УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА

Бранислав Кордић

Формална верификација софтверске трансакционе меморије засноване на временским аутоматима

– ДОКТОРСКА ДИСЕРТАЦИЈА –

Ментор:
проф. др Мирослав Поповић

Нови Сад, 2019

УНИВЕРЗИТЕТ У НОВОМ САДУ ● **ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА**
21000 НОВИ САД, Трг Доситеја Обрадовића 6

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА

Редни број, **РБР**:		
Идентификациони број, **ИБР**:		
Тип документације, **ТД**:		Монографска документација
Тип записа, **ТЗ**:		Текстуални штампани материјал
Врста рада, **ВР**:		Докторски рад
Аутор, **АУ**:		Бранислав Кордић
Ментор, **МН**:		Проф. др Мирослав Поповић
Наслов рада, НР:		Формална верификација софтверске трансакционе меморије засноване на временским аутоматима
Језик публикације, **ЈП**:		Српски
Језик извода, **ЈИ**:		Српски
Земља публиковања, **ЗП**:		Република Србија
Уже географско подручје, **УГП**:		Војводина
Година, **ГО**:		2019.
Издавач, **ИЗ**:		Ауторски репринт
Место и адреса, **МА**:		Нови Сад; Трг Доситеја Обрадовића 6
Физички опис рада, **ФО**: (поглавља/страна/ цитата/табела/слика/графика/прилога)		9 поглавља / 101 страна / 48 цитата / 9 табела / 12 слика
Научна област, **НО**:		Електротехничко и рачунарско инжењерство
Научна дисциплина, **НД**:		Рачунарска техника и рачунарске комуникације
Предметна одредница/Кључне речи, **ПО**:		Формална верификација, софтверске трансакционе меморије, провера модела, исправност, временски аутомати, Пајтон
УДК		
Чува се, **ЧУ**:		У библиотеци Факултета техничких наука, Нови Сад
Важна напомена, **ВН**:		
Извод, **ИЗ**:		Основни циљ истраживања дисертације је примена формализма временских аутомата за моделовање и формалну верификацију софтверске трансакционе меморије (СТМ) који се заснива на детаљима архитектуре и имплементације користећи изворни код решења. Исправност СТМ модела се утврђује машинским (аутоматским) путем употребом алата за проверу исправности модела. Моделовање СТМ, а потом и формална верификација њене исправности, је урађено употребом алата УППААЛ. Верификацијом развијених модела желе се проверити својства исправности, као што су својства сигурности, животности и достижности. Поступак примене је представљен на примеру конкретне имплементације СТМ за програмски језик Пајтон (ПСТМ). Резултати су утврдили да ПСТМ задовољава сва од претходно наведених својстава. Такође, по најбољем сазнању аутора, ПСТМ представља прву формално верификовану СТМ за програмски језик Пајтон.
Датум прихватања теме, **ДП**:		26.09.2019.
Датум одбране, **ДО**:		
Чланови комисије, **КО**:	Председник:	Др Никола Теслић, редовни професор
	Члан:	Др Мило Томашевић, редовни професор
	Члан:	Др Силвиа Гилезан, редовни професор
	Члан:	Др Миодраг Ђукић, доцент
	Члан, ментор:	Др Мирослав Поповић, редовни професор

Потпис ментора

UNIVERSITY OF NOVI SAD ● **FACULTY OF TECHNICAL SCIENCES**
21000 NOVI SAD, Trg Dositeja Obradovića 6

KEY WORDS DOCUMENTATION

Accession number, **ANO**:			
Identification number, **INO**:			
Document type, **DT**:		Monographic publication	
Type of record, **TR**:		Textual printed material	
Contents code, **CC**:		PhD Thesis	
Author, **AU**:		Branislav Kordic	
Mentor, **MN**:		Prof. Miroslav Popovic, PhD	
Title, **TI**:		Formal verification of a software transactional memory based on timed automata	
Language of text, **LT**:		Serbian	
Language of abstract, **LA**:		Serbian	
Country of publication, **CP**:		Republic of Serbia	
Locality of publication, **LP**:		Vojvodina	
Publication year, **PY**:		2019.	
Publisher, **PB**:		Author's reprint	
Publication place, **PP**:		Novi Sad, Dositeja Obradovica sq. 6	
Physical description, **PD**: (chapters/pages/ref./tables/pictures/graphs/appendixes)		9 chapters / 101 pages / 48 references / 9 tables / 12 pictures	
Scientific field, **SF**:		Electrical and Computer Engineering	
Scientific discipline, **SD**:		Computer engineering and communications	
Subject/Key words, **S**/**KW**:		Formal verification, software transactional memory, model checking, correctness, timed automata, Python	
UC			
Holding data, **HD**:		The Library of Faculty of Technical Sciences, Novi Sad, Serbia	
Note, **N**:			
Abstract, **AB**:		The main goal of the PhD thesis research is an approach to formal verification of a software transactional memory (STM) using timed automata formalism which is based on a STM design and implementation details using source code. Correctness of STM model is formally verified in automated and machine-checked manner using a model checker tool. STM models are modeled and verified using UPPAAL tool. The formal verification of the STM models aims to check the correctness of system properties, such as safety, liveness and reachability. The formal verification approach is validated using particular STM for Python, named Python Software Transactional Memory (PSTM). Verification results show that PSTM satisfies all of the above mentioned properties. Also, to the best of author's knowledge, PSTM is the first formally verified STM for the Python programming language.	
Accepted by the Scientific Board on, **ASB**:		26.09.2019.	
Defended on, **DE**:			
Defended Board, **DB**:	President:	prof. Nikola Teslic, PhD	
	Member:	prof. Milo Tomasevic, PhD	
	Member:	prof. Silvia Ghilezan, PhD	Mentor's sign
	Member:	assistant prof. Miodrag Djukic, PhD	
	Member, Mentor:	prof. Miroslav Popovic, PhD	

Искрено и од срца се захваљујем свом професору и ментору Мирославу Поповићу на указаном поверењу и подршци у раду, као и свима онима који су допринели и омогућили овај рад.

Захваљујем се и свим својим колегама са којима сам провео један диван период радећи заједно и дружећи се.

Ипак, највећу захвалност дугујем својој породици, брату Борису, мајци Јани и оцу Анти, који су увек и безрезервно били уз мене пружајући ми ослонац и давајући ми воље и снаге у тренуцима када је то било најпотребније.

САЖЕТАК

Током вишедеценијског истраживања у области Софтверских Трансакционих Меморија (СТМ) предложено је мноштво решења која се темеље на употреби различитих приступа попут коришћења традиционалних брава, механизама без закључавања, оних заснованих на времену, итд. Већина тих решења је доступна за програмске језике Ц, Ц ++, Јава, Хаскел и многе друге. Са друге стране, исправност тих решења није формално верификована, или је верификована у односу на уопштене моделе и / или алгоритме без анализе детаља саме имплементације решења, где се могу десити пропусти попут тога да неки важни делови буду изостављени или погрешно протумачени.

Основни циљ представљеног истраживања је примена формализма временских аутомата за моделовање и формалну верификацију СТМ на начин који покушава да превазиђе поменута ограничења. Начин примене формализације се заснива на прављењу верификационих модела узимајући у разматрање све детаље архитектуре и имплементације користећи изворни код решења, чиме се знатно смањује могућност грешке у моделу. Исправност СТМ модела се утврђује машинским (аутоматским) путем, употребом алата за проверу исправности модела. Моделовање СТМ, а потом и њена формална верификација, је урађено употребом алата УППААЛ. Верификацијом развијених модела желе се проверити својства исправности, као што су својства сигурности, животности и достижности. Поступак примене је представљен на примеру конкретне имплементације СТМ за програмски језик Пајтон (ПСТМ). Резултати потврђују да ПСТМ задовољава сва од претходно наведених својстава. Такође, по најбољем сазнању аутора, ПСТМ представља прву формално верификовану СТМ за програмски језик Пајтон.

Поред формалне провере, исправност ПСТМ је експериментално тестирана и потврђена употребом у оквиру реалне апликације за прорачун и симулацију модела структуре протеина ДЕЕПСАМ, у којој је ПСТМ допринела постизању боље перформансе времена извршавања, без уношења системских ограничења. С обзиром на то да је данас Пајтон један од најчешће коришћених програмских језика, потенцијал њене применљивости је веома висок.

ABSTRACT

During more than two decades of research in the Software Transactional Memory (STM) niche, various STM solutions have been proposed relying on various different implementation approaches and techniques, such as utilizing lock-free mechanisms or traditional locks, time based, etc. Many of those STMs are implemented in mainstream programming languages, such as C, C++, Java, Haskell and many others. On the other hand, correctness of such STMs is not formally verified, or they are proven against more general models or / and algorithms rather than targeting real implementations. This may cause some key aspects of implementation to be omitted or interpreted incorrectly.

In this research, an approach to formal verification of STMs is presented using timed automata formalism, which aims to solve these shortcomings. The formal verification approach relies on detailed verification models which are made by utilizing STM's design and program code, which significantly reduces possibility for errors in a verification model. Correctness of STM model is formally verified in automated and machine-checked manner using a model checker tool. STM models are modeled and verified using UPPAAL tool. The formal verification of the STM models aims to check the correctness of system properties, such as safety, liveness and reachability. The formal verification approach is validated using particular STM for Python, named Python Software Transactional Memory (PSTM). Verification results show that PSTM satisfies all of the above mentioned properties. Also, to the best of author's knowledge, PSTM is the first formally verified STM for the Python programming language.

In addition to the formal verification, the correctness of PSTM is tested and validated in a real-world application. Namely, PSTM is used in a protein structure prediction simulation program DEEPSAM for optimization of inter-process synchronization. The case study results have shown that PSTM based DEEPSAM architecture gained better performance than the original DEEPSAM, without introducing any limitations. Since Python language is one of the most used programming languages nowadays, the software component such as PSTM may be used in a broad range of applications.

САДРЖАЈ

Списак слика

Списак табела

Скраћенице

ТМ	Трансакциона Меморија (енг. Transactional Memory)
СТМ	Софтверска Трансакциона Меморија (енг. Software Transactional Memory)
ФТМ	Физичка Трансакциона Меморија (енг. Hardware Transactional Memory)
ПСТМ	Пајтон Софтверска Трансакциона Меморија (енг. Python Software Transactional Memory)
УМЛ	Обједињени језик за моделовање (енг. Unified Modeling Language)
ИоТ	Интернет ствари (енг. Internet of Things)
ЦСП	Комуницирајући секвенцијални процеси (енг. Communicating Sequential Processes)

ПОГЛАВЉЕ 1.

Увод

Трансакциона Меморија (енг. Transactional Memory – TM) је програмска парадигма [1, 2, 3] која нуди алтернативу традиционалним механизмима закључавања, као што су браве (енг. locks) – у литератури познати као груби механизам закључавања (енг. coarse-grained) – замењујући их са механизмима који се не темеље на закључавању (енг. lock-free). Као парадигма, TM је пред себе поставила два основна захтева: постизање бољих перформанси конкурентних, односно паралелних програма на савременим вишејезгарним (енг. multi-core) и многојезгарним (енг. many-core) системима, као и поједностављивање њиховог писања и одржавања.

На самом почетку истраживачке ере TM, због недостатка физичке подршке у процесорима, створена је Софтверска Трансакциона Меморија (енг. Software Transactional Memory – СТМ) [4]. Дуго времена СТМ су биле једино поље за истраживање у овој области. Архитектуре са TM подршком, тј. са физичком трансакционом меморијом, биле су доступне на ограниченом броју машина, првенствено у оним коришћеним у академске и истраживачке сврхе, а не оним које су намењене конвенционалним корисницима и индустрији. Чак и данас, упркос огромним истраживачким напорима, чини се да физичка подршка за TM још увек није стандардна у комерцијалним архитектурама. Интелово искуство са физичком трансакционом меморијом (енг. Hardware Transactional Memory - ФТМ), односно са проширењем инструкцијског скупа за подршку трансакционог

извршавања (енг. Transactional Synchronization eXtensions), показало је да упркос великом истраживачком труду ипак постоје аспекти који морају подробније бити истражени (проблем HSW136 [5]). Стога, (C)TM и даље остају отворено поље за истраживаче у покушају да се превазиђу постојећа ограничења.

Током више од две деценије истраживања у области CTM предложена су и створена су различита CTM решења и имплементације које се ослањају на разне процедуралне, објектно оријентисане и функционалне програмске парадигме и језике као што су Ц, Ц ++, Јава, Хаскел и многи други. Многи од њих се и даље у великој мери користе у истраживачкој заједници за различите пробе, прототипове, мерење перформанси и експерименталну евалуацију нових технологија у развоју. Интересантно је да, иако је Пајтон један од најзаступљенијих програмских језика, још увек нема развијену применљиву и поуздану CTM. У прошлости је било неких најава попут ПаиПаи [6], али до данас није објављено никакво коначно решење.

Постоје различити приступи за имплементацију CTM: они који користе традиционалне браве (енг. locks), механизме без закључавања (енг. lock-free), и они засновани на времену (енг. time-based), итд. Подразумевана особина у свим решењима је исправност (енг. correctness). Ова особина представља важну улогу у сваком систему, па и у систему заснованом на трансакцијама. Такође, она је и један од кључних критеријума за одабир и примењивање нове парадигме за развој нових система. Одређивање и доказивање исправности CTM решења захтева да се узме у обзир неколико аспеката пројектовања (дизајна) CTM као што су језик имплементације (преведени код се директно извршава на машини или се интерпретира), коришћени синхронизациони механизми, програмске парадигме, контекст апликације, итд.

За различите имплементације (C)TM могу се дефинисати заједнички критеријуми исправности [7, 8, 9, 10, 11, 12], који се у својој основи своде на својстава сигурности (енг. safety) и животности (енг. liveness). Уобичајено се могућност линеаризације (енг. linearizability) и својство приступа снимку конзистентних променљивих (енг. opacity) намећу као преовладавајући критеријуми исправности за својство сигурности, док се за својство животности често користе различити видови напредовања (прогресивности) као што су

својства гаранције да систем не поседује међусобно блокирање (енг. deadlock freedom), и активно међусобно блокирање без напредовања (енг. livelock freedom), као и гаранција да нема међусобне опструкције извршавања (енг. obstruction freedom). Такође, примењују се и нека прилагођавања добро познатих особина база података као што су атомичност (енг. atomicity), конзистентност (енг. consistency), изолација (енг. isolation) и трајност (енг. durability). Најчешће, ова својства се примењују на апстрактном моделу СТМ који се прави на основу спецификације СТМ и / или се преузима из трансакционе семантике, уместо да се директно извлачи из саме имплементације (пројектно решење и изворни код). Константно се тежи да верификациони модел буде што вернији представник верификованог система.

Пример мотивације за истраживање и развој у области СТМ је проблем оптимизације реалне апликације за прорачун и симулацију структуре протеина ДЕЕПСАМ која се користи у фармацеутској индустирији [13, 14]. Аутори радова описују програм симулације хемијских модела, односно програм за прорачун и предикцију структуре протеина, који је написан у програмским језицима Пајтон и Фортран. Мотив истраживања је унапређење постојећих синхронизационих механизама коришћених за међупроцесну синхронизацију у циљу постизања бољих перформанси извршавања на вишејезгреним архитектурама. У то време званично није постојало ниједно решење СТМ за програмски језик Пајтон. Као што је раније споменуто, постојале су најаве за ПаиПаи [6], али изгледа да је развој још увек у току. Као одговор на то, у ранијим истраживањима у којима је учествовао аутор, предложена су два решења СТМ за језик Пајтон [15, 16]. Оба решења су темељена на структурама података доступним у Пајтон пакету за рад у вишепроцесном окружењу (енг. multiprocessing package). Решење [15] се заснива на апстракцијама реда (енг. queue) и цеви (енг. pipe), док се решење [16] заснива на прицину рада дво-приступне меморије. Након обављених мерења и тестирања, као коначно, одабрано је решење [15].

Сигурне, безбедне и поуздане програмске компоненте на којима би се темељили безбедносно-осетљиви програми су императив. Методологија формалне провере, односно формалне верификације, омогућава детаљну анализу и проверу жељених својстава модела система попут својстава сигурности,

безбедности, итд. Од самог почетка формална верификација је коришћена за доказивање исправности различитих система и алгоритама, и одувек је била важна за безбедносно критичне системе. Традиционално, то је прва фаза у моделу развоја софтвера на принципу водопада (енг. waterfall software), као на пример у приступу развоја софтвера компаније ИБМ који се темељи на моделу чисте собе (енг. cleanroom software engineering) [17]. Данас смо сведоци широке употребе формалне верификације за проверу развојних модела софтвера који се користе у аутомобилској индустрији [18].

Нажалост, због убрзаног развојног процеса вођеног све већом потражњом, примена формалне провере развојних модела се недовољно користи. Практична примена у индустрији и научене лекције из прошлости потврдили су да се поуздана софтверска компонента мора формално верификовати. Наиме, у недавној прошлости, Интелово искуство је потврдило да су ригорозне процедуре верификације архитектуре (дизајна) решења неопходне и у области трансакционих меморија. Интел је у неким својима архитектурама, попут архитектуре Хасвел, морао онемогућити проширење инструкцијског скупа за подршку трансакционог извршавања јер његова употреба под одређеним околностима може довести до недефинисаних грешака и сигурносних проблема (проблем HSW136 [5]).

На основу сазнања из доступне литературе, постојећа СТМ решења нису развијена користећи ригорозан процес развоја заснован на моделу. Напротив, већина предложених СТМ решења је директно имплементирана у циљном програмском језику, а постојеће формално верификоване СТМ су проверене у односу на уопштене (генеричке) моделе и спецификације, без разматрања имплементационих детаља. Ово може проузроковати да неки кључни аспекти архитектуре и имплементације буду изостављени или погрешно интерпретирани. У овој дисертацији покушавају се превазићи ова два недостатка користећи приступ који би се могао применити у агилним програмима за развој софтвера, а код којег се верификациони модел система конструише на основу изворног програмског кода имплементације СТМ. Приступ је демонстриран формалном верификацијом конкретне СТМ за програмски језик Пајтон.

Основни циљ истраживања дисертације је примена формализма временских аутомата (енг. timed automata) за моделовање и формалну верификацију конкретне софтверске трансакционе меморије за програмски језик Пајтон, назване Пајтон Софтверска Трансакциона Меморија (енг. Python Software Transactional Memory - ПСТМ). Формализација ПСТМ система се заснива на анализи детаља архитектуре и имплементације користећи изворни код решења. На основу њих, прво су развијени, а потом и верификовани, верни модели ПСТМ система без изостављања детаља имплементације. Делови ПСТМ су моделовани као мрежа недетерминистичких временских аутомата (енг. nondeterministic timed automata) користећи алат УППААЛ [19, 20]. Исправност формализоване ПСТМ се утврђује машинским (аутоматским) путем употребом алата за проверу исправности модела (енг. model checker). Циљ верификације развијених модела је доказати својства исправности ПСТМ, као што су својства сигурности (енг. safety), животности (енг. liveness) и достижности (енг. reachability). Поред исправности ПСТМ, на овај начин се потврђује и њена поузданост, као и оправданост поступка моделовања и формалне верификације софтверске трансакционе меморије употребом формализма временских аутомата. Након верификације исправности, ПСТМ је експериментално тестирана и валидирана у оквиру реалне вишејезгарне наменске апликације ДЕЕПСАМ.

Хипотеза на којој се темељи истраживање је да је могуће применом формализма временских аутомата моделовати реалну софтверску трансакциону меморију, односно на основу архитектуре и изворног кода имплементације СТМ направити њен верни модел, и машинским (аутоматским) путем доказати својства коректности, односно својства сигурности и животности.

Резултати представљени у дисертацији доприносе сродним аспектима формалне верификације (С)ТМ на следећи начин: (i) по сазнању аутора, ово је прва формална верификација СТМ решења за програмски језик Пајтон, (ii) описује поступак моделовања реалне имплементације СТМ помоћу алата заснованог на формализму временских аутомата, као што је УППААЛ алат, а не разматрајући опште (генерализоване) моделе високог нивоа, већ узимајући у обзир све детаље архитектуре и имплементације користећи изворни код решења,

и, такође, (iii) шаблони детаљно параметризованих модела аутомата могу се користити као полазна тачка за верификацију система заснованих на (П)СТМ.

Резултати вишегодишњег истраживања који су део ове дисертације објављени су у радовима [21, 22 , 23].

Остатак рада је организован на следећи начин. Преглед актуелних радова у области дат је у Поглављу 2. Преглед радова обухвата друге актуелне начине и приступе решавању проблема формалне верификације (С)ТМ као и њихову примену и евалуацију у реалним апликацијама. У Поглављу 3 представљена је архитектура Пајтон Софтверске Трансакционе Меморије (ПСТМ). У Поглављу 4 описан је поступак моделовања и представљени су детаљни модели ПСТМ система. Временска анализа ивршавања трансакција у систему заснованом на ПСТМ је дата у Поглављу 5. Опис и дефиниција верификационих својстава су дати у Поглављу 6. Резултати верификације и анализа су дати у Поглављу 7. Поглавље 8 садржи студију случаја применљивости формално верификоване ПСТМ у реалној апликацији, односно у програму за прорачун и симулацију структуре протеина ДЕЕПСАМ. У Поглављу 9 изнети су закључци као и потенцијални правци даљег развоја и истраживања у области (С)ТМ.

