонената повезаних са компонентом која је емитовала догађај 231
Слика 110: Вероватноћа рестарта симулације у зависности од броја радних станица и вероватноће да порука остаје на истом рачунару 231
Слика 111: Резултати аналитичког модела и експерименталних мерења времена извршавања извршног дела симулације у системима без интеракције са корисником у функцији броја расположивих радних станица 233
Слика 112: Резултати аналитичког модела и експерименталних мерења времена извршавања извршног дела симулације у системима без интеракције са корисником у функцији броја расположивих радних станица (детаљни приказ) 233
Слика 113: Резултати аналитичког модела и експерименталних мерења времена извршавања комплетне симулације у системима са интеракције са корисником у

функцији броја расположивих радних станица 234
Слика 114: Резултати аналитичког модела и експерименталних мерења времена извршавања комплетне симулације у системима са интеракције са корисником у функцији броја расположивих радних станица (детаљни приказ) 234
Слика 115: Резултати аналитичког модела и експерименталних мерења искоришћености радних станица приликом симулације у функцији броја расположивих радних станица 235
Слика 116: Резултати аналитичког модела и експерименталних мерења времена извршавања извршног дела симулације у системима без интеракције са корисником у функцији броја симулираних компоненти 236
Слика 117: Резултати аналитичког модела и експерименталних мерења времена извршавања извршног дела симулације у системима без интеракције са корисником у функцији броја симулираних компоненти (детаљни приказ) 236
Слика 118: Резултати аналитичког модела и експерименталних мерења времена извршавања комплетне симулације у системима са интеракције са корисником у функцији броја симулираних компонената 237
Слика 119: Резултати аналитичког модела и експерименталних мерења времена извршавања комплетне симулације у системима са интеракције са корисником у функцији броја симулираних компонената (детаљни приказ) 237
Слика 120: Резултати аналитичког модела и експерименталних мерења искоришћености радних станица приликом симулације у функцији броја симулираних компонената 238
Слика 121: Изглед ANT симулатора 262
Слика 122: Пример кода ANT симулатора 262
Слика 123: Изглед CPU Sim симулатора 263
Слика 124: Пример кода CPU Sim симулатора 263
Слика 125: Изглед DigLC2 симулатора 264
Слика 126: Пример кода DigLC2 симулатора 264
Слика 127: Изглед DLXview симулатора 265
Слика 128: Пример кода DLXview симулатора 265
Слика 129: Изглед EDCOMP симулатора 267
Слика 130: Пример кода EDCOMP симулатора 267
Слика 131: Изглед HASE симулатора 268
Слика 132: Пример кода HASE симулатора 268
Слика 133: Изглед HASE-Dinero симулатора 269
Слика 134: Пример кода HASE-Dinero симулатора 269
Слика 135: Изглед JCachesim симулатора 270
Слика 136: Пример кода JCachesim симулатора 270
Слика 137: Изглед JHDL симулатора 272
Слика 138: Пример кода JHDL симулатора 272
Слика 139: Изглед Logisim симулатора 273
Слика 140: Пример кода Logisim симулатора 273
Слика 141: Изглед M5 симулатора 274
Слика 142: Пример кода M5 симулатора 274
Слика 143: Изглед RM симулатора 275
Слика 144: Пример кода RM симулатора 275
Слика 145: Изглед RSIM симулатора 277
Слика 146: Пример кода RSIM симулатора 277

Слика 147: Изглед SIMCA симулатора 278
Слика 148: Пример кода SIMCA симулатора 278
Слика 149: Изглед SimFlex симулатора 279
Слика 150: Пример кода SimFlex симулатора 279
Слика 151: Изглед Simcs симулатора 280
Слика 152: Пример кода Simcs симулатора 280
Слика 153: Изглед SimOS симулатора 282
Слика 154: Пример кода SimOS симулатора 282
Слика 155: Изглед SimpleScalar симулатора 283
Слика 156: Пример кода SimpleScalar симулатора 283
Слика 157: Изглед VSDS симулатора 284
Слика 158: Пример кода VSDS симулатора 284

Прилог Д: Листа табела:

Табела 1: Листа тема унутар појединих образовних јединица области архитектура и организација рачунара 6
Табела 2: Основне карактеристике одабраних симулатора........ 8
Табела 3: Евалуација симулатора са становишта покривања области........ 17
Табела 4: Евалуација симулатора са становишта карактеристика 20
Табела 5: Преглед тема повезаних са конкурентношћу и дистрибуираним системима које су покривене различитим областима........ 23
Табела 6: Преглед курсева који покривају област конкурентности и дистрибуираних система на основним студијама на одабраним универзитетима 25
Табела 7: Нивои апстракције компонената логичког слоја........ 33
Табела 8: Нивои апстракције веза логичког слоја........ 35
Табела 9: Нивои апстракције компонената слоја физике 71
Табела 10: Нивои апстракције веза слоја физике 73
Табела 11: Скуп операција над дигиталним вредностима........ 119
Табела 12: Карактеристике развијеног скрипт језика........ 154
Табела 13: Покривањем одабраних тема предложеним лабораторијским вежбама 197
Табела 14: Преглед најчешћих грешака у области конкурентног и дистрибуираног програмирања 198
Табела 15: Параметри симулације, као и типичне вредности добијене током анализе........ 226

Биографија аутора

Захарије Радивојевић, магистар електротехничких наука, рођен 29.09.1978. године у Смедереву република Србија од мајке Ружице Радивојевић. Основну и средњу школу завршио у Смедереву као један од најбољих ученика. Од ране младости исказивао велико интересовање за природне науке. Поред овог интересовања показао је интересовање и за историју, посебно националну историју.