ПОГЛАВЉЕ 2.

СТАЊЕ У ОБЛАСТИ

У овом поглављу дат је преглед актуелних радова у области који се односе на верификацију софтверских трансакционих меморија користећи различите формалне методе и приступе моделовања, као и њиховој примени у реалним апликацијама која је од интереса за екпериментални валидацију ПСТМ у програму ДЕЕПСАМ.

2.1 Приступи формалној верификацији

Коен и други у [7] су верификовали исправности имплементације трансакционе меморије користећи дозвољену размену трансакционих операција (енг. admissible interchange of transactional operations). Представили су апстрактни модел за спецификацију семантике трансакционе меморије и правило провере да ли имплементација задовољава спецификацију трансакционе меморије. Приступ је евалуиран у односу на различите класе имплементација трансакционих меморија. Имплементације су моделоване у ТЛА+ језику за дефинисање спецификација и анализиране су помоћу алата за проверу модела ТЛЦ. Касније, у [24] овај приступ је проширен да би обухватио приступ меморији који није део (софтверске) трансакционе меморије. Међутим, изведени модели имплементација коришћени у овом раду су још увек превише апстрактни јер се провера ради на нивоу спецификација.

Герови и Капалка у [25] уводе својство приступа снимку конзистентних променљивих (енг. opacity) као критеријум исправности имплементација трансакционих меморија. Модел трансакционе меморије је изведен тако да се прилагоди различитим врстама имплементације ТМ, а не да се фокусира на једну одређену (С)ТМ. У каснијим радовима [9, 26] потврђена су својства сигурности и животности за нека постојећа решења (С)ТМ. Својство сигурности подразумева строгу серијализабилност (енг. strict serializability) и својство приступа снимку конзистентних променљивих, док се код својства животности разматра непостојање опструкције (енг. obstruction freedom), непостојање активног међусобног блокирања без напредовања (енг livelock freedom) и непостојање чекања (енг. wait freedom). Даље, користећи неке од структурних својстава ТМ, општи проблем верификације исправности сведен је на општи проблем конкурентности између две нити које раде над две заједничке т-променљиве, како би се ублажио проблем експлозије броја стања (енг. the state space explosion problem). Затим су формално доказана својства у односу на тај уопштени проблем са две нити и две т-променљиве. Ипак, да би се доказала одређена својства, у оквиру верификације се претпоставља да ТМ алгоритми комуницирају са (специфичним) модулом за управљање конфликтима (енг. contention manager), што не мора увек бити случај. У предложеном верификационом приступу својства ПСТМ доказујемо без ослањања на друге додатне компоненте.

Еми и други [8] такође су свој метод верификације засновали на својству стриктне серијализабилности. За разлику од ранијих истраживања, они су развили технику за верификацију параметризованих система примењујући технике коришћене у верификацији физичких система (енг. hardware) и протокола, као и конвенционалне приступе у верификацији софтвера. Резултат је оквир за верификацију (С)ТМ који пружа ефикасну анализу система без проблема експлозије броја стања. Модели верификованих трансакционих алгоритама изведени су из [9, 25, 26].

Доерти и други у [10, 27] дефинисали су две спецификације исправности трансакционих меморија, односно TMC1 и TMC2. TMC1 обухвата карактеристике ТМ библиотека имплементираних у програмским језицима који

се преводе тако да се извршавају непосредно на физичкој машини (енг. unmanaged languages), као што су то програмски језици Ц и Ц ++. Да би се обухватио широк спектар ТМ, примена ТМС1 је уопштенија, што суштински доводи до вишег нивоа апстракције. ТМС2 се бави овим проблемом тако што се фокусира на детаље конкретне спецификације или алгоритма који стоји иза уобичајених техника примене ТМ. Обе спецификације су моделоване користећи формализам У / И аутомата (енг. I / O automaton). Осим тога, формално је потврђено да спецификација ТМС2 задовољава спецификацију ТМС1. Сличан приступ за верификацију су користили Лесани и други у [11]. Као и у претходним приступима верификацији ТМ, ТМС1 је општији и зато може изоставити детаље имплементације, док ТМС2 још увек није довољно близу стварним детаљима имплементације (С)ТМ скривеним у изворном коду.

Ел-кустабан и други у [28] предлажу други модел спецификације исправности користећи апстрактну (С)ТМ. Намера је да се он користи као основа за развој општег и флексибилног формалног оквира који може да се користи за доказивање исправности различитих ТМ система. Они користе апстрактне моделе из [7, 25], с том разликом што је хронологија догађаја моделована као временски интервал користећи формализам логике временских интервала (енг. interval temporal logic).

У свом раду, Абдула и други [29] анализирали су својства сигурности и животности хибридне трансакционе меморије узимајући у обзир аспекте специфичне за физичку и софтверску трансакциону меморију. Они су најпре приступили проблему тако што су потврдили одређена својства у односу на релативно мале, али детаљне, моделе који омогућавају поједностављење својстава сигурности и животности. Касније, слично приступу у [9, 26], својства су верификована у контексту две нити и две заједничке т-променљиве.

Верификација сложених система као што су трансакционе меморије није тривијалан нити лак задатак. Сродни радови које смо управо поменули значајно су допринели овој области демонстрирајући различите методе формализације и приступе верификацији за неке (С)ТМ. Они примењују различите технике да би доказали исправност, али обично раде са генерализованим моделима. На рачун имплементационих детаља, генерализовани модели омогућавају алатима за

верификацију да провере систем у односу на различите параметре као што су број трансакција (тј. нити) или број трансакционих променљивих. Циљ ове дисертације је да се направи верни и фино параметризовани модел СТМ решења, почевши од архитектуре решења и детаља имплементације, коришћењем временских аутомата. У претходним истраживањима представљени су прелиминарни резултати верификације за својства сигурности, животности и достижности [21]. Проширени и свеобухватни резултати истраживања објављени су у [22 , 23].

2.2 Примена у реалним системима

Традиционално, програми за мерење перформанси (енг. benchmarks) се користе као прво средство за процену перформанси система. Током година развоја и истраживања у области ТМ, развијено је неколико референтних оквира за мерење перформанси како СТМ тако и ФТМ имплементација. Без сумње, СТАМП [30] је један од најчешће коришћених програма за мерење перформанси. Сачињен је од осам апликација које циљају различите апликационе домене. Осим за СТМ, он се може користити и за евалуацију перформанси ФТМ. Такође, један од уобичајених избора је СТМБенч7 [31]. Ова два програма за мерење перформанси оригинално су имплементирана у програмским језивима Ц / Ц++ и JAВА. Међутим, упркос томе што су погодни за коришћења у раним фазама развоја и без улагања додатног напора, програми за мерење перформанси могу дати замагљену слику у случају сложених система, тако да се они углавном користе у раним фазама развоја и / или за доказивање концепта.

Зиулкиаров [32] и Гајинов [33] су међу првима који су представили сложену архитектуру реалне апликације засноване на СТМ – раније у прошлости су објављивани радови са применама СТМ, али углавном су то биле знатно поједностављене верзије реалних апликација. У свом раду, Зиулкиаров и Гајинов експериментисали су са светски познатом игрицом за више играча Квејк (енг. Quake). Гајинов и други [33] су пошли од секвенцијалне верзије имплементације и развили су Квејк ТМ, односно Квејк верзију која се заснива на проширеној верзији МцРт-СТМ [34] система. С друге стране, Зиулкиаров и други [32] су пошли од паралелне имплементације засноване на механизмима закључавања и

прилагодили је како би искористили ТМ парадигму (Атомик Квејк). У претходним случајевима коришћен је преводилац Интел СТМ Ц / Ц ++, верзије 3.0 и 2.0. У обе студије случаја дати су и анализирани су позитивни и негативни аспекти, и потенцијални проблеми приликом интеграције СТМ у сложени реални систем. Резултати показују да у неким случајевима употреба СТМ није оправдана због пада перформанси узрокованог повећаним бројем неуспешних трансакција.

Накаике и други [35] су експериментисали са ЈАВА заснованим апликацијама, и то конкретно са базом података ХСКЛДБ, апликационим сервером Геронимо и апликационим сервером ГласФиш. Све апликације су имале проблем скалабилности који је проузрокован традиционалним механизмима заснованим на закључавању. Трансакционо извршавање је обезбеђено применом СТМ библиотека заснованих на дељеним 128-битним меморијским локацијама и табели власничких записа која се користи за контролу приступа меморији и детекцију конфликата. Такође, у раду су на свеобухватан начин анализирали процес интеграције и процене напора за развој, односно прилагођавање СТМ-заснованој архитектури. Посебно су вредна следећа два закључка која треба додатно нагласити: (i) откривање конфликата узрокованих оптимизацијом на нивоу апликације и (ii) број променљивих које заиста узрокују конфликте у подацима. На први поглед, они су добијали веома лоше резултате за верзије апликација које су засноване на СТМ. Лошији резултати су били проузроковани оптимизацијом на апликативном нивоу. Наиме, оптимизација на апликативном нивоу смањује употребу и приступ меморији наметањем поновног коришћења објеката, што као споредни ефекат подразумева непотребне конфликте у подацима. Када су решили проблем оптимизације на апликативном нивоу, утврдили су да је број (трансакционих) променљивих које узрокују конфликте података мањи од броја укупних променљивих које се чувају закључавањем. Ово указује на то да је трансакциона верзија апликације лакша за писање и одржавање него што је то случај са еквивалентом заснованим на закључавању због смањеног броја променљивих, што може довести до потенцијално опасних сценарија.

Са истим циљем на уму, Х. Е. Рамадан и други [36] радили су на анализи и примени ФТМ на архитектуру оперативног система. Они су представили трансакциону верзију оперативног система Линукс, која је способна да уместо објеката заснованих на механизму закључавања користи трансакције. Подршка за извршавање трансакција је омогућена унутар језгра оперативног система и обезбеђена је за неколико главних подсистема. Рад трансакционе верзије Линукс оперативног система је симулиран на новој ФТМ архитектури званој МетаТМ, која је у ствари слична архитектури x86 са побољшаним системом за управљање прекидима.

У ранијем истраживању у којем је аутор учествовао [23] представљена је примена ПСТМ у реалној апликацији за прорачун и симулацију структуре протеина ДЕЕПСАМ који се користи у фармацеутској индустрији. Нова ДЕЕПСАМ архитектура заснована на ПСТМ има за циљ побољшање постојеће синхронизације која је у оригиналној ДЕЕПСАМ архитектури имплементирана помоћу баријера, као и обезбеђивање нових механизама синхронизације заснованих на трансакционој меморији. У раду је представљена нова ДЕЕПСАМ архитектура заснована на ПСТМ. Такође, анализиране су мере перформансе двеју архитектура као што су време извршавања система и скалабилност решења у односу на број језгара. Експериментални резултати су процењени у односу на два пептида, *енкефалин* и *2мк5*. Прелиминарни резултати показују да нова верзија ДЕЕПСАМ архитектуре заснована на ПСТМ има упоредиво или боље време извршавања у односу на оригиналну верзију. Такође, резултати ове студије нису открили никаква ограничења нове архитектуре. Детаљни резултати из [23] су представљени у Поглављу 9.

ПОГЛАВЉЕ 3.

ПАЈТОН СОФТВЕРСКА ТРАНСАКЦИОНА МЕМОРИЈА

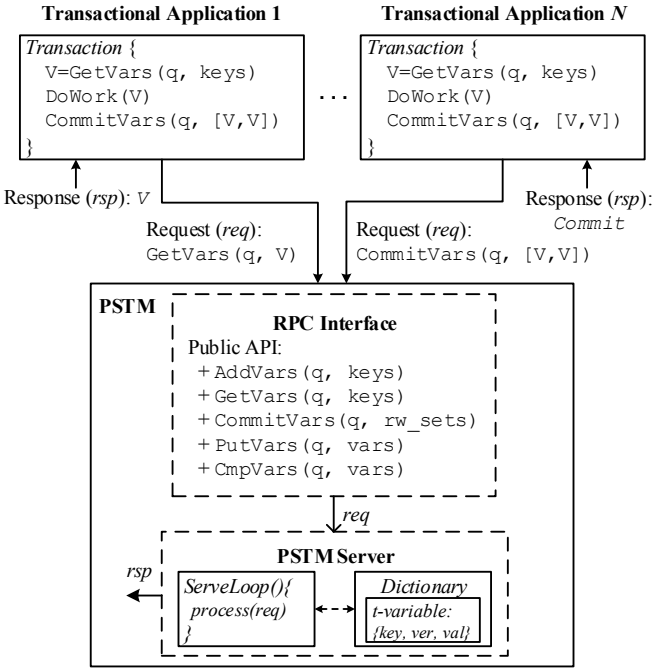
У овом поглављу описана је архитектура, функционалност и имплементација Пајтон софтверске трансакционе меморије. Њен детаљан опис први пут је објављен у [15].

Уопштено, трансакциони систем заснован на софтверској трансакционој меморији као средишњу компоненту укључује софтверску трансакциону меморију (СТМ) која је задужена за обраду трансакционих захтева. У даљем разматрању, та компонента је СТМ, која је развијена за програмски језик Пајтон, односно Пајтон Софтверска Трансакциона Меморија (ПСТМ).

У оваквом трансакционом систему, појединачна трансакција се може посматрати као низ инструкција које се извршавају непрекидиво (атомично) над скупом одређених трансакционих променљивих. Због једноставности, у наставку, трансакционе променљиве скраћено су називане т-променљиве. Уобичајени кораци понашања трансакције су следећи: (i) добављање локалног скупа т-променљивих из ПСТМ који је предмет обраде – и још се назива и *снимак* (енг. snapshot) – (ii) извршавање обраде над локалним снимком података, при чему обрада може бити произвољна и дефинисана је контекстом саме трансакције, и (iii) ажурирање вредности т-променљивих, ако за то има потребе.

Архитектура ПСТМ је приказана на слици 1. У оквиру архитектуре постоје две компоненте, а то су (i) трансакциона апликација и (ii) ПСТМ.

Трансакције се извршавају као део, и у контексту трансакционе апликације, и подржане су од стране ПСТМ. Сама ПСТМ се састоји од два важна дела, а то су (i) апликативна спрега (енг. application interface) која је доступна трансакцијама и (ii) сервер који имплементира функционалност апликативне спреге.



Слика 1 – Архитектура ПСТМ

ПСТМ обезбеђује јавну, апликативну спрегу према трансакцијама која обухвата све захтеве дефинисане њиховим уобичајеним понашањем. Функције апликативне спреге су доступне преко механизма спреге за позивање удаљене процедуре (енг. Remote Procedure Call – RPC). Механизам спреге за позивање удаљене процедуре је кључни део архитектуре ПСТМ који трансакцијама обезбеђује сигуран, истовремени, односно конкурентни приступ ПСТМ – овај механизам постиже то тако што ради серијализацију приспелих трансакционих захтева. У имплементацији, механизам спреге за позивање удаљене продецуре је представљен користећи структуру података ред (енг. queue). Све трансакције у ПСТМ систему користе (деле) јединствени ред за слање захтева.

Апликативну спрегу ПСТМ чине следеће функције:

- `AddVars(q, keys)`
- `GetVars(q, keys)`
- `CommitVars(q, rw_sets)`
- `PutVars(q, vars)`
- `CmpVars(q, vars)`

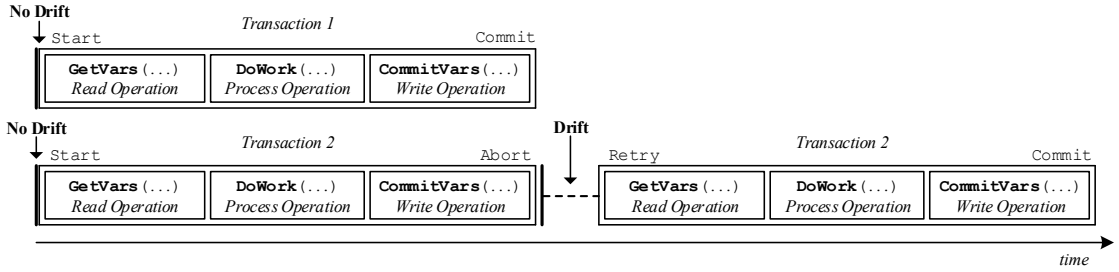
Да би се боље разумела функционалност апликативне спреге ПСТМ, представљени су појмови *речник* и *т-променљива*. Трансакциона променљива, односно т-променљива, означава променљиву складиштену унутар ПСТМ, којој се може приступити само кроз функције апликативне спреге. Свака т-променљива је јединствено одређена са три атрибута, а то су *кључ*, *верзија* и *вредност*. Кључ т-променљиве је приступни идентификатор за т-променљиву унутар трансакционе меморије. Верзија одговара редном броју измене т-променљиве, где се најновијом сматра она која је затечена у време читања. Вредност представља податке, тј. вредност одговара актуелном стању т-променљиве. Све т-променљиве унутар ПСТМ се чувају у њеном локалном речнику. Само је ПСТМ серверу дозвољено да непосредно чита и пише у речник. Речник и т-променљиве су имплементирани као структуре података речник (енг. dictionary) и торка (енг. tuple), респективно. Све функције апликативне спреге ПСТМ деле заједнички аргумент `q`. Аргумент `q` представља ред који се користи као комуникациони канал између трансакција и ПСТМ. Остали аргументи понаособ се односе на одређену функцију којој припадају. Функција `AddVars` уводи нове т-променљиве у ПСТМ. Обично се т-променљиве додају током покретања система, али нове т-променљиве се такође могу додати и у току рада. Функција `GetVars` враћа најновију верзију т-променљивих које су тренутно ускладиштене у речнику. Функција `GetVars` такође осигурава семантику невидљивог читања. Функције `AddVars` и `GetVars` као аргумент кључева `keys` очекују скуп кључева т-променљивих. Функција `CmpVars` је помоћна функција која се користи за поређење (или проверу) скупа верзија т-променљивих у односу на тренутне верзије одговарајућих т-променљивих у речнику. Аргумент `vars` означава скуп т-променљивих са свим атрибутима.

Функција `CommitVars` покушава да ажурира, тј. упише нову вредност т-променљиве у локални речник ПСТМ. Функција `CommitVars` као аргумент `rw_sets` прима два скупа т-променљивих, скуп за читање и скуп за писање. Скуп за читање се састоји од т-променљивих које су претходно прочитане, тј. састоји се од т-променљивих локалног снимка трансакције, док скуп за писање носи промене, тј. садржи измењене вредности т-променљивих које требају бити ажуриране у речнику. Функција је успешно извршена само ако су све верзије т-променљивих у локалном снимку једнаке верзијама т-променљивих у речнику, што значи да одређена трансакција има најновије верзије т-променљивих са којима ради. Верзија т-променљиве у речнику се мења тек када је трансакција успешно завршена и нова вредност уписана. Функција `PutVars` омогућава други начин да се изврши промена вредности т-променљиве.

ПСТМ сервер је кључни део ПСТМ архитектуре. ПСТМ сервер се користи за обраду захтева трансакција. Он обезбеђује функционалности иза апликативне спреге ПСТМ. Сервер преузима захтев из реда чекања редом један за другим, извршава га и шаље одговор назад ка трансакцији. Окосница ПСТМ архитектуре заснована је на конвенционалној клијент-сервер архитектури и ослања се на комуникацију више чворова ка јединственом чвору (енг. multipoint-to-point) и непосредној комуникацији између два чвора (енг. point-to-point). Трансакциони захтеви се шаљу од стране више чворова (трансакције) ка једном чвору (ПСТМ), док се одговор шаље непосредно од чвора (ПСТМ) ка чвору (трансакција). Непосредна комуникација између сервера и клијента остварена је употребом цеви (енг. pipe) где свака страна поседује једну страну комуникационог канала, тј. путање. У Пајтон имплементацији, овај механизам комуникације је доступан као структура података цеви (енг. pipe).

Семантика појединачних трансакција је чврсто везана за семантику трансакционих апликација чији су оне део. У ПСТМ моделу извршавања, трансакција започиње извршавање операцијом читања, затим следи операција обраде, и на крају, трансакција завршава операцијом писања. Операције читања и писања су операције трансакционе меморије, и извршава их ПСТМ у име трансакције. Свака трансакција захтева операцију трансакционе меморије слањем захтева ка ПСТМ преко механизма позива удаљене процедуре, након

чега чека одговор. Извршавање трансакције наставља се тек након што добије одговор. Свака трансакција појединачно извршава операцију обраде. Мапирање између функција апликативне спреге ПСТМ и трансакционих операција приказано је на слици 2. Функције `GetVars` и `CommitVars` одговарају операцијама читања и писања респективно. Функција `DoWork` је локална операција обраде трансакције и она није део апликативне спреге ПСТМ.



Слика 2 – Пример извршавања две временски померене трансакције

Слика 2 илуструје пример извршавања две трансакције у конфликту. Претпоставља се да ове две трансакције деле заједнички скуп података. У општем случају, време почетка трансакције је вођено трансакционом апликацијом. У примеру са слике, обе трансакције започињу извршавање у исто време. Због конфликта на дељеним подацима, у зависности од редоследа трансакционих захтева, само једна од њих се може извршити успешно, док се друга трансакција, уколико жели ажурирати податке, мора поново извршити. Конкретно, на примеру са слике, прва трансакција је извршила ажурирање података док је код друге то прошло неуспешно, због чега се она поново извршава.

Слично почетном времену, време поновног покушаја може бити различито за сваку трансакцију унутар скупа. Скуп трансакција за извршавање може бити сачињен од временски поравнатих трансакција (енг. aligned transactions) – трансакције које извршавање започињу истовремено (симултано), и временски померених трансакција (енг. drifted transactions) – трансакције које извршавање могу започети у било ком временском тренутку (стохастички). Унутар скупа поравнатих трансакција, све трансакције започињу извршавање у исто време, а у случају неуспелог извршавања, све трансакције одмах започињу ново извршавање. За разлику од скупа временски поравнатих трансакција, у скупу

временски померених трансакција, све трансакције могу започети поновљено извршавање на случајан начин, тј. на начин као што започињу почетно извршавање. У примеру са слике, скуп за извршавање чине две временски померене трансакције. У случају временски поравнатих трансакција, неуспешна трансакција (трансакција број два) не би чекала пре поновног покушаја. Суштински посматрано, временски померене трансакције могу да испоље понашање временски поравнатих трансакција, али не и обрнуто, односно, временски поравнате трансакције су подскуп временски померених трансакција. Са становишта откривања конфликата између трансакција, ПСТМ може бити сврстана као СТМ са лењим откривањем конфликата [2, 3].

Број ПСТМ инстанци унутар система може бити произвољан. У даљој анализи се претпоставља да систем заснован на ПСТМ поседује само једну њену инстанцу.

ПОГЛАВЉЕ 4.

ФОРМАЛИЗАЦИЈА ПСТМ

У овом поглављу описан је начин и приступ моделовања архитектуре ПСТМ употребом временских аутомата у алату УППААЛ. У првом делу поглавља представљена је архитектура УППААЛ ПСТМ модела и описане су основне функционалности и прицип рада модела. У другом делу поглавља детаљно је анализирана структура и понашање аутомата ПСТМ модела.

4.1 Моделовање ПСТМ

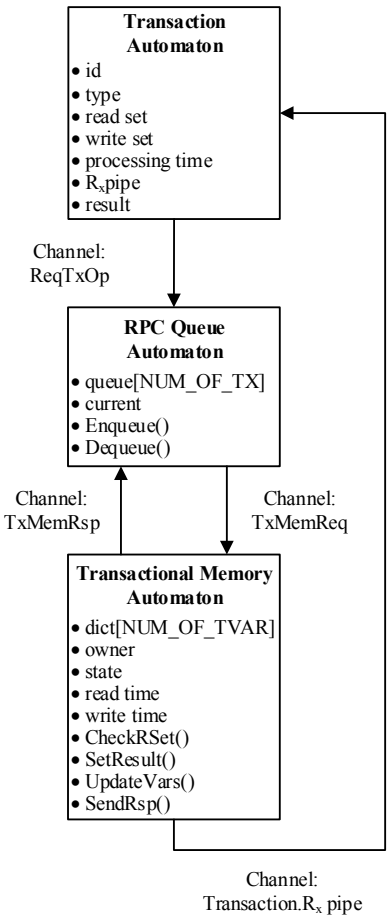
Формална верификација заснована на алату за проверу модела саставни је део процеса развоја система. Алат за проверу модела се примењује с циљем да се утврди да ли крајњи жељени систем обезбеђује одређена својства која су важна за такав систем. Скуп алата УППААЛ обезбеђује оквир за такву проверу модела [19, 20]. Алат УППААЛ је индустријски проверен алат, који се често користи и у академским истраживањима.

Верификациони модел ПСТМ система је представљен као мрежа паралелних временских недетерминистичких аутомата међусобно повезаних каналима који служе за комуникацију и / или синхронизацију. У току моделовања и верификације модела алат УППААЛ омогућује дефиницију и употребу типова података и функција сличним онима у програмском језику Ц.

Анализирајући архитектуру ПСТМ могу се идентификовати три главна дела УППААЛ ПСТМ система.

- *Трансакција*
- *Ред чекања*
- *Трансакциона меморија*

Архитектура УППААЛ ПСТМ система је приказана на слици 3. Аутомати су приказани као функционални блокови међусобно повезани комуникационо-синхронизационим каналима. Аутомати су имплементирани као шаблони заједно са својим локално дефинисаним променљивама и функцијама.



Слика 3 – Архитектура УППААЛ ПСТМ система

Аутомат *Трансакција* (енг. Transaction Automaton) моделује типично понашање трансакције приказано на слици 2. За исправну верификацију, потребно је укључити све кораке извршавања трансакције. На функционалан

начин, аутомат трансакције је еквивалент трансакцији читања и писања са дефинисаном локалном функцијом обраде.

Структура података *T_Tx* је придружена свакој инстанци аутомата *Трансакција.* Структура *T_Tx* представља контекст трансакције и прилагођена је њеној семантици. Састоји се од следећих атрибута: *id*, *type*, *read set*, *write set*, *type*, *processing time*, *Rx pipe* и *result*.

Трансакција је јединствено идентификована уз помоћ атрибута *id*. Атрибут *type* означава тип трансакције и он може бити једна од две могуће вредности, *TX_R* и *TX_W*, које означавају операције читања и писања, респективно. Скуп т-променљивих дефинисаних за читање из речника представљен је низом *read set*. Слично томе, скуп т-променљивих дефинисаних за писање у речник, тј. ажурирање вредности, представљен је низом *write set*. Та два скупа т-променљивих представљају снимак локалних података трансакције.

Канал *Rx pipe* се користе за комуникацију између трансакционе меморије и трансакција. Наменски канал *Rx pipe* је додељен свакој трансакцији. Ови канали омогућавају пренос порука (одговора) од трансакционе меморије ка свакој појединачној трансакцији. Трајање обраде трансакције је дефинисано временом обраде *processing time*. Резултат извршавања задњег траженог захтева трансакције од трансакционе меморије је смештен у променљиву *result*. У почетку, променљива *result* је постављена на вредност *TX_RSP_NONE*, што значи да трансакција још није затражила ни једну операцију. Резултат извршавања се поставља на вредност *TX_RSP_OK* сваки пут након што се операција читања успешно изврши. Ако је операција писања, односно ажурирања, успешно извршена онда се резултат поставља на вредност *TX_RSP_COMMITTED*, у супротном поставља се вредност *TX_RSP_ABORTED*.

Из перспективе ПСТМ, трансакција је извршена неуспешно (енг. abort) када се верзија прве од т-променљивих из скупа т-променљивих не слаже са верзијом те конкретне т-променљиве, која се у тренутку провере налази у речнику. То значи да величина скупа т-променљивих није пресудна. Све док се обрада трансакције заснива на најновијој верзији т-променљиве, нове вредности т-променљиве могу бити ажуриране. Ради једноставности, али без губитка општости, све трансакције су моделоване тако да деле једну т-променљиву, а као

функцију обраде, користе једноставну операцију сабирања. Наравно, трансакција извршава пар операција читања и писања пре и након локалне функције обраде, респективно.

Аутомат *Ред чекања* (енг. RPC Queue Automaton) моделује комуникациони механизам између трансакција и трансакционе меморије који је обезбеђен од стране механизма позива удаљене процедуре. То је спона која повезује трансакције са једне стране, и трансакциону меморију са друге стране.

Структура аутомата *Ред чекања* гарантује својство први на улазу – први на излазу (енг. First-In-First-Out – FIFO), стога се редослед захтева одржава – захтев који је први стигао се први обрађује. Аутомати *Ред чекања* и *Трансакција* су повезани преко заједничког канала *ReqTxOp* који одговара путањи где више чворова шаље поруке ка јединственом чвору. Са тачке гледишта трансакција, слање захтева трансакционој меморији укључује припрему података (подешавање контекста) и позивање операције слања поруке канала (*ReqTxOp*!). На супротној страни, позива се операција канала за пријем поруке (*ReqTxOp*?) где се преузимају подаци поруке које трансакција шаље.

У контексту аутомата реда чекања, улазни подаци су захтеви трансакција упућени ка трансакционој меморији, чији се садржај не анализира, већ се посматрају по принципу црне кутије. Захтеви се чувају у локалном низу *queue* унутар аутомата *Ред чекања*. Функција *Enqueue()* управља уносом и додавањем пристиглих захтева. Она ажурира низ *queue* сваки пут када на улазу аутомата пристигне нови захтев. Касније, помоћу функције *Dequeue()* захтеви на чекању се прослеђују аутомату *Трансакциона меморија*. Функција *Dequeue()* ослобађа место за нови захтев и уједно ажурира индекс *current* који показује на следећи захтев који је спреман за обраду. У току обраде, индекс *current* показује на тренутно обрађивани захтев у низу *queue*. Да би се опонашао реални сценарио, број места за захтеве у локалном низу *queue* једнак је броју трансакција у систему *NUM_OF_TX*. У сценарију у којем све трансакције захтевају операцију у исто време, сваки захтев мора бити прихваћен. Пошто трансакција може да захтева тачно једну операцију у времену, нема потребе за више места него што има трансакција.

Аутомат *Трансакциона Меморија* (енг. Transactional Memory Automaton) моделује понашање ПСТМ сервера. Аутомат *Ред чекања* је задужен за обезбеђивање улазних података аутомата *Трансакциона Меморија*. Након што аутомат *Ред чекања* прими захтев од трансакције, он обавештава аутомат трансакционе меморије. Поред обавештења, аутомат *Ред чекања* ажурира промењиве *owner* и *state*. Промењива *owner* представља тренутно обрађивани захтев, тј. захтев који је последњи послат ка трансакционој меморији. Трансакциона меморија може бити у једном од два стања, *доступна* или *заузета*.

Аутомат *Трансакциона меморија* преузима захтеве редом један по један. По пријему захтева, одређује се тип тражене операције. У случају операције писања неопходно је проверити скуп коришћених т-променљивих, што се обавља функцијом *CheckRSet()*. Функција *CheckRSet()* упоређује верзије т-променљивих из скупа *read* са одговарајућим верзијама т-променљивих које се налазе у локалном речнику ПСТМ. Као резултат, функција *CheckRSet()* враћа логички тачну вредност (енг. true), уколико верзије свих т-променљивих из скупа одговарају верзијама у локалном речнику ПСТМ, у супротном повратна вредност је логички нетачна (енг. false).

Други корак је процена исхода операције. У зависности од повратне вредности функције *SetResult()*, резултат извршавања трансакције се поставља на вредност *TX_RSP_COMMITTED* уколико је повратна вредност функције логички тачна, односно на вредност *TX_RSP_ABORT* уколико је повратна вредност функције логички нетачна. Ако је трансакција успешно извршена, функција *UpdateVars()* ажурира вредности т-променљиве унутар локалног речника користећи се вредностима т-променљивих дефинисаним у скупу *write*. Коначно, након обрађеног захтева, функција *SendRsp()* припрема и шаље одговор од аутомата *Трансакциона меморија* ка аутомату *Трансакција*. Након послате поруке аутомат *Трансакциона меморија* обавештава аутомат *Ред чекања* да је захтев сервисиран и да је трансакциона меморија слободна.

Једина разлика у понашању операције читања т-променљивих из трансакционе меморије у односу на операцију писања је то што функција *SetResult()* ажурира скуп *read* са т-променљивама из локалног речника, а скуп

write се не користи. У том случају, резултат извршења трансакције је увек позитиван *TX_RSP_READ_OK*.

Локални речник ПСТМ је моделован као низ структура *T_Tvar*. Структура *T_Tvar* одражава структуру т-променљиве у Пајтон имплементацији. Она садржи следеће атрибуте: *key*, *ver* и *val* који директно одговарају атрибутима т-променљиве описане раније (Поглавље 3). Величина речника је дефинисана променљивом *NUM_OF_TVAR*.

Верификациони параметри као што су број трансакција, трајање обраде трансакција, итд., могу варирати за сваки УППААЛ ПСТМ систем. Трајање операција трансакционе меморије је такође параметризовано. Трајање операција читања и писања моделује се посебним променљивама.

Комуникација између аутомата остварена је употребом следећих комуникационо-синхронизационих канала: *ReqTxOp*, *TxMemReq*, *TxMemRsp* и *Rxpipe* канала. Аутомати *Ред чекања* и *Трансакциона меморија* комуницирају путем канала *TxMemReq* и *TxMemRsp*. Када постоји захтев на чекању, за обавештавање аутомата *Трансакциона меморија,* аутомат *Ред чекања* користи канал *TxMemReq*. Коришћењем канала *TxMemRsp* аутомат *Трансакциона меморија* обавештава аутомат *Ред чекања* да је обрада завршена. У оквиру ПСТМ архитектуре, аутомати *Ред чекања* и *Трансакциона меморија* раде симултано, и без одлагања извршавања захтева који чекају – одмах након што постане доступан, аутомат *Трансакциона меморија* започиње обраду новог захтева, ако постоји. Да би се то постигло, *TxMemReq* и *TxMemRsp* су дефинисани као ургентни канали (енг. urgent channel) [20].

Као што је раније описано, канал *ReqTxOp* деле све инстанце аутомата *Трансакција* и користи се за слање захтева ка аутомату *Трансакциона меморија*. Свака инстанца аутомата *Трансакција* има по један наменски канал *Rx pipe* који служи за преузимање одговора непосредно од стране трансакционе меморије.

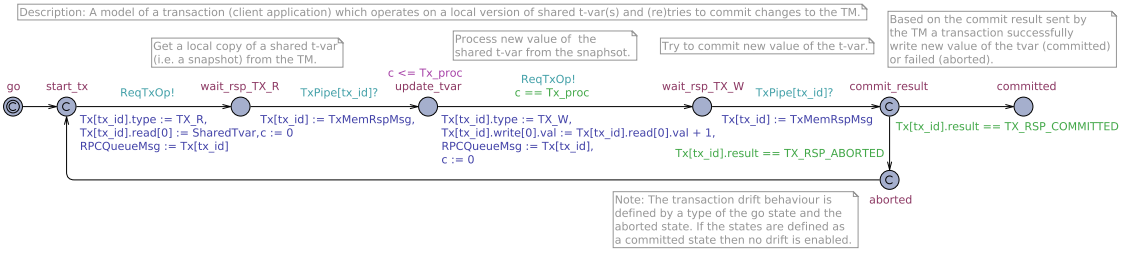
Уопштено, УППААЛ ПСТМ систем може чинити произвољан број инстанци аутомата *Трансакција*, док су инстанце аутомата *Ред чекања* и *Трансакциона меморија* јединствени.

4.2 *УППААЛ ПСТМ модели*

У овом одељку детаљно је описана унутрашња структура модела аутомата представљених у претходном одељку. У УППААЛ алату, аутомати *Трансакција*, *Ред чекања* и *Трансакциона меморија* имплементирани су као шаблони `Transaction`, `RPCQueue` и `TxMemory`, респективно.

4.2.1 Аутомат *Трансакција*

Структура аутомата `Transaction` је приказана на слици 4. Инстанца аутомата почиње извршавање са преласком из стања `go` у стање `start_tx`. Одмах након почетка, инстанца трансакције захтева операцију читања дељене т-промењиве из трансакционе меморије.



Слика 4 – УППААЛ модел трансакције - аутомат `Transaction`[1]

Из трансакционе меморије, т-промењива се чита у два корака. Први корак је слање захтева за операцију читања од трансакције према трансакционој меморији, а други корак је чекање одговора од стране трансакционе меморије. Захтев за читањем се шаље на прелазу из стања `start_tx` у стање `wait_rsp_TX_R`, позивањем команде слања на каналу *ReqTxOp*. Као део слања поруке, подешавају се подаци о трансакцији (контекст) – прво се подешава тип тражене операције *TX_R*, затим се специфицира тражена дељена т-промењива *SharedTvar* и, на крају, подаци се прослеђују аутомату `RPCQueue` преко промењиве за размену порука *RPCQueueMsg*.

Заједничка дељена т-промењива, названа *SharedTvar*, се поставља на прву позицију у низу т-промењивих дефинисаних за читање (*Tx[tx_id].read[0]*). Идентификатор трансакције *Tx_id* је дефинисан као улазни аргумент сваке инстанце аутомата `Transaction` и користи се за приступ подацима унутар

[1] Слике свих УППААЛ модела у увећаном формату су приложене у Прилогу А.

глобалног низа трансакција *Tx*. Након што је захтев послат, инстанца трансакције прелази у стање `wait_rsp_TX_R` и чека док се операција читања не обради.

Трансакција преузима одговор од стране трансакционе меморије на прелазу из стања `wait_rsp_TX_R` у стање `update_tvar`. Одговор се преузима позивом на каналу *TxPipe[tx_id]*. Као и у случају слања захтева, подаци о одговору се шаљу преко глобалне променљиве *TxMemRspMsg*. Након преласка у стање `update_tvar`, на првом месту у скупу т-променљивих *Tx[tx_id].read* се налази најновија верзија заједничке дељене т-променљиве *SharedTvar*.

Време обраде трансакција је моделовано коришћењем временске инваријанте (енг. time invariant) [20]. Наиме, трансакција напушта стање `update_tvar` након тачно *Tr_proc* временских јединица чиме се симулира време обраде. Све до тог тренутка аутомату је онемогућено да напредује. Вредност променљиве *Tr_proc* означава време обраде трансакција и дефинисана је као улазни аргумент шаблона.

Све трансакције обрађују заједничку т-променљиву на исти начин, односно повећавају њену тренутну вредност за један. Да би трансакција ажурирала вредност т-променљиве у речнику, у скупу *Tx[tx_id].write[0]* потребно је дефинисати нову вредност која ће бити коришћена у кораку ажурирања као и подесити тип захтеване операције на *TX_W*. Обе ове ставке се подешавају на одлазном прелазу из стања `update_tvar`.

Операција ажурирања, односно писања, се изводи у иста два корака као операција читања. Трансакција остаје у стању `wait_rsp_TX_W` све док се операција писања обрађује. Када је захтев обрађен, трансакција добија одговор и прелази у стање `commit_result`. У стању `commit_result`, резултат операције је познат, тако да, на основу њега, трансакција завршава своје извршавање успешно, преласком у стање `commited`, или неуспешно, преласком у стање `aborted`.

За потребе верификације, моделоване су две варијанте аутомата `Transaction`. Аутомат `Transaction` приказан на слици 4 се такође назива *циклична* или *кружна* трансакција. Циклична трансакција моделује понашање (функционалност) поновљеног извршавања трансакције (енг. retry). Наиме, након неуспелог извршавања (стање `aborted`) трансакција поново покреће

извршавање, које је претходни пут завршено неуспешно. Прецизније, циклична трансакција наставља са покушавањем све док не дође до успешног извршења.

Модел трансакције који након неуспешног извршења не покушава поново назван је *линеарна* трансакција. Након првобитног покушаја, извршавање линеарне трансакције се заврши или у стању `commited` или у стању `aborted`. Разлика у имплементацији између линеарног и цикличног модела трансакције се своди на транзицију од стања `aborted` до стања `start_tx`, која је додата моделу цикличне трансакције.

И циклична и линеарна трансакција моделују понашање временски померених и временски поравнатих трансакција. Понашање временски померене трансакције моделовано је паром стања `go` и `aborted`. Када су оба стања дефинисана као стандардна, односно подразумевана стања (енг. normal state) [20], временско померање трансакције је омогућено. У стандардном или подразумеваном стању време може напредовати, и, на тај начин, почетак извршавања трансакције може бити одложен. Временски поравнате трансакције моделоване су употребом извршних стања (енг. committed state) [20]. У тим стањима проток времена није дозвољен, те се на тај начин аутомат не задржава у том стању, већ га одмах напушта, чиме се постиже да све трансакције одмах (за)почињу извршавање.