Електротехнички факултет у Београду уписао 1997. године. Након пет година дипломирао као један од најбољих студената у класи са просечном оценом 9.43 током студија и оценом 10 на дипломском. У току свог студирања све своје обавезе одрађивао у предвиђеном року.

Магистарску тезу под насловом „Методологија пројектовања реконфигурабилних симулатора дигиталних система" одбранио је јула 2006. године на Електротехничком факултету смер Архитектура и организација рачунарских система и код ментора проф. др Јована Ђорђевића и проф. др Вељка Милутиновића.

Од марта 2003. ради на Електротехничком факултету у Београду на месту асистента из предмета: Основе рачунарске технике 1 и 2, Архитектура рачунара, Архитектура и организација рачунара 1, Конкурентно и дистрибуирано програмирање. Од октобра 2008. године је ангажован и као саветник управника Рачунског центра Електротехничког факултета.

Кандидат је учествовао на више међународних ФП6 и ФП7 пројекта, као и на више пројекта финансираних од Министарства за науку Републике Србије.

Коаутор је пет збирки задатака и приручника за лабораторијске вежбе који се користе на Електротехничком факултету у Београду

Аутор или коаутор три рада у међународним часописима са impact фактором са SCI листе од којих су два у директној вези са дисертацијом, једног рада у часопису, осам радова са међународних конференција и осам радова који су саопштени на националним конференцијама.

Прилог 1.

Изјава о ауторству

Потписани-а Захарије Радивојевић

број индекса ______________________

Изјављујем

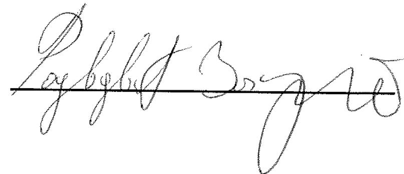
да је докторска дисертација под насловом

Методологија пројектовања симулатора архитектуре и организације рачунара

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

Потпис докторанда

У Београду, 27.04.2012.



Прилог 2.

Изјава о истоветности штампане и електронске верзије докторског рада

Име и презиме аутора ЗАХАРИЈЕ РАДИВОЈЕВИЋ

Број индекса ________________

Студијски програм ________________

Наслов рада МЕТОДОЛОГИЈА ПРОЈЕКТОВАЊА СИМУЛАТОРА АРХИТЕКТУРЕ И ОРГАНИЗАЦИЈЕ РАЧУНАРА

Ментор ДР ЈОВАН ЂОРЂЕВИЋ, РЕДОВНИ ПРОФЕСОР ЕТФ-УБ

Потписани/а ЗАХАРИЈЕ РАДИВОЈЕВИЋ

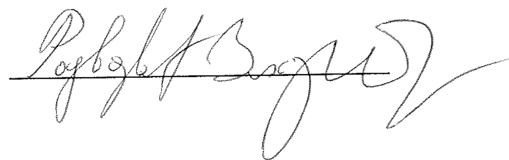
Изјављујем да је штампана верзија мог докторског рада истоветна електронској верзији коју сам предао/ла за објављивање на порталу **Дигиталног репозиторијума Универзитета у Београду.**

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

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

Потпис докторанда

У Београду, 27.04.2011.



Прилог 3.

Изјава о коришћењу

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

Методологија пројектовања симулатора архитектуре и организације рачунара

која је моје ауторско дело.

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

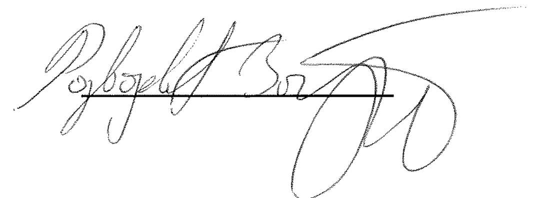
Моју докторску дисертацију похрањену у Дигитални репозиторијум Универзитета у Београду могу да користе сви који поштују одредбе садржане у одабраном типу лиценце Креативне заједнице (Creative Commons) за коју сам се одлучио/ла.

1. Ауторство
2. Ауторство - некомерцијално
3. Ауторство – некомерцијално – без прераде
4. Ауторство – некомерцијално – делити под истим условима
5. Ауторство – без прераде
6. Ауторство – делити под истим условима

(Молимо да заокружите само једну од шест понуђених лиценци, кратак опис лиценци дат је на полеђини листа).

Потпис докторанда

У Београду, 27.04.2012.



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 прераде. 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 прераде. 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.