На основу претходних описа, у верификацији ПСТМ система, могу се користити четири врсте трансакција:

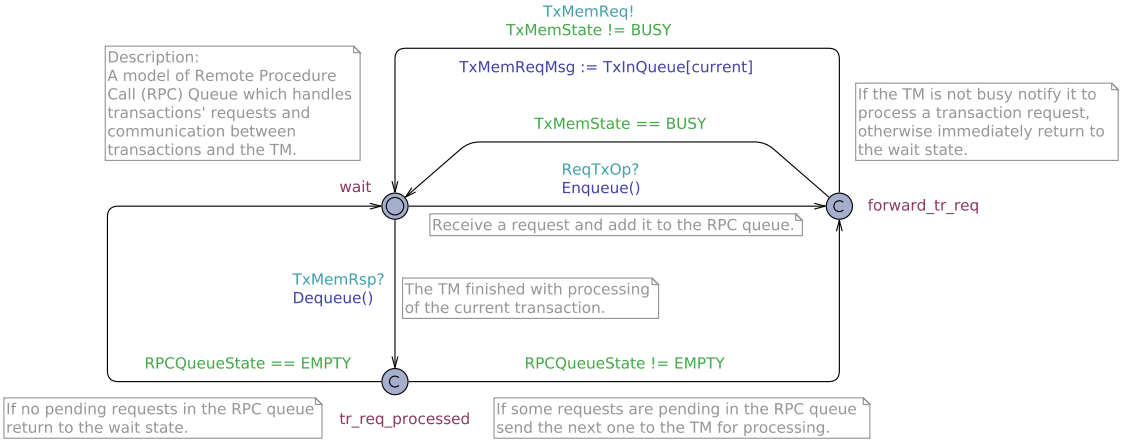
(i) циклична временски померена
(ii) циклична временски поравната
(iii) линеарна временски померена
(iv) линеарна временски поравната

4.2.2 Аутомат *Ред чекања*

Структура аутомата `RPCQueue` је приказана на слици 5. Инстанца аутомата `RPCQueue` започиње извршавање из стања `wait`. У стању `wait` аутомат чека да буде обавештен од стране неке од трансакција или трансакционе меморије. На почетку извршавања, аутомат `RPCQueue` је празан и не постоје

необрађени захтеви, односно захтеви на чекању. Број захтева који чекају представља стање аутомата и моделовано је променљивом *RPCQueueState*.

Помоћу функције *Enqueue()*, тек пристигли захтев се смешта у локални низ доступан само аутомату `RPCQueue`. Након што је захтев преузет, постоје две могућности за извршавање, које обе зависе од тренутног стања аутомата `TxMemory`. Ако је аутомат `TxMemory` заузет, онда се аутомат `RPCQueue` враћа у стање `wait` док аутомат `TxMemory` не постане доступан, у супротном аутомат `RPCQueue` прелази у стање `forward_tr_req`.



Слика 5 – УППААЛ модел реда чекања - аутомат `RPCQueue`

У другом случају, аутомат `RPCQueue` обавештава аутомат `TxMemory` да постоји захтев за обраду и одмах га прослеђује. Након што је аутомат `TxMemory` обавештен, аутомат `RPCQueue` се враћа да чека нове долазне захтеве, или обавештење да је обрада последњег послатог захтева готова.

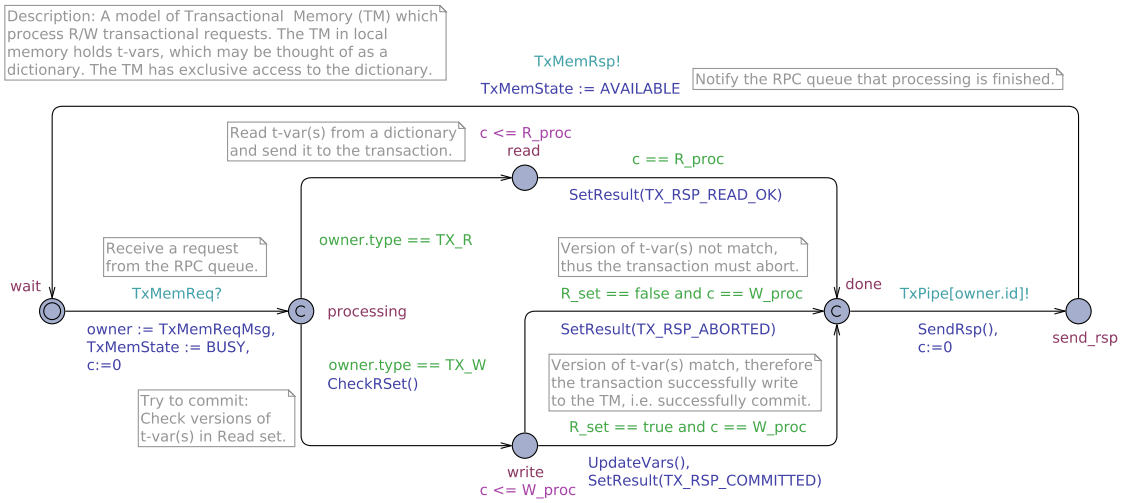
Аутомат `TxMemory`, након завршене обраде захтева, обавештава аутомат `RPCQueue`. На прелазу између стања `wait` и стања `tr_req_forward` позива се функција *Dequeue()*. Као у случају пријема новог захтева, у стању `forward_tr_req` постоје две могуће путање извршавања, али у овом случају, обе зависе од броја захтева који чекају. Ако је ред чекања празан, аутомат прелази у стање `wait` како би сачекао нови захтев. Ако ред чекања није празан, онда се први захтев из реда чекања прослеђује трансакционој меморији на обраду.

4.2.3 Аутомат *Трансакциона меморија*

Структура аутомата `TxMemory` је приказана на слици 6. Инстанца аутомата `TxMemory` започиње извршавање из стања `wait`. Докле год се налази у стању `wait`, инстанца је на располагању за преузимање нових захтева, док у супротном, инстанца је заузета обрађивањем текућег захтева.

Једино у стању `wait` аутомат `TxMemory` може бити обавештен од стране аутомата `RPCQueue`. Одмах након обавештења, аутомат прелази из стања `wait` у стање `processing`. На том прелазу ажурира се власник и стање аутомата. Власником аутомата се сматра тренутно обрађивана трансакција на основу чијег контекста су доступне све информације потребне за обраду.

У стању `processing` потребно је анализирати тип тражене операције. На основу типа операције аутомат одређује будућу путању извршавања, односно функционалност – обрађује ли се операција читања или писања. Тип операције се одређује провером атрибута операције унутар контекста трансакције.



Слика 6 – УППААЛ модел трансакционе меморије - аутомат `TxMemory`

Најпре се анализира операција читања. Обрађивање операције читања започиње преласком из стања `processing` у стање `read`. Додатно, временска инваријанта је придружена стању `read` чиме се дефинише трајање операције. Временска инваријанта осигурава да аутомат остане у стању `read` тачно *R_proc* временских јединица. Локални сат *c* служи за мерење времена. Када истекне *R_proc* временских јединица, аутомат напредује до стања `done`. Промењиве

R_proc и *W_proc* означавају трајање обраде операција читања и писања, респективно.

Одговор трансакцији се генерише на прелазу из стања `read` у стање `done`. Функција *SetResult()* ажурира припремљени скуп т-променљивих које су дефинисане за читање на тај начин што редом пролази кроз све т-променљиве из скупа и ажурира њихове вредности са вредностима из локалног речника. Такође, функција *SetResult()* ажурира резултат тражене операције. Резултат операције читања је увек успешан.

Након припремљеног одговора, обавештавају се конкретна трансакција и ред чекања. Трансакциона меморија и конкретна трансакција су синхронизоване на прелазу из стања `done` у стање `send_rsp`. У току прелаза се извршава функција *SendRsp()*. Функција *SendRsp()* шаље одговор ка трансакцији уписивањем података у променљиву *TxMemRspMbx*. Коначно, на прелазу из стања `send_rsp` у стање `wait`, трансакциона меморија обавештава аутомат реда чекања да је обрада текућег захтева готова, односно да је поново доступна.

Операција писања је моделована у два корака. Први корак је провера прочитаних т-променљивих које су коришћене у функцији обраде трансакције. У ту сврху се користи функција *CheckRSet()*. Функција *CheckRset()* проверава да ли се верзија сваке т-променљиве из локалног скупа трансакција поклапа са тренутном (најновијом) верзијом т-променљивих у локалном речнику. Функција *CheckRSet()* се извршава на прелазима из стања `processing` у стање `read` или стање `write`.

Операција писања је моделована на сличан начин као и операција читања. Додатна временска инваријанта осигурава да аутомат остане у стању `write` тачно *W_proc* временских јединица, симулирајући трајање обраде. По истеку времена *W_proc*, аутомат напредује до стања `done`.

Други корак операције писања је покушај ажурирања нових вредности т-променљивих у локални речник. Одлука о ажурирању се доноси на основу повратне вредности функције *CheckRSet()*, која се смешта у променљиву *R_set*. Вредност променљиве *R_set* је логички истинита ако се све верзије т-променљивих коришћених у трансакцији подударају са тренутним верзијама т-

променљивих у речнику. Ако се не подударају, онда је вредност логички неистинита.

Интуитивно, ако је повратна вредност функције *CheckRSet()* логички истинита, трансакција обавља ажурирање успешно, у противном ажурирање је неуспешно и обрада захтева се прекида. Обе путање извршавања су моделоване прелазима из стања `write` у стање `done`. У случају успешног ажурирања, нове вредности т-променљивих се уписују у речник а резултат трансакције се поставља на вредност *TX_RSP_COMMITED*. Ажурирање вредности у речнику се врши функцијом *UpdateVars()*. У случају неуспешног ажурирања, т-променљиве у речнику остају непромењене а резултат трансакције се поставља на вредност *TX_RSP_ABORTED*.

Коначно, на исти начин као у случају операције читања, на прелазу из стања `done` у стање `wait`, трансакциона меморија обавештава аутомат реда чекања да је обрада текућег захтева готова, односно да је поново доступна.

ПОГЛАВЉЕ 5.

АНАЛИЗА ВРЕМЕНА ИЗВРШАВАЊА ТРАНСАКЦИЈА

Вођени жељом да се направи што веродостојнији модел трансакције, моделовано је понашање временски померених и временски поравнатих трансакција. У основи, сви модели трансакција су моделовани помоћу недетерминистичких стања. Чињеница да се систем може понашати недетерминистички спречава да се извршавањем трансакција рукује на јединствен и унифициран начин. Ово значајно утиче на временску анализу система која се може анализирати и са аспекта распоређивања трансакција, што, међутим, није фокус овог рада.

За анализу временске компоненте ПСТМ система разматра се најгори случај извршавања у којем све трансакције започињу извршавање истовремено. Да би се временска анализа могла спровести, потребно је прецизирати претпоставке о трајању операција читања и писања и трајању функције обраде трансакције. Циљ ове теоријске анализе је да се направи математички оквир за проверу временског понашања ПСТМ система у најгорем случају извршавања.

Усвојене су три претпоставке. Прва претпоставка је да трајање функције обраде трансакције t_{p_i} буде исто за све трансакције, $t_{p_1} = t_{p_2} = \cdots = t_{p_N} = t_p$. Индекс t_{p_i} означава i–ту трансакцију у скупу трансакција. Прва претпоставка се заснива

на чињеници да све трансакције у скупу обично извршавају релативно кратке функције, које трају слично време извршења – на пример, можемо размотрити банкарске трансакције или неке од услужних система као што је систем за продају авио-карата, итд. Рецимо, код банковног аутомата, операције попут подизања готовине, провере депозита, преноса средстава или добијања информација о рачунима трају веома слично, ако не и идентично.

Друга претпоставка је да трајање функције обраде трансакције t_p буде исто као и време трајања трансакционих операција читања t_r и писања t_w, $t_p = t_r = t_w$. За физичке трансакционе меморије операције читања и писања се могу разликовати, што зависи од начина њихове имплементације, али у ПСТМ систему, с обзиром на то да се операције читања и писања своде на операције приступа структури података речник, претпоставка $t_r = t_w$ се чини основаном. Даље, претпоставка $t_p = t_r = t_w$ се такође чини утемељеном, с обзиром на то да су функције трансакционе обраде најчешће кратке и једноставне (лаке) а не рачунарски захтевне и интензивне.

Трећа претпоставка је да трансакција након неуспешног покушаја писања (аборта) одмах покушава поновно извршавање, и тако све док коначно не успе. Заправо, ово је понашање које моделује циклична трансакција. На овај начин, занемарује се време обраде (режије) које је у стварном систему потребно за поновно покретање трансакције. За упоредиве вредности режије резултати анализе остали би непромењени, а у супротном, анализа би била непотребно компликована и нереална за стварни систем. Дакле, ове претпоставке нам омогућавају да анализи времена извршавања трансакција приступимо на унифициран начин и без губитка општости добијених резултата.

Извршавање скупа трансакција за $N > 2$ се развија кроз три карактеристичне фазе. Ове фазе извршавања се користе за извођење формуле за рачунање времена (тренутка) завршетка трансакције. Пре детаљне анализе следе дефиниције основних појмова.

Лема 1. Време извршавања једне трансакције траје тачно $t = t_r + t_p + t_w$.

Доказ. Трансакција се извршава секвенцијално, тако да је њено време извршавања једнако трајању свих извршених операција. *Напомена*: *Лема 1* важи

у случају када је трансакција сама у систему, односно када скуп трансакција за извршавање чини само једна транскација. □

Лема 2. У сваком покушају извршавања максимално време кашњења реда чекања $d_{q_{MAX}}$ једнако је броју трансакција чије извршавање је још у току p пута време трајања неке од операција обраде трансакционе меморије, односно време трајања операције читања t_r или операције писања t_w.

Доказ. Ако p трансакција истовремено захтева неку од операција трансакционе меморије, онда је последњи послати захтев заправо p-ти захтев у реду чекања. Даље, ако операција трансакционе меморије траје неко време t_r или t_w, онда је p-ти захтев у реду обрађен након тачно p пута време t_r или t_w, респективно. □

Тврдња 1. Прва трансакција из скупа која затражи операцију читања успешно завршава извршавање у времену $t_c = N \cdot t_r + t_w$.

Доказ. Прва трансакција која затражи операцију читања предњачи у односу на остале трансакције, тако да она прва захтева и операцију писања. Захтев за операцију писања (ажурирање) се извршава одмах након последњег захтева читања, који се налази у реду чекања. На овај начин, први захтев за ажурирање касниће за максимално кашњење реда чекања $d_{q_{MAX}}$. Како вредност $d_{q_{MAX}}$ одговара броју трансакција које су још у току извршавања p, а којих на почетку извршавања има N, време завршетка прве трансакције t_c ($c = 1$) из скупа трансакција једнако је N пута време обраде операције читања t_r увећано за време обраде операције писања t_w.

Тврдња 2. Скуп трансакција завршава извршавање у времену $t_{ter_c} = t_{ter_{c-1}} + p \cdot (t_r + t_w)$.

Доказ. Сви захтеви за операцију читања се опслужују у времену $p \cdot t_r$. Како број трансакција у току извршавања остаје непромењен, потребно је $p \cdot t_w$ времена да и сви захтеви операције писања буду завршени. Последњи покушај ажурирања је завршен у тренутку када последња операција писања из реда чекања буде опслужена, $t_{ter_c} = t_{ter_{c-1}} + p \cdot t_r + p \cdot t_w = t_{ter_{c-1}} + p \cdot (t_r + t_w)$. Вредност c одговара

броју успешно завршених трансакција, односно броју покушаја извршавања скупа трансакција. У првом покушају извршавања члан $t_{ter_{c-1}}$ је једнак нули.

Тврдња 3. У покушају извршавања скупа трансакција c, где је $c \geq 2$, нека од трансакција успешно завршава извршавање у времену $t_c = t_{ter_{c-1}} + p \cdot t_r + t_w$.

Доказ. Нови покушај извршавања скупа трансакција које су завршене неуспешно започиње када захтев за операцију читања прве трансакције из скупа преосталих трансакција доспе на обраду до трансакционе меморије. Неуспеле трансакције одмах започињу поновљено извршавање слањем захтева за операцију читања, који се могу опслужити тек након завршетка свих захтева из претходног покушаја ($t_{ter_{c-1}}$). Нови скуп трансакција за извршавање се састоји од свих неуспелих трансакција из претходног покушаја, стога, прва трансакција из тог скупа успешно бива завршена у времену $t_c = t_{ter_{c-1}} + p \cdot t_r + t_w$ (*Тврдња 1*).

Тврдња 4. Последња трансакција из скупа успешно завршава извршавање у времену $t_c = t_{ter_{c-1}} + t_r + t_p + t_w$.

Доказ. У последњем покушају извршавања $c = N$, преостало је да се изврши само једна трансакција, тако да се она извршава секвенцијално (*Лема 1*). Као и у претходном случају, последњи покушај извршавања скупа трансакција започиње непосредно након што сви захтеви из претходног покушаја буду завршени ($t_{ter_{c-1}}$).

На основу пређашње анализе изведена је следећа формула за израчунавање времена завршетка трансакција у сценарију најгорег случаја извршавања:

$$t_c(c,p) = \begin{cases} N \cdot t_r + t_w, & | \; c = 1 \\ t_{ter_{c-1}} + p \cdot t_r + t_w, & | \; 2 \leq c < N \\ t_{ter_{N-1}} + t_r + t_p + t_w, & | \; c = N \end{cases} \quad (1)$$

Формула (1) израчунава време завршетка трансакције t_c као функцију броја успешно извршених трансакција c и броја трансакција које су још у току извршавања p. Укупан број трансакција у скупу означен је са N. Однос између укупног броја успешно извршених трансакција c и броја трансакција које су још у току извршавања p одређен је линеарном функцијом $N = c + p$. У почетном тренутку $c = 0$, у систему не постоји ни једна завршена трансакција а време

завршетка претходног покушаја извршавања скупа трансакција $t_{ter_{c-1}}$ се одређује рекурзивно и дефинисано је *Тврдњом 4*.

ПОГЛАВЉЕ 6.

ВЕРИФИКАЦИЈА ПСТМ МОДЕЛА

У овом поглављу дефинисана су верификациона својства система од интереса, која су коришћена у процесу провере исправности и анализирана је њихова семантика у контексту архитектуре ПСТМ. Такође, сви упити су генерализовани и изражени су у језику упита (енг. query language) који је доступан као део алата УППААЛ.

Пожељна својства готово сваког система укључују својства исправности (енг. correctness), односно сигурности (енг. safety) и животности (енг. liveness). Обично се, као један од облика својства животности, користи и провера непостојања међусобног блокирања (енг. deadlock freeness) којом се потврђује да систем неће доћи у стање међусобног блокирања. ПСТМ систем није изузетак. Али, упркос чињеници да су претходно поменута својства међу најјачима, у проверу исправности ПСТМ система је укључено и својство достижности (енг. reachability). Ово својство је од великог значаја за проверу стабилности рада система.

За проверу исправности рада ПСТМ коришћена су следећа својства:

1 Непостојање међусобног блокирања
2 Сигурност
3 Животност
4 Достижност

Сценарио извршавања који се користи за дефинисање, односно проверу сваког својства, може бити другачији. Наиме, нека својства се могу верификовати коришћењем свих типова трансакција, док је за друге погоднији сценарио извршавања ограничен на само једну врсту трансакција, јер, у супротном, провера својства би била неважећа.

6.1 Непостојање међусобног блокирања

ПСТМ систем мора задовољавати својство непостојање међусобног блокирања у било ком сценарију извршавања. Својство непостојања међусобног блокирања може се проверити следећим једноставним упитом:

```
A[] not deadlock
```

Претходни упит се именује као *Непостојање међусобног блокирања*. Идеја иза овог својства је прилично интуитивна и проверава да ли је систем поседује међусобно блокирање у некој од путања извршавања. Релевантни параметар за верификацију је број инстанци аутомата `Transaction`, што утиче на конкурентност у систему, као и на сложеност верификационог модела у погледу броја стања која треба испитати.

У циљу разумевања овог својства важно је разумети како алат УППААЛ дефинише стање међусобног блокирања. У УППААЛ алату, стање међусобног блокирања представља стање у којем нема одлазних прелаза према другим стањима нити прелаза стања у само себе (петља) [23]. С друге стране, алат УППААЛ не поседује дефинисање коначног стања извршавања симулације које је могуће унапред дефинисати и које означава исправно понашање аутомата (система). Ово је важно и за линеарне и за цикличне трансакције. Наиме, линеарне трансакције своје извршавање завршавају или у стању `aborted` или у стању `committed`, док цикличне трансакције своје извршавање завршавају само у стању `committed`. Због недостатка коначних стања ово се може сматрати међусобним блокирањем, што у основи није тачно јер је аутомат дошао до свог коначног и валидног стања, и ни у ком случају семантички не представља ситуацију у којој је систем блокиран. Да би се превазишло ово ограничење алата,

у крајња стања аутомата у којима се појављује ово нежељено понашање додаје се прелаз из стања у себе само, чиме се прави петља која онемогућује детекцију међусобног блокирања. Конкретно, ово је потребно урадити само код линеарних и цикличних трансакција. Додатни прелаз никако не смета извршавању других процеса нити утиче на њих јер, у стањима у којима су додати (`aborted` и / или `committed`), трансакциони процеси су већ завршени и нема међусобне интерференције.

6.2 Сигурност

У трансакционом окружењу, својство сигурности се обично своди на својство недељивости (енг. atomicity). Да би се стекла јасна слика о проблему, потребно је размотрити извршавање скупа трансакција у ПСТМ систему и анализирати релевантне верификационе сценарије.

Извршавање трансакција се може посматрати из перспективе извршавања трансакционе операције. Генерално, у систему заснованом на ПСТМ, свака трансакција се може понашати другачије, али да би се извршила промена т-променљивих у систему, она се мора ослањати на операције трансакционе меморије. Из ове перспективе, неопходно је да трансакционе операције буду безбедне. Карактеристика сигурности је верификована из два комплементарна угла, (i) из перспективе извршавања операција трансакционе меморије и (ii) из перспективе извршавања трансакција у случају највеће конкурентности.

Прво својство сигурности, названо *Сигурност I*, тврди да се у било ком сценарију извршавања, операција трансакционе меморије извршава недељиво (атомично), тј. трансакциона меморија увек опслужује само један захтев операције истовремено, и то тачно захтев који се тренутно налази на врху реда чекања. Својство *Сигурност I* може бити потврђено са следећим упитом:

```
A[] TxMemory.processing imply
    queue[current].id == owner.id
```

Упит се може применити на било који сценарио извршавања јер не зависи од типа трансакција. Упит у фокус ставља компоненте фундаменталне за

обезбеђивање сигурног и правичног извршења трансакционих захтева као што су трансакциона меморија и ред чекања. Упитом се проверава да ли аутомат `TxMemory` у стању обраде `processing` увек имплицира да је текући захтев у реду чекања управо захтев који поседује трансакциону меморију и тренутно је обрађивани.

Друго, снажније својство сигурности, названо *Сигурност II*, разматра сценарио највеће конкурентности у систему у којем су све трансакције у сукобу. Оно тврди да се у једном покушају извршавања само једна трансакција може извршити успешно. Ово се даље може дефинисати и на следећи начин: из скупа *N* трансакција које су у конфликту, само се једна трансакција може извршити успешно, и то трансакција чији је захтев за ажурирање примљен *први*, а тада преосталих *N-1* трансакција морају да прекину са извршавањем: $Tr_{commit} = Tr_{id=first}$; $Tr_{abort} \in \{Tr_{id\neq first}\} = \{Tr_{id=0}, Tr_{id=1} \dots, Tr_{id=N-1}\} - Tr_{id=first}$. За верификацију овог својства сценарио извршавања се састоји само од временски поравнатих линеарних трансакција. Својство *Сигурност II* може бити потврђено са следећим упитом:

```
A[] forall (i:IDs) ((i == first imply S1)
    and (i != first imply S2))

S1 := Tx[i].result != TX_RSP_ABORTED
S2 := Tx[i].result != TX_RSP_COMMITTED
```

У датом упиту *IDs* означава идентификацију трансакције. Опсег идентификација је дефинисан као [0, *N*), где је *N* број трансакција. Да би се проверио резултат трансакције, итератор *i* се користи за итерацију кроз низ контекста трансакција *Tx*. Помоћни изрази *S1* и *S2* не утичу на логику упита; они се користе у циљу читљивости верификационог упита. Израз *S1* ограничава резултат трансакције на две могуће вредности, *TX_RSP_NONE* и *TX_RSP_COMMITTED*. Израз *S2* ограничава резултат трансакције на вредности *TX_RSP_NONE* и *TX_RSP_ABORT*. Да би се изразило иницијално стање резултата извршења операције пре него што нека од операција буде позвана, користи се вредност *TX_RSP_NONE*. Помоћна променљива *first* се користи само

у сврху верификације – она чува индекс, односно идентификацију трансакције која је прва послала захтев за ажурирање. Трансакција чија је идентификација једнака променљивој *first* биће успешно извршена, а све преостале трансакције прекидају своје извршавање, тј. бивају извршене неуспешно.

6.3 Животност

Својство животности се обично користи као синоним за напредовање система. У контексту ПСТМ својство животности се односи на гаранцију да ће на крају све трансакције завршити своје извршавање. Уведена су три својства животности, *Животност I*, *Животност II* и *Животност III*. Свака дефиниција својства животности има за циљ да верификује напредак система. Разлика између њих је у броју детаља које они обухватају. Својства су дефинисана у растућем редоследу њихове јачине, односно важности (од најслабијих до најјачих). Најјача особина *Животност III* уводи временске периоде еволуције система. Својства су дефинисана у контексту *N* цикличних трансакција, како временски померених тако и временски поравнатих.

Својство *Животност I* се користи као основни (почетни) тест функционалне исправности. Оно проверава да ли је могуће да систем дође до стања у којем су све трансакције завршене. Својство *Животност I* осигурава следеће: за скуп од *N* цикличних трансакција, постоји путања извршавања система која доводи до стања у којем ће све трансакције бити успешно извршене, и то у било ком распореду трансакција. Својство *Животност I* се може проверити следећим упитом:

```
E<> forall (i:N) TxW(i).committed
```

Дефинисани упит верификације је интуитиван; постоји пут извршавања система такав да сви процеси аутомата `Transaction` коначно заврше своје извршавање у стању `committed`.

Својство *Животност II* је јаче од претходног у смислу да оно укључује трансакције које још нису завршиле своје извршавање, односно чије извршавање још траје (енг. pending transactions). Својство *Животност II* проверава да ли се

одржава тачан однос између броја извршених трансакција, трансакција које су у току и укупног броја трансакција, у било ком стању система. Овај однос дефинише укупан број трансакција у систему *txs* као збир већ извршених трансакција *c* и трансакција које су још увек у току *pending; txs = c + pending*. Поред прогресивности, систем са таквим карактеристикама такође доказује постојано и доследно понашање. Својство се може верификовати помоћу следећег упита:

```
A[] forall (i:N) ((i == pending)
   imply (L == (NUM_OF_TX - i))

L := Tx(0).committed + ... + Tx(N-1).committed
```

Идеја иза овог упита је да провери колико је трансакција завршено док се неке од њих још извршавају. Опсег *i* је дефинисан као [0, *N*] чија је вредност *N* једнака броју трансакција *NUM_OF_TX*. Вредност итератора *i* одговара броју трансакција које се још извршавају. Стварни број трансакција на чекању означен је променљивом *pending*. Вредност променљиве *pending* се ажурира сваки пут када трансакција буде извршена. Помоћни израз *L* дефинише укупан број инстанци аутомата `Transaction` *Tx(i)* које су завршиле своје извршавање и налазе се у стању `committed`. Јасно је да ако трансакција није у стању `committed`, онда је и даље у току извршавања. Однос између текућих и завршених трансакција је очигледан: ако постоји *i* успешно завршених трансакција, онда мора да постоји тачно *NUM_OF_TX - i* трансакција које су још у току.

Својство животности под именом *Животност III,* као гаранцију за напредак система, проверава временски распоред система. Оно користи формулу (1) за дефинисање горње границе временског прозора унутар којег систем напредује. За врсту својства као што је животност, временски распоред напредовања система је критичан. Пример негативног и нежељеног понашања је активно међусобно блокирање без напредовања (енг. livelock) код којег је систем активан, али ипак без суштинског напредовања Да би се потврдио позитиван

жељени напредак ПСТМ система, потребно је у времену анализирати број завршених трансакција.

ПСТМ систем напредује на позитиван начин све док се неке од трансакција завршавају. На неки начин, ово се може тумачити и као дефиниција својства *Животност I*. Заиста, својство *Животност III* се може посматрати као временски одређена верзија својства *Животност I*. Наиме, својство *Животност III* се дефинише на сличан начин, али са додатним критеријумом који дефинише горњу границу времена до којег одређени број трансакција мора да се заврши. Унутар сваког временског оквира са вишим вредностима доње и горње границе повећава се број завршених трансакција. Са оваквим приступом, својство *Животност III* се може сматрати доказом да ПСТМ систем не поседује активно међусобно блокирање. Својство *Животност III* се може верификовати помоћу следећег упита:

$$\texttt{E<>}\ \texttt{now} == t_{committ}\ \texttt{and}\ (L == num\_of\_committed)$$

$$L := \texttt{Tx(0).committed} + \ldots + \texttt{Tx(}N\texttt{-1).committed}$$
$$num\_of\_committed \in \{1,\ldots,N\}$$
$$t_{commit} = \texttt{commit\_time(}num\_of\_committed, N\texttt{)}$$

Упит се састоји од два дела. Први део упита се користи за проверу временског аспекта система. Он укључује глобални сат *now* и вредност $t_{committ}$. Глобални сат *now* се користи за мерење протока времена у систему. Очекивана вредност $t_{committ}$ значи да су се до тог тренутка требале извршити укупно *num_of_committed* трансакција. Вредност $t_{committ}$ се израчунава функцијом *commit_time()* која заправо представља имплементацију формуле (1). Вредност *num_of_committed* је параметар који је дефинисан пре верификације.

Други део упита се односи на број извршених трансакција. Помоћни израз *L* дефинише укупан број инстанци аутомата `Transaction` *Tx(i)* које су завршиле своје извршавање и налазе се у стању `committed`, док је вредност *num_of_committed* очекивани број окончаних трансакција. Да би се верификовала сва извршења скупа од *N* трансакција, потребно је дефинисати исти број упита *N*

– за сваки број успешно завршених трансакција мора се дефинисати нови упит са израчунатим временом извршења $t_{committ}$.

6.4 Достижност

Сасвим сигурно својства међусобног блокирања, сигурности и животности су довољна за верификацију функционалности модела ПСТМ. Ипак, доказивање функционалности је ојачано укључивањем својство достижности. Својство достижности се ослања на својство животности и користи се за проверу завршетка рада система – стање система након што су све трансакције завршиле извршавање. Својство достижности се користи за проверу завршетка рада система у било ком сценарију извршавања. Својство *Достижност* се може проверити следећим упитом:

```
R --> ((TxMemory.wait and TxMemState == AVAILABLE)
     and (RPCQueue.wait and RPCQueueState == EMPTY))

R := (Tx(0).committed or Tx(0).aborted)
    and … and (Tx(N-1).committed or Tx(N-1).aborted
```

Својство достижности повезује завршно стање, када су све трансакције завршене, са почетним стањем, када још ниједна трансакција није започела извршавање. У завршном као и у почетном стању система, оба аутомата, `TxMemory` и `RPCQueue`, морају бити у стању чекања `wait` означавајући да су у стању мировања. Поред тога, аутомат `TxMemory` не сме бити заузет, а `RPCQueue` не сме да садржи ни један захтев, тј. ред чекања мора бити празан. Помоћни исказ *R* дефинише тврдњу да је *N* инстанци аутомата `Transaction` завршило извршавање, без обзира на крајњи резултат сваке од трансакција.

ПОГЛАВЉЕ 7.

РЕЗУЛТАТИ ВЕРИФИКАЦИЈЕ И ДИСКУСИЈА

У овом поглављу приказани су резултати верификације ПСТМ модела. За потребе верификације коришћен је алат за проверу модела који долази као саставни део алата УППААЛ. Поступак верификације се спроводи са циљем да се тачност раније дефинисаних жељених својства система потврди независно и на машински контролисан начин.

Верификациони упити се могу применити на систем с произвољним бројем трансакција које имају различите дужине трајања операција читања, писања и функције обраде. Због тога, у поступку верификације коришћене су трансакције чија је дужина трајања операција читања, писања и функције обраде једнака једној временској јединици, $t_p = t_r = t_w = 1$. Ове конкретне вредности чине процес верификације одрживим, при чему се не губи на општости примењене методе.

Током поступка верификације број трансакција је повећаван за један све док је спровођење експеримената било изводљиво. Повећање броја трансакција значајно утиче на време верификације тако што увећава простор стања које алат за проверу модела мора да истражи како би донео одлуку о својству система које проверава. Заправо, у неком тренутку, величина простора стања заузима целокупну радну меморију рачунара на којем се спроводи верификација. У том

случају, употребом механизма страничења, алат је способан да искористи ресурсе масовне меморије, што веома утиче на укупно време извршавања.

У спроведеним експериментима број трансакција је повећан за по једну трансакцију до тачке у којој алат не почне да користи масовну меморију система као ресурс, односно све док се оперативна меморија рачунара не препуни. Број и тип трансакција зависи од својства система које је у том тренутку проверавано. Резултат верификације у којој је дошло до потрошње свих ресурса сматран је неодлученим (енг. inconclusive). За овакве случајеве овај приступ делује као најобјективнији и најнеутралнији.

Сумарни резултати верификације су дати у табели 1. Приложени резултати садрже исход верификације, статистику и мере перформансе које откривају више детаља о самом процесу верификације, као што су број претражених стања од стране алата за проверу модела и време трајања верификације. Резултати верификације су потврдили да ПСТМ модел задовољава сва својства дефинисана у претходном одељку (ознака Т). Верификације код којих је број трансакција превазишао капацитет радне меморије означене су као неодлучене (ознака -). Детаљни резултати процеса верификације су доступни у Прилогу Б.

Општи закључак о резултатима верификације се углавном односи на сложеност верификованог система. Сложеност система се значајно повећава увећањем броја трансакција што је директна последица ширења простора стања која се морају проверити. Поред броја трансакција, релевантан је и тип коришћених трансакција. На пример, верификација својстава која користи скуп цикличних трансакција свакако је захтевнија од верификације својстава које користе скуп линеарних трансакција чија унутрашња структура је мање сложена.

Такође, употреба временски померених трансакција још више усложњава процес провере. Разлог је укорењен у структури аутомата и начину на који се моделује временско померање трансакције. Наиме, обрада извршних стања у којима није дозвољено задржавање, чиме су моделоване временски поравнате трансакције, је мање захтевна него обрада стандардних стања у којима је дозвољено задржавање, тј. протицање времена, чиме су моделоване временски померене трансакције.

Несумњиво, број временских променљивих (инваријанте) значајно утиче на простор стања који се мора истражити. Алат УППААЛ је веома осетљив на број временских променљивих што утиче на брже ширење простора претраге. У текућем моделу, свака трансакција користи по један локални сат за моделовање интерног трајања функције обраде. Такође, трансакциона меморија користи један сат за моделовање трајања операције обраде читања и писања. Уопштено говорећи, смањење броја временских променљивих позитивно би утицало на величину простора стања –смањило би га – чиме би се створио простор за већи број трансакција.

Табела 1 – Сумарни резултати верификације

Својство	Број трансакција	Тип трансакције	Време	Број стања	Резултат
Непостојање међусобног блокирања	6	Линеарна померена	1m 2s 110ms	9 045 757	T
	5	Циклична померена	1m 2s 440ms	10 140 401	T
	8	Линеарна поравната	1m 2s 90ms	8 014 336	T
	7	Циклична поравната	24s 930ms	3 597 232	T
Сигурност I	6	Линеарна померена	32s 180ms	9 045 757	T
	5	Циклична померена	34s 670ms	10 140 401	T
	8	Линеарна поравната	34s 790ms	8 014 336	T
	7	Циклична поравната	12s 950ms	3 597 232	T
Сигурност II	8	Линеарна поравната	38s 740ms	8 014 336	T
Животност I	5	Циклична померена	1m 0s 20ms	17 489 881	T
	7	Циклична поравната	13s 450ms	3 597 232	T
Животност II	5	Циклична	5s 510ms	1 690 633	T

		померена			
	7	Циклична поравната	12s 420ms	3 577 073	T
Животност III	7	Циклична поравната	13s 680ms	3 577 073	T
Достижност	6	Линерна померена	32s 500ms	9 186 157	T
	5	Циклична померена	52s 620ms	14 008 901	T
	8	Линеарна поравната	34s 650ms	8 094 976	T
	7	Циклична поравната	13s 500ms	3 662 752	T

Експерименти су спроведени унутар оперативног система Убунту 14.04 ЛТС 64bit и на рачунару опремљеном процесором Intel Core Intel® Core ™ i7-3770 CPU @ 3.40GHz × 8 64bit и са 16 GB радне меморије. Коришћен је алат УППААЛ 64-4.1.19 (рев. 5648) са академском лиценцом.

ПОГЛАВЉЕ 8.

СТУДИЈА СЛУЧАЈА ПРИМЕНЕ ПСТМ У ПРОГРАМУ ДЕЕПСАМ

Експериментална евалуација ПСТМ спроведена је применом у реалној апликацији за прорачун и симулацију структуре протеина ДЕЕПСАМ, која се користи у фармацеутској индустрији. Поступак и резултати експерименталне евалуације су представљени у овом поглављу.

8.1 Увод

Као парадигма, ТМ је пред себе поставила два основна захтева: постизање бољих перформанси конкурентних, односно паралелних програма на савременим архитектурама опремљеним са више језгара, као и поједностављивање писања и одржавања таквих програма. Као што је раније напоменуто, постоје многе имплементације софтверских, физичких, па и хибридних трансакционих меморија, али са друге стране, постоји свега неколико примера њихове примене у реалним апликацијама. Многе од њих се још увек користе у истраживачкој заједници за пробе и експерименталне процене, које се предоминанто спроводе на (синтетичким) програмима за мерење перформанси, или значајно поједностављеним апликацијама уместо у стварним.

Свакако најпопуларнији међу програмима за мерење перформанси (С)ТМ су СТАМП [30] и СТМБенч7 [31]. Оба програма краси богат скуп тестних

апликација за процену перформанси. Међутим, резултати мерења употребом таквих програма понекад могу дати само увид у стварне перформансе. Ретки примери евалуације и мерења перформанси СТМ у реалним апликацијама представљени су у [32, 33, 35] (одељак 2.2.).

Многи математички задаци у хемијским проблемима, као што су проверавање структуре молекула за откривање нових лекова [37], секвенцирање нове генерације [38], предвиђање структуре протеина [39], и још много тога, представљају сложене и рачунарски захтевне задатке из реалног живота чији је главни проблем дужина времена извршавања. Примена ТМ парадигме управно покушава превазићи овакве проблеме.

У овој студији случаја анализирана је примена ПСТМ у програму ДЕЕПСАМ [13, 14]. Паралелни програм за предвиђање структуре протеина ДЕЕПСАМ написан у језицима Пајтон и Фортран, чија укупна величина је реда 100.000 линија кода. Главни циљеви употребе ПСТМ су: (i) побољшавање постојеће синхронизације процеса оригиналне верзије ДЕЕПСАМ засноване на баријерама, (ii) анализа утицаја ПСТМ на перформансе тако сложеног система и (iii) провера исправности формално верификоване ПСТМ у пракси.

Резултати студије случаја примене ПСТМ у програму ДЕЕПСАМ представљени су у [23].

Студија је организована на следећи начин. Прелиминарне информације о програму ДЕЕПСАМ и ПСТМ дате су у одељку 8.1. Пројектовање нове архитектуре и интеграција ПСТМ у програм ДЕЕПСАМ описана је у одељку 8.2. Валидација и експериментални резултати су представљени у одељку 8.3. Закључци спроведене студије су дати у одељку 8.4.

8.1.1 Програм ДЕЕПСАМ

У овом кратком прегледу дате су основне информације о програму ДЕЕПСАМ. Детаљнији опис је доступан у радовима [13, 14].

Потенцијална енергетска површина биомолекула (пептида, протеина, итд.) је вишедимензионална функција која има експоненцијални број локалних минимума који су могућа решења за те биомолекуле (приближно10^N, где је N дужина секвенце биомолекула). Добро је познато да је природна структура биомолекула одговара глобалном минимуму потенцијала енергетске површине.

Узимајући у обзир секвенцу аминокиселина биомолекула и израчунавањем тог глобалног минимум, могуће је решити проблем предвиђања структуре протеина [40], то јест, могуће је предвидети природну структуру тог биомолекула.

За процену глобалног минимума, ДЕЕПСАМ користи еволуциони алгоритам [41, 42, 43]. Овај алгоритам врши ефикасну претрагу простора претражујући паралелно скуп од $n > 1$ широко распрострањених подрегија. Тих n подрегија су одређене n молекуларним решењима која се насумично генеришу у почетном кораку стварања популације еволутивног алгоритма [44], из готово линеарне молекуларне структуре одређене биомолекулском секвенцом. Еволутивни алгоритам је написан у Пајтон 2.x, док је програмски пакет ТИНКЕР [45], који је везан за молекуларно моделовање написан у Фортрану 77.

ДЕЕПСАМ оператори мутације, названи ДЕМСА, комбинују комплементарне предности три добро познате процедуре минимизације: ДЕМ (енг. Diffusion Equation Method) [46], СА (енг. Simulated Annealing) [47], и квази-*Newton L-BFGS* [48] локалну процедуру минимизације. У свакој итерацији скуп од пет различитих ДЕМСА оператера се извршава на свакој од n потенцијалних решења тренутне популације, производећи породицу од шест потенцијалних решења – главно потенцијално решење и пет потенцијалних решења потомака. Ова обрада је имплементирана у два нивоа паралелизма: један процес се ствара за свако од n решења у тренутној популацији, а пет ДЕМСА процеса се стварају као подпроцеси сваког од тих n процеса.

У свакој итерацији еволутивног алгоритма задржава се популација до тада најбољих решења пронађених током извршавања алгоритма. Када се алгоритам заустави, пронађена решења се узимају као најбоља процена глобалног минимума потенцијала енергетских површина биомолекула.

ДЕЕПСАМ нуди четири алтернативна начина рада: паралелни, серијско-паралелни, паралелно-серијски и серијски. Ако ДЕЕПСАМ ради у (потпуно) паралелном режиму, онда свих $6n$ процеса раде паралелно. Ако ДЕЕПСАМ ради у серијско-паралелном режиму, n процеса првог нивоа паралелизма се стварају секвенцијално, један по један, али, на сваком од њих, пет процеса другог нивоа се покреће паралелно – у сваком тренутку паралелно се извршава само шест процеса. Ако се ДЕЕПСАМ покреће у паралелно-серијском режиму, n процеса

првог нивоа раде паралелно, и, за сваки од њих, пет процеса другог нивоа се извршавају секвенцијално, један по један – у било ком тренутку 2*n* процеса се извршавају паралелно. Ако се ДЕЕПСАМ покреће у серијском режиму, покрећу се само два процеса јер се сви процеси другог нивоа покрећу секвенцијално. Подразумевана величина популације (вредност *n*) коју користи ДЕЕПСАМ је пет, и то је вредност која је коришћена у овом истраживању. Ова вредност *n* је такође коришћена у претходним истраживачким радовима у којима је ДЕЕПСАМ произвео успешна предвиђања структуре биомолекула.

8.1.2 ПСТМ за Пајтон 2.7x

За интеграцију ПСТМ у ДЕЕПСАМ потребно је обезбедити извршавање ПСТМ на одговарајућој верзији Пајтона, конкретно, то је верзија 2.7.14. Иницијално, ПСТМ је имплементиран у Пајтон верзији 3.5.1. Иако је спрега ка процесима између верзија готово идентична, ипак постоје неке разлике. Главни проблем је био немогућност слања (дељења) одређених објеката комуникације између процеса, што јесте познат проблем. Овај проблем се појавио интерно у ПСТМ имплеметацији механизма позива удаљене процедуре. Наиме, потребно је динамички направити канал који се прослеђује ка трансакцији (други процес) преко које она прима повратну поруку од стране ПСТМ. Проблем управо јесте динамичко дељење објеката између два процеса.

Идеја решења за корекцију проблема је следећа. Главни процес иницијално прави канал за сваког клијента ПСТМ, укључујући и себе самог, и уводи мапу *pname_to_pipe* која мапира име процеса на конкретни канал. Ова мапа се затим прослеђује заједно са ПСТМ редом како ПСТМ серверу тако и клијентским процесима (трансакцијама). Недостаци овог решења су следећи: (1) комплетна мапа *pname_to_pipe* мора бити направљена на почетку извршавања, и (2) апликативна спрега ПСТМ мора бити мало измењена. С друге стране, предност је у томе што се канали праве једном, а затим се само користе, што је ефикасније уместо да се динамички праве и уништавају за сваки позив функције.

За имплементацију решења апликативна спрега ПСТМ измењена је тако што је параметар *q* замењен са новим параметром који се зове *comm_links* (комуникациони линкови), што је торка (*q*, *pname_to_pipe*). У мапи *pname_to_pipe*, имена процеса представљају кључ, а мапиране вредности су

канали организовани у торке (*parent_conn*, *child_conn*), где *parent_conn* и *child_conn* представљају делове канала намењених ПСТМ и конкретној трансакцији, респективно.

8.2 Пројектовање програма ДЕЕПСАМ заснованог на ПСТМ

У овом одељку изложени су кључни аспекти синхронизације процеса потомака на првом нивоу паралелизма у програму ДЕЕПСАМ. Такође, описане су модификације ДЕЕПСАМ програма урађене током ПСТМ интеграције.

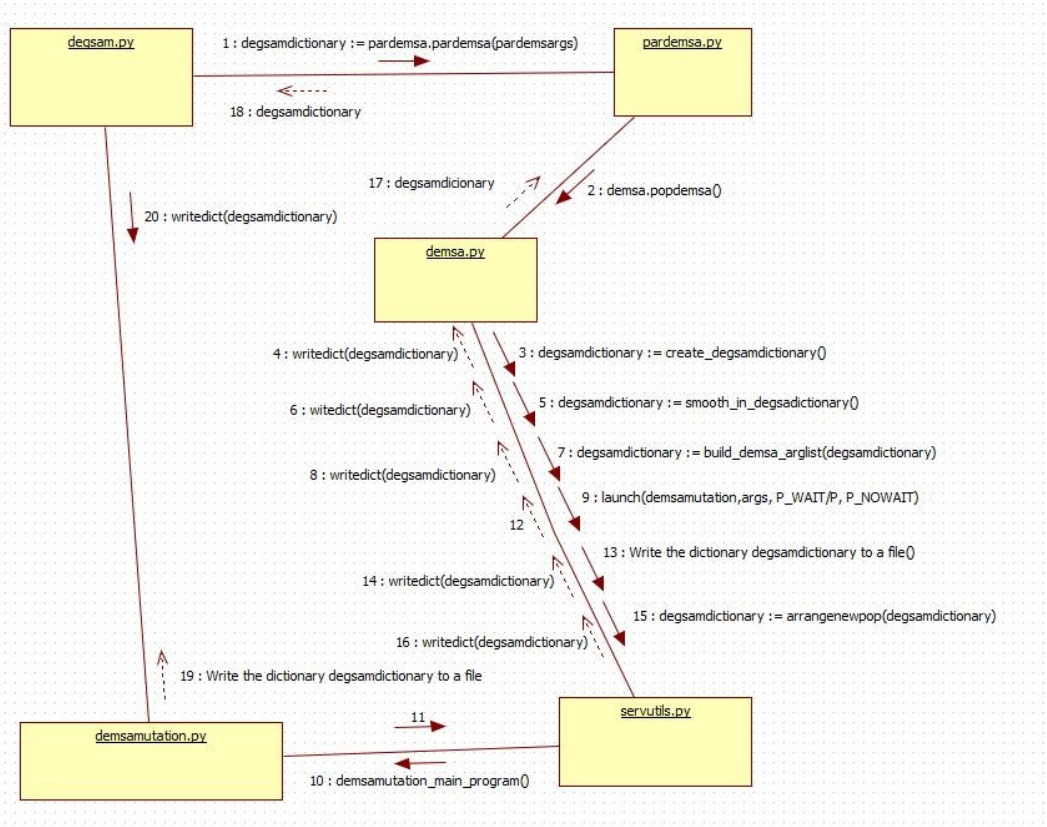
8.2.1 Архитектура програма ДЕЕПСАМ

УМЛ дијаграм сарадње програма ДЕЕПСАМ је приказан на слици 7. ДЕЕПСАМ започиње извршавање са главним процесом *degsam* који се налази у модулу *degsam.py*. На основу улазних аргумената, он извршава главну итерациону петљу еволутивног алгоритма. Главна функција синхронизације се налази у модулу *demsa.py*. Он координише мрешћење и синхронизацију Пајтон процеса који чине први ниво паралелизма. Због чињенице да је обрада алгоритма имплементирана у два различита окружења, Пајтон и Фортран, као и због претходних избора решења у пројектовању архитектуре ДЕЕПСАМ, синхронизација процеса је урађена коришћењем система датотека оперативног система.

Централна структура података која садржи кључне детаље за процесе потомке је *degsamdictionary* речник. У првој итерацији, на основу улазних параметара, речник се иницијализује, а потом се у свакој итерацији редовно ажурира. Главни процес га дистрибуира својим процесима потомцима чувањем у датотеци *degsamdict.dat*. Процеси потомци (први ниво паралелизма), који су имплементирани као Пајтон скрипте, користе га само за читање. Касније, на основу прочитаних података, сваки од ових процеса потомака ствара нову групу Фортран процеса (други ниво паралелизма).

Након овог корака, зависно од начина рада, процеси се извршавају паралелно, серијски, или комбиновано. У оквиру ове студије, анализиран је серијско-паралелни и паралелно-серијски начин извршавања процеса.

Када се Фортран програми заврше, њихов предак (Пајтон процес) преузима резултате обраде. Када сви *degsam* процеси заврше са извршавањем, и када се контрола врати у *degsam* процес, главни процес преузима податке обраде његових потомака и ажурира *degsamdictionary* речник. После тога, *degsam* процес улази у следећу итерацију, а описани циклус се понавља све док се не изврше све итерације еволуционог алгоритма.

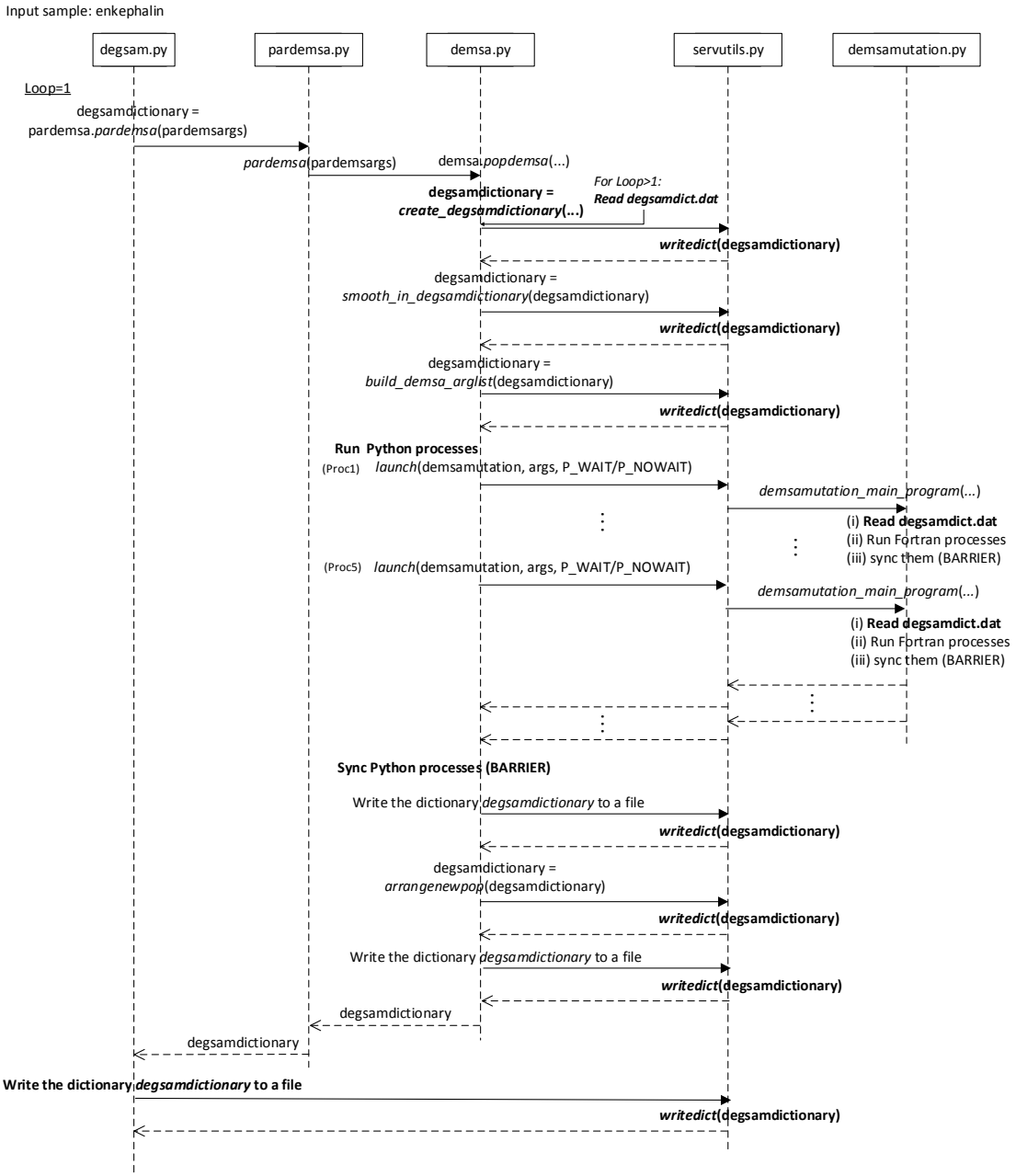


Слика 7 – Дијаграм сарадње програма ДЕЕПСАМ

Недостаци овог начина синхронизације су: (i) улазни подаци се процесима потомцима преносе преко скупа И / У операција читања и писања, који су спори у односу на извршавање процеса, и (ii) као баријера за синхронизацију Пајтон процеса првог нивоа паралелизма коришћено је упошљено чекање унутар бесконачне петље (енг. busy waiting).

Датотека *degsamdict.dat* се ажурира при позиву функције *writedict()*. Анализом тока података програма утврђено је да је функција *writedict()* добра

полазна тачка за увођење ПСТМ функционалности. Дијаграм размене порука, односно тока података програма ДЕЕПСАМ је приказан на слици 8.

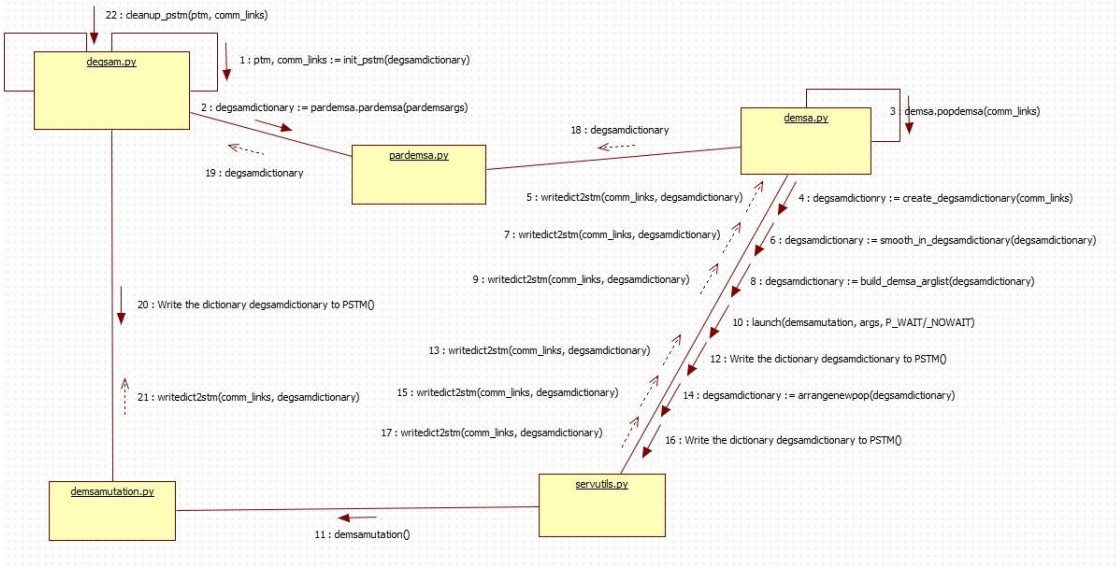


Слика 8 – Дијаграм размене порука програма ДЕЕПСАМ

8.2.2 Архитектура програма ДЕЕПСАМ заснована на ПСТМ

УМЛ дијаграм сарадње ДЕЕПСАМ програма заснованог на ПСТМ приказан је на слици 9. Интеграција ПСТМ је спроведена у четири корака. У првом кораку идентификовани су сви модули који користе застарелу Пајтон спрегу за руковање процесима. Конкретно, да би се избегло дељење датотека

између Пајтон процеса у првом нивоу паралелизма, сви процеси морају бити направљени употребом новог пакета за управљање процесима (*multiprocessing module*). Он омогућава коришћење модерних међупроцесних комуникационих механизама, као што су редови (енг. queue) и цеви (енг. pipe), који су неопходни за ПСТМ функционалност. Укупно су модификована само два модула, модул *degsam* и модул *demsamutation*.

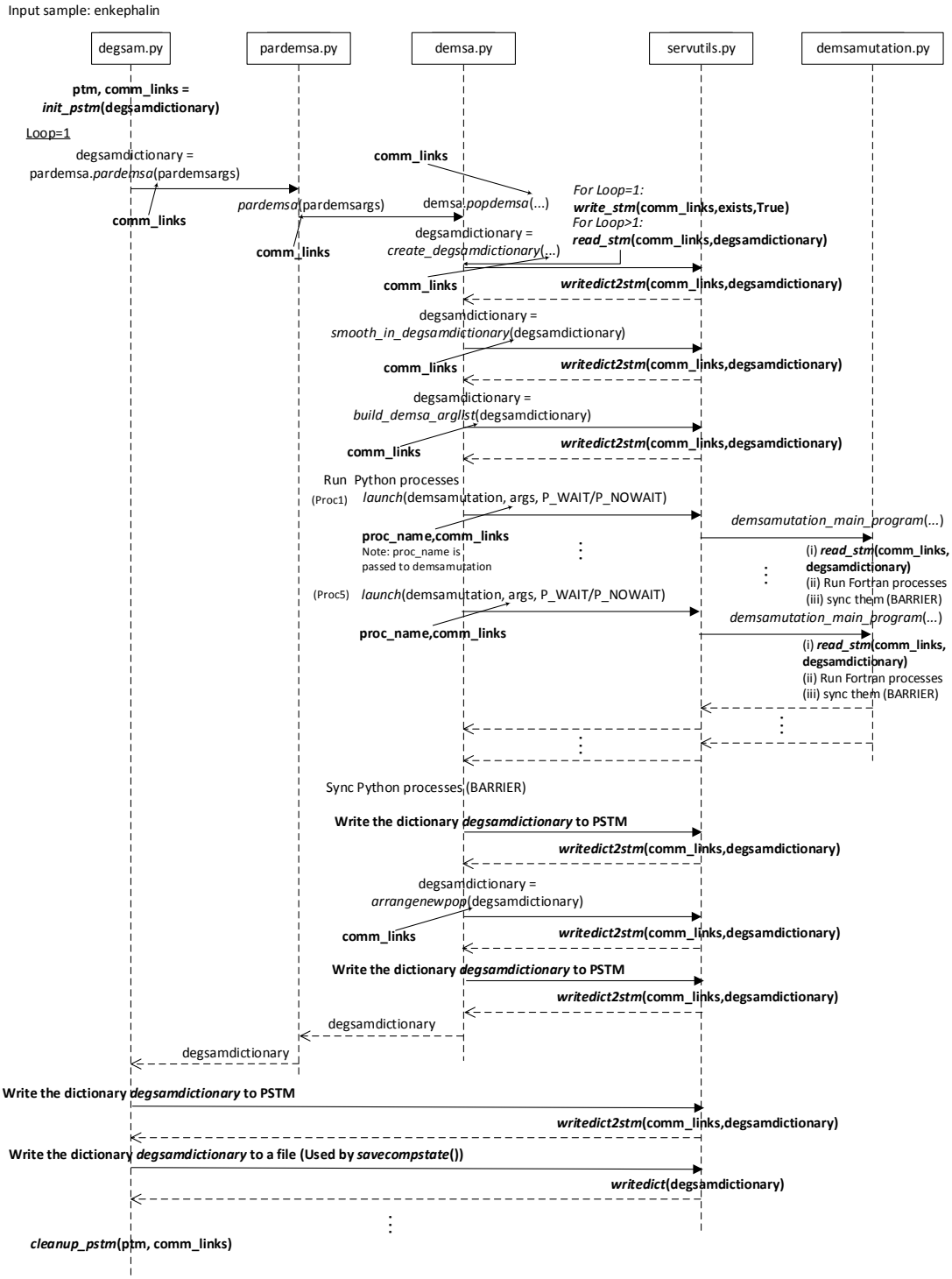


Слика 9 – Дијаграм сарадње програма ПСТМ-ДЕЕПСАМ

У другом кораку извршена је иницијална интеграција ПСТМ. Циљ овог корака је да се оспособи основна функционалност ПСТМ унутар ДЕЕПСАМ, без ометања извршавања главног алгоритма. ПСТМ функционалност је имплементирана у модулу *stm.py*. Због велике сложености и величине програма ДЕЕПСАМ, постепено се прелазило на коришћење ПСТМ функционалности. У овом кораку уз помоћ функција *init_pstm()* и *cleanup_pstm()* омогућено је успешно иницијализовање и гашење ПСТМ. Функција *init_pstm()* прави комуникационе везе, иницијализује т-променљиве и покреће ПСТМ сервер. Функција *cleanup_pstm()* брише све т-променљиве и комуникационе везе и зауставља ПСТМ сервер.

У трећем кораку сви Пајтон процеси првог нивоа паралелизма, укључујући и главни процес *degsam*, поседују скуп канала *comm_links* који се користе за комуникацију са ПСТМ. Као аргумент функција *popdemsa()* и *demsamutation_main_program()* скуп канала *comm_links* се преноси до модула

degsam.py и *demsamutation.py*, респективно. Функција *launch()* је такође модификована – она као параметре прима скуп канала и име процеса, и користи их за покретање Пајтон процеса. Након трећег корака, сви Пајтон процеси поседују све потребне податке за почетак коришћења ПСТМ функционалности.



Слика 10 – Дијаграм размене порука програма ПСТМ-ДЕЕПСАМ

У четвртом кораку уведене су т-променљиве које одговарају коришћеним променљивима у оригиналном решењу, чиме се у потпуности прешло на коришћење ПСТМ. У смислу трансакционог система, као што је ПСТМ, процеси првог нивоа паралелизације су представљени као трансакције које користе услуге (П)СТМ, а њихови улазни-излазни речници који су дељени као датотеке су представљени као т-променњиве. Стога, уместо чувања и ажурирања садржаја *degsamdictionary* речника у датотеци, главни процес га чува и ажурира у ПСТМ, а процеси потомци читају његов садржај без приступа датотекама, односно без позива И / О операција. На исти начин, уместо писања у излазне датотеке, процеси потомци чувају своје резултате у ПСТМ, што их чини тренутно доступним главном процесу. Дијаграм размене порука, односно тока података програма ДЕЕПСАМ заснованог на ПСТМ је приказан на слици 10.

Подаци у ПСТМ се уписују помоћу функције *writedict2stm()*. За разлику од функције *writedict()*, функција *writedict2stm()* објекат речника *degsamdictionary* уместо у датотеку *degsamdict.dat* уписује у ПСТМ као т-променљиву. За потребе провере стања речника уведена је т-променљива *exist*. Она се користи у првој итерацији еволуционе петље да би се проверило да ли речник постоји. Семантика ове т-променљиве је иста као у претходном случају, али са разликом да је сада складиштена у ПСТМ. У ДЕЕПСАМ окружењу, функције *stm_read()* и *stm_write()* обезбеђују спрегу за читање и писање т-променљивих.

8.3 Валидација и резултати

Кроз спроведене експерименте настојао се одредити утицај ПСТМ на перформансе система као што су време обраде и скалабилност система, а уједно се желело и потврдити исправност ПСТМ.

8.3.1 Тестирање и валидација

Структура пептида *енкефалин* је мање сложена од пептида *2мк5*, који је и значајно дужи. Енкефалин је погодан за почетне експерименте и интензивно је коришћен током развојне фазе, јер је његово време извршавања релативно кратко у поређењу са другим пептидима или протеинима. Због тога, енкефалин није погодан за добијање релевантних резултата скалабилности система. Да би се тестирала скалабилност, потребно је користити дуже пептиде и протеине.

Међутим, употребом пептида који се називају "мини" протеини (дуги између 10 и 20 аминокиселина), и стварних протеина који садрже много више аминокиселина, времена обраде могу постати многоструко дужа.

У покушају да се направи компромис између интензивних обрада које дуго трају, а да се ипак обезбеде релевантни резултати за анализу утицаја ПСТМ, кориштен је пептид 2мк5, који производи знатно дуже време обраде у односу на енкефалин. С једне стране, у поређењу са протеином, пептид 2мк5 је мање сложен (дуг је само 10 аминокиселина), али са друге стране, у поређењу са енкефалинским пептидом, његова структура је знатно сложенија – на пример, просечно време обраде 2мк5 је реда величине веће од просечног времена обраде енкефалина.

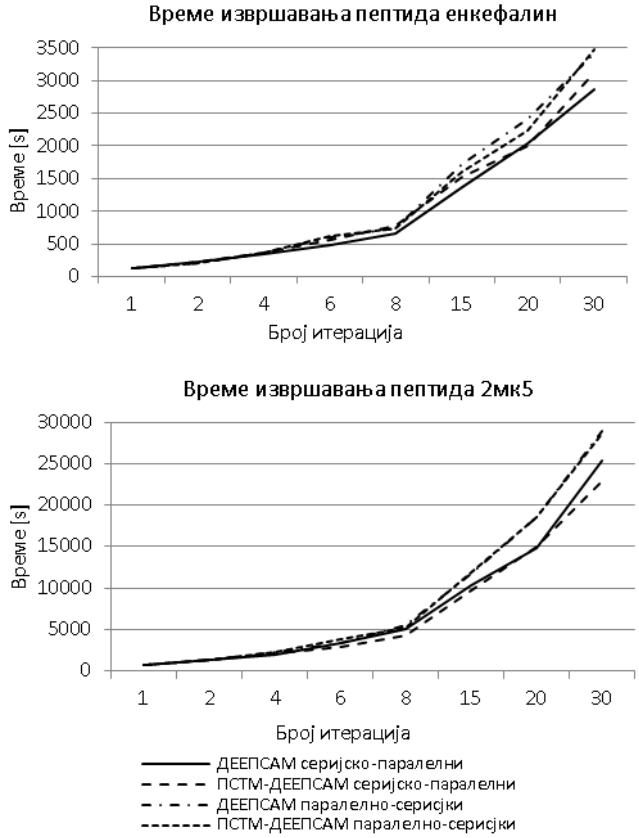
За сваки пептид извршене су по 1, 2, 4, 6, 8, 15, 20 и 30 итерација еволутивног алгоритма. Већи број итерација је помогао да се стекне увид у динамику извршавања и понашања система у случају дугих обрада. За сваки број итерација, обе ДЕЕПСАМ верзије су извршене три пута. Експерименти су спроведени под 64-битним оперативним системом Убунту и на рачунару опремљеном процесором Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz 3.90GHz, са 16 GB оперативне меморије и магнетним HDD.

8.3.2 Експериментални резултати и дискусија

Резултати трајања обраде пептида енкефалин и 2мк5 дати су на слици 11. Резултати су веома слични и упоредиви за оба пептида и за све итерације. Уопштено, обе ДЕЕПСАМ верзије показују слично понашање које је потврђено малим разликама времена обраде. Иако је понекад нека од имплементација била боља од оне друге, резултати су у очекиваним оквирима. Резултати указују на чињеницу да обе имплементације имају упоредну динамику извршавања, тј. примена ПСТМ није проузроковала никакве споредне ефекте у оригиналној ДЕЕПСАМ имплементацији.

Резултати приказани на слици 11 откривају спорадичне сметње у извршавању које се појављују током спровођења експеримената. Идеално би било да се програми извршавају изоловано и без интерференције са оперативним системом и другим програмима, али то у стварности није изводљиво. Према

томе, ове сметње се манифестују због ометања извршавања програма од стране других позадинских процеса које није могуће контролисати нити спречити.



Слика 11 – Резултати времена обраде пептида енкефалин и 2мк5

Узимајући у обзир пептид 2мк5 као тест скалабилности, резултати показују предност решења заснованог на ПСТМ. У том конкретном случају, имплементација програма ДЕЕПСАМ заснована на ПСТМ је показала добру скалабилност у односу на искоришћавање доступних језгара. За оба пептида резултати времена трајања обраде у паралелно-серијском режиму су мало већа него у серијски-паралелном режиму. Дужа трајања времена обраде су проузрокована споредним ефектима механизма упошљеног чекања који се користи за синхронизацију Фортран процеса. Наиме, синхронизација је имплементирана као бесконачна петља, која на тај начин стално окупира једно од могућих језгара, што имплицира дуже укупно време извршавања за све процесе који учествују у обради. Треба имати на уму да су времена извршавања за 2мк5 за један ред величине већа од времена извршавања за енкефалин.

За потребе анализе израчуната је и релативна грешка времена обраде – слика 12. Релативна грешка је рачуната као однос између две вредности, *x* и *y*, и њихове просечне вредности: $x - y/((x + y)/2)$. Вредности x и y представљају време трајања обраде оригиналне и ПСТМ-засноване ДЕЕПСАМ верзије.

Вредности релативне грешке се могу посматрати као вредности случајне промењљиве. Наиме, вредности релативне грешке су расуте у распону од минималних 0,14% до максималних 15,22%. Такође, након примене стандардне девијације на средњем узорку, једна од три измерене вредности је елиминисана, тј. преостале су само две измерене вредности које су коришћене за израчунавање просечних вредности. Мањи број измерених вредности директно утиче на расипање вредности релативне грешке, тако да се вредности релативне грешке могу наћи између ~ 1% и ~ 15%.

Слика 12 – График релативне грешке времена обраде пептида енкефалин и 2мк5

Са повећањем броја итерација еволутивног алгоритма (дуже време трајања обраде) вредности релативне грешке остају исте, што је доказ да верзија

заснована на ПСТМ нема ограничења и уска грла проузрокована решењем архитектуре. Средња релативна грешка у најгорем случају износи ~ 8%, а у најбољем случају ~ 3%. За потребе ове студије, граница грешке је била 10%, све испод тога је сматрано прихватљивим. Просечни резултати релативне грешке дати су у табели 2.

Табела 2 – Просечне вредности релативне грешке пептида енкефалина и 2мк5

Начин извршавања	серијски-паралелни / паралелни-серијски	
Пептид	Енкефалин	2мк5
Просечна релативна грешка [%]	7,7 / 3,07	6,5 / 3,38

8.4 Закључак студије

У овој студији представљена је интеграција (П)СТМ у један сигурносно-осетљив систем, као што је програм за предвиђање структуре протеина ДЕЕПСАМ који се користи у фармацеутској индустрији. Почевши од оригиналне ДЕЕПСАМ верзије развијена је нова ДЕЕПСАМ архитектура заснована на ПСТМ. Описани систем представља потврду концепта применљивости ПСТМ у реалним и сложеним програмима.

Представљени резултати нису открили никакав систематски проблем или ограничење нове ДЕЕПСАМ архитекуре засноване на ПСТМ. Такође, у току извршавања програма нису уочене никакве неправилности. Овакви резултати су охрабрујући и стимулативни, посебно када се ПСТМ посматра као потенцијално средство за будућа унапређења. Узимајући у обзир и резултате тестирања и валидације, закључује се да је формално верификована исправност ПСТМ и експериментално потврђена.

Пошто оригинална ДЕЕПСАМ верзија већ подржава паралелно извршавање, није се очекивало значајно побољшање перформанси, као што би то био случај да је као референтна верзија за поређење била коришћена секвенцијална имплементација. Ипак, да би се спознао стварни утицај ПСТМ на перформансе времена извршења, потребно је спровести евалуацију на рачунару са много језгара који је у стању да покреће десетине, или чак стотине процеса

истовремено. Очекивано је да ће у том случају потенцијал ПСТМ доћи више до изражаја.

ПОГЛАВЉЕ 9.

ЗАКЉУЧАК И БУДУЋИ РАД

Основни циљ истраживања обухваћеног дисертацијом била је формална верификација софтверске трансакционе меморије, односно примена формализма временских аутомата за моделовање, које се заснива на детаљима архитектуре и имплементације користећи изворни код решења, а потом и формална верификација исправности која је спроведена машинским (аутоматским) путем употребом алата за проверу исправности модела. За моделовање и верификацију модела коришћен је алат УППААЛ. Формализација и верификација исправности представљени су на примеру конкретне имплементације софтверске трансакционе меморије за програмски језик Пајтон, названом Пајтон Софтверска Трансакциона Меморија, односно ПСТМ. Такође, по најбољем сазнању аутора, ПСТМ представља прву формално верификовану софтверску трансакциону меморију за програмски језик Пајтон.

Формализација софтверске трансакционе меморије је први корак у поступку верификације који служи као поступак за прављење верног модела система, без изостављања детаља имплементације. Други корак чини провера исправности жељених својстава система на основу направљеног модела, односно провера својстава сигурности, животности, достижности, као и провера међусобног блокирања. ПСТМ систем је верификован у најгорем сценарију извршавања у којем све трансакције деле исте податке. Резултати верификације

показују да ПСТМ систем задовољава сва од претходно наведених својстава. Поред формалне провере, исправност ПСТМ је тестирана и потврђена експерименталним путем употребом у оквиру реалне апликације за прорачун и симулацију хемијских модела структуре протеина ДЕЕПСАМ, у којој је ПСТМ допринео остваривању бољих перформанси извршавања, без уношења системских ограничења.

Резултати представљени у овој докторској дисертацији могу се користити како за академска, тако и у пољу индустријских истраживања. Резултати се могу посматрати у два аспекта. Са једне стране, демонстрирана је применљивост приступа формалној верификацији софтверских трансакционих меморија употребом временских аутомата на примеру конкретне СТМ за програмски језик Пајтон, ПСТМ. Са друге стране, управо као артефакт тог процеса, остаје формално верификовано решење СТМ за програмски језик Пајтон, које до сада није постојало, и које је експериментално валидирано у реалним условима користећи програм ДЕЕПСАМ. С обзиром на то да је у данашње време Пајтон један од најчешће коришћених програмских језика, потенцијал применљивости овакве компоненте је веома висок.

Током процеса верификације уочена су два потенцијална ограничења. Једно од потенцијалних ограничења се односи на уведене претпоставке трајања локалне функције обраде трансакције, односно трајања операција читања и писања трансакционе меморије, које у општем случају не морају бити једнаке. Друго потенцијално ограничење је број трансакција који је коришћен у процесу верификације, а који у основи указује на други проблем, а то је брзина увећања простора стања које алат треба да провери како би утврдио истинитост одређеног својства. Узрок овога се налази у самом алату УППААЛ који је веома осетљив на употребу временских инваријанти. Оба потенцијална ограничења су отклоњена адекватном параметризацијом верификационих модела. На крају, треба напоменути да ниједно од ова два потенцијална ограничења ни на који начин не утиче на представљене резултате нити изведене закључке.

На основу претходно спроведене опсежне анализе и представљених резултата може се закључити да је првобитно дефинисана хипотеза потврђена.

Будуће истраживање и правци даљег рада усмерени су ка изради формално-верификованих прототипова паралелних дистрибуираних инфраструктура које су засноване на трансакционим системима, такође за програмски језик Пајтон. Ове арихтектуре могу бити коришћене за општенаменске обраде са меким временским ограничењима у облаку, као и у све заступљенијим ИоТ апликацијама. Истраживање обухвата развој дистрибуираних архитектура и дистрибуираних програма за мерење перформанси, које за програмски језик Пајтон такође не постоје. У истраживању се планира развој две врсте модела, УМЛ модели и КлаудСим модели. За формалну верификацију планирају бити коришћени формализам временских аутомата као и формализам ЦСП.

Литература

[1] M. Herlihy, J. E. B. Moss: “Transactional Memory: Architectural Support for Lock-Free Data Structures”. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA’93)*, pp. 289-300, 1993.

[2] T. Harris, J. R. Larus, R. Rajwar: “Transactional Memory”, 2nd edition. Morgan and Claypool, 2010.

[3] R. Guerraoui, M. Kapalka: “Principles of Transactional Memory”, *Synthesis Lectures on Distributed Computing Theory*. Morgan and Claypool, 2010.

[4] N. Shavit, D. Touitou: “Software Transactional Memory”. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC’95)*, pp. 204-213, 1995.

[5] https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-e3-1200v3-spec-update.pdf (приступљено 17.08.2019.)

[6] http://doc.pypy.org/en/latest/introduction.html (приступљено 17.08.2019.)

[7] A. Cohen, J. W. O’Leary, A. Pnueli, M. R. Tuttle, L. D. Zuck: “Verifying Correctness of Transactional Memories”. In *Proceedings of the 7th International Conference on Formal Methods in Computer - Aided Design (FMCAD 2007)*, pp. 37-44, 2007.

[8] M. Emmi, R. Majumdar, R. Manevich: “Parameterized Verification of Transactional Memories”. In *Proceedings of the 31st Conference on Programming Language Design and Implementation (PLDI’10)*, pp. 134-145, 2010.

[9] R. Guerraoui, T. A. Henzinger, V. Singh: “Model checking transactional memories”. In *Distributed computing*, Vol. 22 (3), pp. 129-145, 2010.

[10] S. Doherty, L. Groves, V. Luchangco, M. Moir: “Towards formally specifying and verifying transactional memory”. In *Formal Aspects of Computing (FAOC)*, Vol. 25 (5), pp. 769-799, 2013.

[11] M. Lesani, V. Luchangco, M. Moir: “A Framework for Formally Verifying Software Transactional Memory Algorithms”. In *Lecture Notes in Computer Science*, Vol. 7454, pp. 516-530, 2012.

[12] V. Bushkov, R. Guerraoui: “Liveness in Transactional Memory”. In *Lecture Notes in Computer Science*, Vol. 8913, pp. 32-49, 2015.

[13] M. Goldstein, E. Fredj, R. B. Gerber: “A new hybrid algorithm for finding the lowest minima of potential surfaces: Approach and application to peptides”. In *Journal of Computational Chemistry*, Vol. 32 (9) , pp. 1785-1800, 2011.

[14] M Amitay, M Goldstein: “Evaluating the peptide structure prediction capabilities of a purely ab-initio method”. In *Protein Engineering, Design and Selection*, Vol. 30 (10), pp. 723-727, 2017.

[15] M. Popovic, B. Kordic: “PSTM: Python Software Transactional Memory”. In *22nd Telecommunications Forum Telfor (TELFOR 2014)*, pp. 1106-1109, 2014.

[16] B. Kordic, M. Popovic, I. Basicevic: “DPM-PSTM: Dual-Port Memory Based Python Software Transactional Memory”. In *4th Eastern European Regional Conference on the Engineering of Computer Based Systems (ECBS-EERC)*, pp. 126-129, 2015.

[17] H. Mills, M. Dyer, R. Linger: “Cleanroom Software Engineering”. In *IEEE Software*, Vol. 4 (5), pp. 19-25, 1987.

[18] G. Bahig, A. El-Kadi: “Formal Verification of Automotive Design in Compliance with ISO 26262 Design Verification Guidelines”. In *IEEE Access*, Vol. 5, pp. 4505-4516, 2017.

[19] http://www.uppaal.org (приступљено 17.08.2019.)

[20] G. Behrmann, A. David, K. G. Larsen: “A tutorial on Uppaal”. In *Lecture Notes in Computer Science*, Vol. 3185, pp. 200-236, 2004.

[21] B. Kordic, M. Popovic, S. Ghilezan, I. Basicevic: “An Approach to Formal Verification of Python Software Transactional Memory”. In *Proceedings of the Fifth Conference on the Engineering of Computer-Based Systems (ECBS’17)*, pp. 1-10, 2017.

[22] B. Kordic, M. Popovic, S. Ghilezan: “Formal Verification of Python Software Transactional Memory Based on Timed Automata”. In *Acta Polytechnica Hungarica Journal of Applied Science (ACTA)*, Vol. 16 (7), pp. 197-216, 2019.

[23] B. Kordic, M. Popovic, M. Popovic, M. Goldstein, M. Amitay, D. Dayan: “A Protein Structure Prediction Program Architecture Based on a Software Transactional Memory”. In *Proceedings of the Sixth Conference on the*

Engineering of Computer-Based Systems (ECBS'19), pp.1-9, 2019.

[24] A. Cohen, A. Pnueli, L. D. Zuck: "Mechanical Verification of Transactional Memories with Non-Transactional Memory Accesses". In *Proceedings of the 20th International Conference on Computer Aided Verification (CAV'08)*, pp. 121-134, 2008.

[25] R. Guerraoui, M. Kapałka: "On the Correctness of Transactional Memory". In *Proceedings of the 13th Symposium on Principles and Practice of Parallel Programming (PPoPP'08)*, pp. 175-184, 2008.

[26] R. Guerraoui, T. A. Henzinger, V. Singh: "Completeness and Nondeterminism in Model Checking Transactional Memories". In *Proceedings of the 19th International Conference on Concurrency Theory (CONCUR 2008)*, pp. 21-35, 2008.

[27] S. Doherty, L. Groves, V. Luchangco, M. Moir: "Towards Formally Specifying and Verifying Transactional Memory". In *Electronic Notes in Theoretical Computer Science (ENTCS)*, Vol. 259, pp. 245-261, 2009.

[28] A. El-kustaban, B. Moszkowski, A. Cau: "Formalising of Transactional Memory Using Interval Temporal Logic (ITL). In *2012 Spring Congress on Engineering and Technology (S-CET)*, pp. 1-6, 2012.

[29] P. Abdulla, S. Dwarkadas, A. Rezine, A. Shriraman, Y. Zhu: "Verifying safety and liveness for the FlexTM hybrid transactional memory". In *Proceedings of Design, Automation Test in Europe (DATE'13)*, pp. 785-790, 2013.

[30] C. C. Minh, J. Chung, C. Kozyrakis, K. Olukotun: "STAMP: Stanford Transactional Applications for Multi-Processing". In *2008 IEEE International Symposium on Workload Characterization*, pp. 35-46, 2008.

[31] R. Guerraoui, M. Kapalka, J. Vitek: "STMBench7: A Benchmark for Software Transactional Memory". In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pp. 315-324, 2007.

[32] F. Zyulkyarov, V. Gajinov, O. S. Unsal, A. Cristal, E. Ayguade, T. Harris, M. Valero: "Atomic Quake: Using Transactional Memory in an Interactive Multiplayer Game Server". In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 25-34, 2009.

[33] F. Zyulkyarov, V. Gajinov, O. S. Unsal, A. Cristal, E. Ayguade, T. Harris, M. Valero: “QuakeTM: Parallelizing a Complex Sequential Application Using Transactional Memory”. In *Proceedings of the 23rd International Conference on Supercomputing*, pp. 126-135, 2009.

[34] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, B. Hertzberg: “McRT-STM: A High Performance Software Transactional Memory System for a Multi-Core Runtime”. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming - PPoPP ’06*, pp. 187-197, 2006.

[35] T. Nakaike, R. Odaira, T. Nakatani, M. M. Michael: “Real Java Applications in Software Transactional Memory”. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC’10)*, pp. 1-10, 2010.

[36] O. S. Hofmann, D. E. Porter, E. Witchel, C. J. Rossbach, H. E. Ramadan, A. Bhandari: “MetaTM/TxLinux: Transactional Memory for an Operating System”. In *IEEE Micro*, Vol. 28 (1), pp. 42-51, 2008.

[37] P. Lu, D. R. Bevan, A. Leber, R. Hontecillas, N. uria Tubau-Juni, J. Bassaganya-Riera: “Computer-Aided Drug Discovery”. In *Accelerated Path to Cures*, pp. 7-24, 2018.

[38] K. Reinert, B. Langmead, D. Weese, DJ. Evers: “Alignment of Next-Generation Sequencing Reads”. In *Annual Review of Genomics and Human Genetics*, Vol. 16 (1), pp. 133-151, 2015.

[39] B. D. Weitzner, J. R. Jeliazkov, S. Lyskov, N. Marze, D. Kuroda, R. Frick, J. Adolf-Bryfogle, N. Biswas, R. L. Jr. Dunbrack, J. J. Gray: “Modeling and docking of antibody structures with Rosetta,”. In *Nature Protocols*, Vol. 12 (2), pp. 401-416, 2017.

[40] H. A. Scheraga, J. Lee, J. Pillardy, Y. J. Ye, A. Liwo, D. Ripoll: “Surmounting the Multiple-Minima Problem in Protein Folding”. In *Journal of Global Optimization*, Vol. 15 (3), pp. 235-260, 1999.

[41] A. E. Eiben, J. E. Smith: “Introduction to evolutionary computing”. Corrected. Berlin Heidelberg: Springer, 2007.

[42] T. Back, D. B. Fogel, T. Michalewicz (Eds.): “Evolutionary Computation 1: Basic Algorithms and Operations”. IOP Publishing Ltd. UK, 2000.

[43] L. J. Fogel, A. J. Owens, M. J. Walsh: “Artificial Intelligence through Simulated Evolution”. Wiley, USA, 1966.

[44] E. Fredj, M. Goldstein: “A Knowledge-Based Approach to Initial Population Generation in Evolutionary Algorithms : Application to the Protein Structure Prediction Problem”. In *Lecture Notes in Computer Science*, Vol. 8001, pp. 252-262, 2014.

[45] J. W. L. Ponder:“TINKER Molecular Modelling Package”. 2003. https://dasher.wustl.edu/tinker/ (приступљено 17.08.2019.)

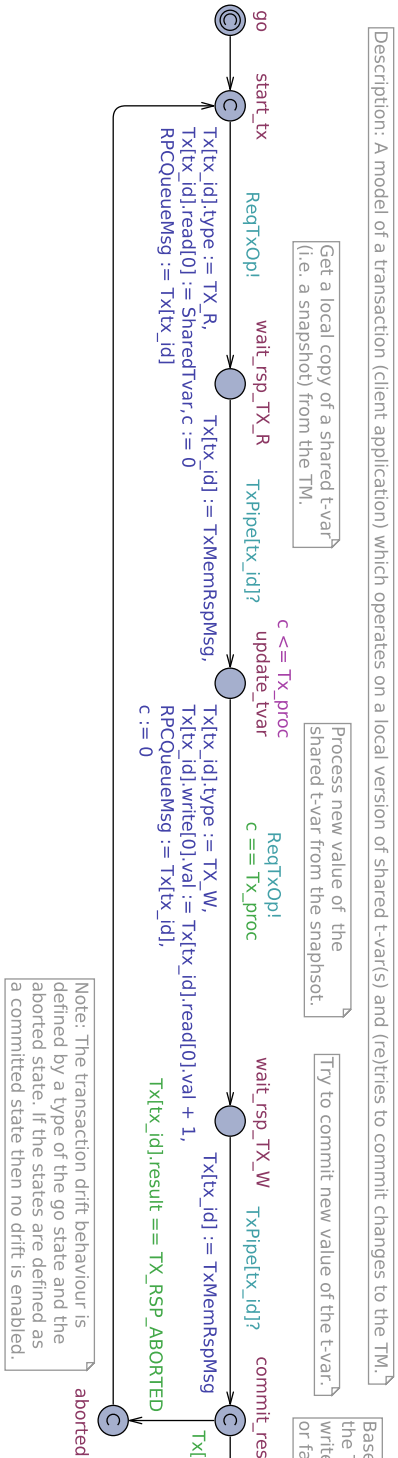
[46] J. Kostrowicki, H. A. Scheraga: “Application of the Diffusion Equation Method for Global Optimization to Oligopeptides”. In *The Journal of Physical Chemistry*, Vol. 96 (18), pp. 7442-7449, 1992.

[47] S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi: “Optimization by Simulated Annealing”. In *Science, New Series*, Vol. 220 (4598), pp. 671-80, 1983.

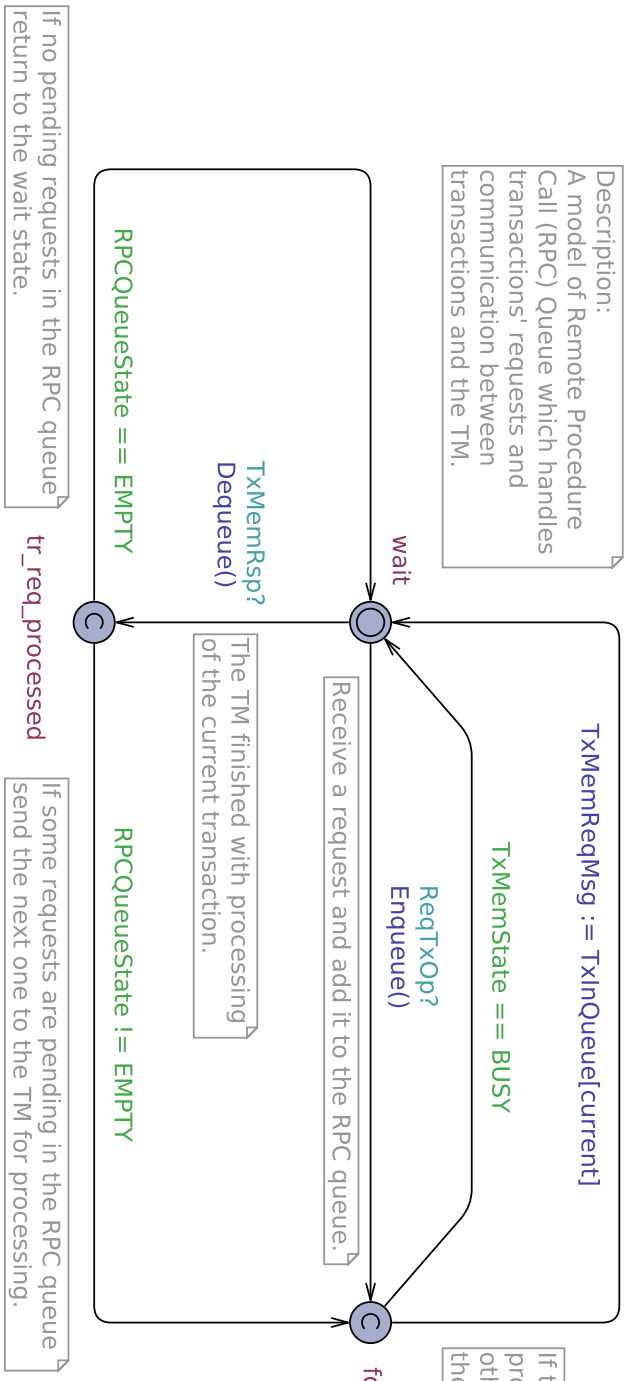
[48] D. C. Liu, J. Nocedal: “On the Limited Memory BFGS Method for Large Scale Optimization,” In *Mathematical Programming*, Vol 45 (1-3), pp. 503-528, 1989.

Прилог А

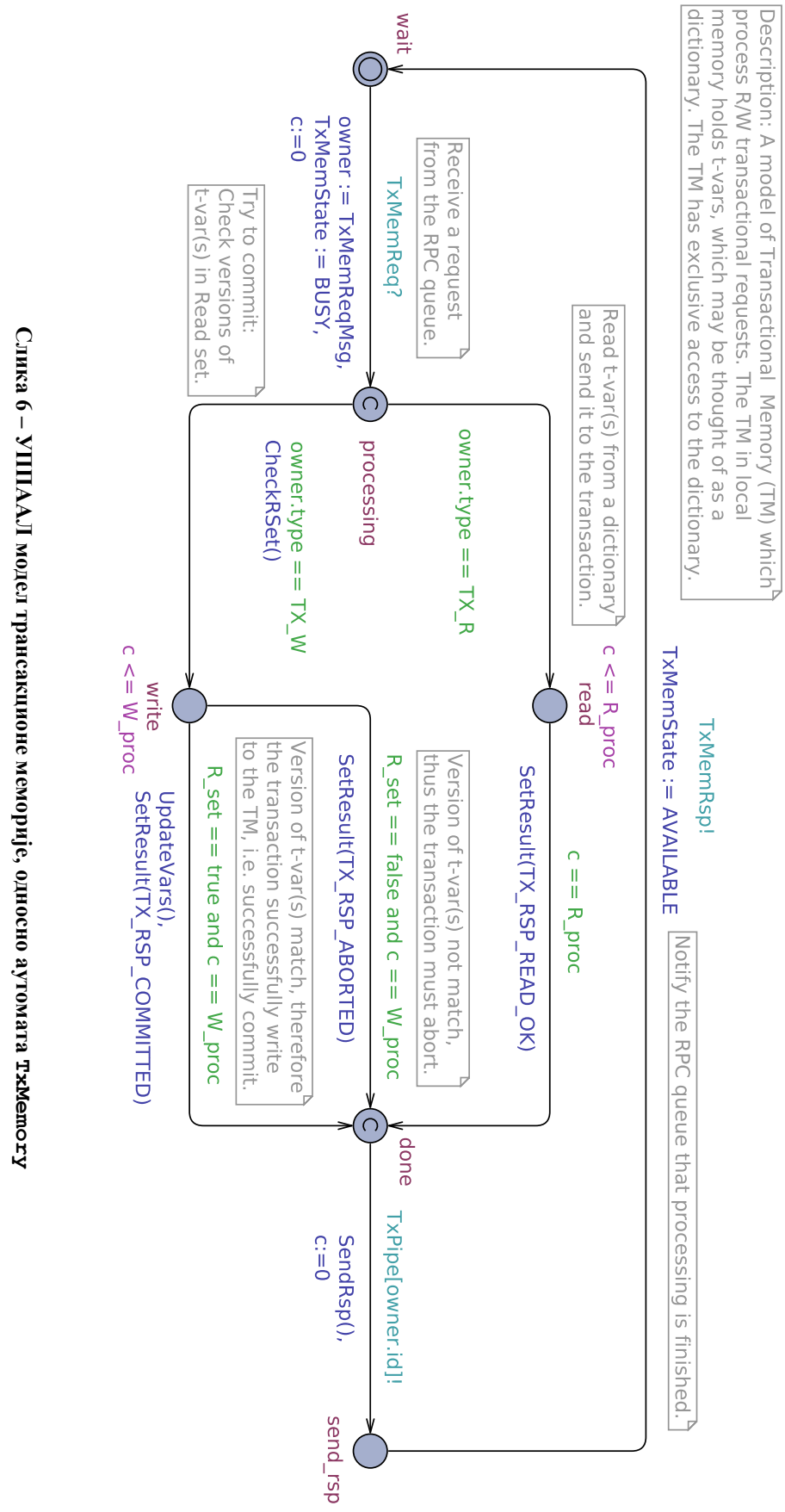
У овом прилогу дате су слике структуре УППААЛ аутомата *Трансакција*, *Ред чекања* и *Трансакциона меморија* у већем формату, а које су претходно приказане на сликама 4, 5 и 6, респективно.



Слика 4 – УППААЛ модел трансакције, односно аутомата `Transaction`



Слика 5 – УППАА.Л модел реда чекања, односно аутомата RPCQueue



Слика 6 – УППААЛ модел трансакционе меморије, односно аутомата `TxMemory`

Прилог Б

У табелама 2 до табеле 7 приложени су детаљни резултати, статистика и перформансе извршавања верификације за својства *Непостојање међусобног блокирања*, *Сигурност I*, *Сигурност II*, *Животност I*, *Животност II*, *Животност III* и *Достижност*, респективно. Свака табела садржи детаље о верификованом својству (Својство), броју трансакција коришћених у спроведеној верификацији (Број трансакција), типу трансакција (Тип трансакције), времену трајања верификације (Време), броју претражених стања (Број стања) и резултату верификације (Резултат). Поред њих, у табели 7 се налазе и детаљи о броју извршених трансакција (Број завршених трансакција). Време извршавања је заокружено на милисекунде. Верификације које су успешно оконачне и чији је исход потврдио својство означене су са ознаком Т. Верификације код којих је број трансакција превазишао капацитет радне меморије, тј. оне које нису биле завршене означене су као неодлучене (ознака -).

Табела 3 – Резултати верификације својства *Непостојање међусобног блокирања*

Својство	**Број трансакција**	**Тип трансакције**	**Време**	**Број стања**	**Резултат**
Непостојање међусобног блокирања	1	Линеарна померена	1ms	17	T
	2		1ms	181	T
	3		10ms	2 071	T
	4		150ms	28 313	T
	5		2s 750ms	466 741	T
	6		1m 2s 110ms	9 045 757	T
	7		-	-	-
Непостојање међусобног блокирања	1	Циклична померена	1ms	17	T
	2		1ms	221	T
	3		40ms	4 747	T
	4		780ms	158 129	T
	5		1m 2s 440ms	10 140 401	T
	6		-	-	-
Непостојање међусобног	1	Линеарна поравната	1ms	17	T
	2		1ms	100	T

блокирања	3		1ms	482	T
	4		30ms	2 528	T
	5		90ms	15 512	T
	6		650ms	109 888	T
	7		5s 990ms	885 712	T
	8		1m 2s 90ms	8 014 336	T
	9		-	-	-
Непостојање међусобног блокирања	1	Циклична поравната	1ms	17	T
	2		1ms	134	T
	3		1ms	878	T
	4		50ms	6 008	T
	5		330ms	45 512	T
	6		2s 380ms	384 208	T
	7		24s 930ms	3 597 232	T
	8		-	-	-

Табела 4 – Резултати верификације својства *Сигурност I*

Својство	**Број трансакција**	**Тип трансакције**	**Време**	**Број стања**	**Резултат**
Сигурност I	1	Линеарна померена	1ms	17	T
	2		1ms	181	T
	3		10ms	2 071	T
	4		70ms	28 313	T
	5		1s 450ms	466 741	T
	6		32s 180ms	9 045 757	T
	7		-	-	-
Сигурност I	1	Циклична померена	1ms	17	T
	2		1ms	221	T
	3		20ms	4 747	T
	4		430ms	158 129	T
	5		34s 670ms	10 140 401	T
	6		-	-	-
Сигурност I	1	Линеарна поравната	1ms	17	T
	2		1ms	100	T
	3		1ms	482	T
	4		20ms	2 528	T

	5		70ms	15 512	T
	6		400ms	109 888	T
	7		3s 470ms	885 712	T
	8		34s 790ms	8 014 336	T
	9		-	-	-
Сигурност I	1	Циклична поравната	1ms	17	T
	2		1ms	134	T
	3		10ms	878	T
	4		30ms	6 008	T
	5		130ms	45 512	T
	6		1s 250ms	384 208	T
	7		12s 950ms	3 597 232	T
	8		-	-	-

Табела 5 – Резултати верификације својства *Сигурност II*

Својство	**Број трансакција**	**Тип трансакције**	**Време**	**Број стања**	**Резултат**
Сигурност II	1	Линеарна поравната	1ms	17	T
	2		1ms	100	T
	3		1ms	482	T
	4		20ms	2 528	T
	5		50ms	15 512	T
	6		440ms	109 888	T
	7		3s 700ms	885 712	T
	8		38s 740ms	8 014 336	T
	9		-	-	-

Табела 6 – Резултати верификације својства *Животност I*

Својство	**Број трансакција**	**Тип трансакције**	**Време**	**Број стања**	**Резултат**
Животност I	1	Циклична померена	1ms	17	T
	2		1ms	255	T
	3		30ms	6 703	T
	4		760ms	253 861	T
	5		1m 0s 20ms	17 489 881	T

	6		-	-	-
Животност I	1	Циклична поравната	1ms	17	T
	2		1ms	134	T
	3		1ms	878	T
	4		30ms	6 008	T
	5		170ms	45 512	T
	6		1s 290ms	384 208	T
	7		13s 450ms	3 597 232	T
	8		-	-	-

Табела 7 – Резултати верификације својства *Животност II*

Својство	**Број трансакција**	**Тип трансакције**	**Време**	**Број стања**	**Резултат**
Животност II	1	Циклична померена	1ms	14	T
	2		1ms	167	T
	3		20ms	2 397	T
	4		200ms	53 641	T
	5		5s 510ms	1 690 633	T
	6		-	-	-
Животност II	1	Циклична поравната	1ms	14	T
	2		1ms	127	T
	3		1ms	855	T
	4		30ms	5 913	T
	5		180ms	45 033	T
	6		1s 200ms	381 329	T
	7		12s 420ms	3 577 073	T
	8		-	-	-

Табела 8 – Резултати верификације својства *Животност III*

Својство	**Број трансакција**	**Тип трансакције**	**Број завршених трансакција**	**Време**	**Број стања**	**Резултат**
Животност III	1	Циклична поравната	1	1ms	14	T
Животност III	2	Циклична поравната	1	1ms	65	T
			2	1ms	127	T

Животност III	3	Циклична поравната	1	1ms	321	T
			2	1ms	669	T
			3	10ms	855	T
Животност III	4	Циклична поравната	1	10ms	1 713	T
			2	10ms	3 777	T
			3	10ms	5 169	T
			4	20ms	5 913	T
Животност III	5	Циклична поравната	1	50ms	10 593	T
			2	80ms	24 033	T
			3	110ms	34 353	T
			4	120ms	41 313	T
			5	130ms	45 033	T
Животност III	6	Циклична поравната	1	270ms	75 329	T
			2	570ms	174 689	T
			3	850ms	255 329	T
			4	1s 030ms	317 249	T
			5	1s 200ms	359 009	T
			6	1s 250ms	381 329	T
Животност III	7	Циклична поравната	1	2s 470ms	608 513	T
			2	5s 550ms	1 435 073	T
			3	8s 150ms	2 130 593	T
			4	10s 310ms	2 695 073	T
			5	11s 880ms	3 128 513	T
			6	12s 930ms	3 420 833	T
			7	13s 680ms	3 577 073	T
Животност III	8	Циклична поравната	1	25s 780ms	5 514 497	T
			2	58s 740ms	13 175 297	T
			3	-	-	-
			4	-	-	-
			5	-	-	-
			6	-	-	-
			7	-	-	-
			8	-	-	-

Табела 9 – Резултати верификације својства *Достижност*

Својство	Број трансакција	Тип трансакције	Време	Број стања	Резултат
Достижност	1	Линеарна померена	1ms	19	T
	2		1ms	189	T
	3		10ms	2 131	T
	4		100ms	28 985	T
	5		1s 480ms	475 621	T
	6		32s 500ms	9 186 157	T
	7		-	-	-
Достижност	1	Циклична померена	1ms	19	T
	2		1ms	255	T
	3		30ms	5 965	T
	4		650ms	205 385	T
	5		52s 620ms	14 008 901	T
	6		-	-	-
Достижност	1	Линеарна поравната	1ms	19	T
	2		1ms	104	T
	3		1ms	494	T
	4		10ms	2576	T
	5		80ms	15 752	T
	6		400ms	111 328	T
	7		3s 350ms	895 792	T
	8		34s 650ms	8 094 976	T
	9		-	-	-
Достижност	1	Циклична поравната	1ms	19	T
	2		1ms	160	T
	3		1ms	956	T
	4		30ms	6 320	T
	5		190ms	47 072	T
	6		1s 290ms	393 568	T
	7		13s 500ms	3 662 752	T
	8		-	-	-