УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА

Илија Купрешанин

ГЕНЕРИЧКИ МЕТОД ЗА СТАТИСТИЧКО ТЕСТИРАЊЕ ПАРАЛЕЛНИХ ПРОГРАМА БАЗИРАНИХ НА СТАБЛУ ЗАДАТАКА

ДОКТОРСКА ДИСЕРТАЦИЈА

Нови Сад, 2012

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 публикација
Тип записа, **ТЗ**:	Текстуални штампани материјал
Врста рада, **ВР**:	Докторска дисертација
Аутор, **АУ**:	Мр Илија Купрешанин
Ментор, **МН**:	Проф. др Мирослав Поповић
Наслов рада, **НР**:	ГЕНЕРИЧКИ МЕТОД ЗА СТАТИСТИЧКО ТЕСТИТАЊЕ ПАРАЛЕЛНИХ ПРОГРАМА БАЗИРАНИХ НА СТАБЛУ ЗАДАТАКА
Језик публикације, **ЈП**:	Српски
Језик извода, **ЈИ**:	Српски
Земља публиковања, **ЗП**:	Република Србија
Уже географско подручје, **УГП**:	Војводина
Година, **ГО**:	2012.
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)	9/46/35/3/6/3/0
Научна област, **НО**:	Електротехника и рачунарство
Научна дисциплина, **НД**:	Рачунарска техника и рачунарске комуникације
Предметна одредница/Кључне речи, **ПО**:	Вишепроцесорски, паралелно програмирање, паралелни програми, структурно тестирање, статистичко тестирање
УДК	
Чува се, **ЧУ**:	Библиотека Факултета техничких наука, Нови Сад
Важна напомена, **ВН**:	
Извод, **ИЗ**:	Теза се бави посебном класом паралелних програма који су базирани на стаблу задатака. Предмет рада је прилагођавање генеричког метода за статистичко тестирање секвенционалних програма за ову класу паралелних програма. У раду се упоређује овако прилагођен генерички метод са прилагођеном методом потпуног тестирања и претходно прилагођеном статистичком методом тестирања, базираном на експерименталном мерењу времена извршења тестних случајева и покривености путева преко стабла задатака.
Датум прихватања теме, **ДП**:	30.05.2012.
Датум одбране, **ДО**:	
Чланови комисије, **КО**: Председник:	др Владимир Ковачевић, проф. емеритус
Члан:	др Јован Ђорђевић, ред. проф.
Члан:	др Никола Теслић, ред. проф.
Члан:	др Илија Башичевић, доцент
Члан, ментор:	др Мирослав Поповић, ред. проф.

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**:	Ilija Kupresanin
Mentor, **MN**:	PhD Miroslav Popovic
Title, **TI**:	GENERIC METHOD FOR STATISTICAL TESTING OF PARALLEL PROGRAMS BASED ON TASK TREES
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**:	2012.
Publisher, **PB**:	Author's reprint
Publication place, **PP**:	Serbia, Novi Sad, Trg Dositeja Obradovica 6
Physical description, **PD**: (chapters/pages/ref./tables/pictures/graphs/appendixes)	9/46/35/3/6/3/0
Scientific field, **SF**:	Electrical Engineering
Scientific discipline, **SD**:	Computer Engineering and Computer Communication
Subject/Key words, **S/KW**:	Multicores, parallel programming, parallel programs, structural testing, statistical testing
UC	
Holding data, **HD**:	The Library of Faculty of Tehnical Sciences, Novi Sad
Note, **N**:	
Abstract, **AB**:	
Accepted by the Scientific Board on, **ASB**:	30th May 2012.
Defended on, **DE**:	

Defended Board, **DB**:		
President:	PhD Vladimir Kovacevic	
Member:	PhD Jovan Dordevic	
Member:	PhD Nikola Teslic	Menthor's sign
Member:	PhD Ilija Basicevic	
Member, Mentor:	PhD Miroslav Popovic	

С А Д Р Ж А Ј

УВОД

Системи са којима се данас сусрећемо, углавном су базирани на рачунару. Не може се замислити ни једна од области људског деловања без помоћи одговарајућих програмских решења. Примери су многобројни: различити ручни уређаји, кућни апарати, аутомобили, авиони, индустријске машине, развијени информациони системи, итд. Као резултат напретка технологије постоји континуалан раст процесне снаге, што омогућава развој све сложенијих система, тако да данас имамо веома сложене системе, који у себи садрже више различитих система. Заједно са растом система, расте и потреба за унапређењем поузданости. Поузданост није више само интерес оних који развијају и користе програмску подршку, већ је он и интерес опште јавности. Да би се обезбедио одговарајући квалитет програмске подршке, нису само важни модели за процену поузданости већ и практично питање израчунавање ових процена. Ово захтева нове методе, приступе и технике тестирања за веома широку класу уграђених система.

Тестирање је процес који је концентрисан на проналажење грешака у систему, представља основни елеменат развоја система и чини основу унапређења квалитета система. Генерално, тестирање програмске подршке састоји се од избора тестних података, извршавања теста и верификације излаза, односно поређење понашања система са очекиваним. Различити приступи у избору тестних података могу се сврстати у три групе: структурне или метода беле кутије (white-box) које врше избор тестних података на основу детаља унутрашње структуре програма, функционалне методе или методе црне кутије (black-box) који бирају тестне податке на основу спецификације програма и статистиче методе које бирају тестне податке случајно, према расподели, често преко улазног домена. Методе и технике статистичког тестирања не могу бити ефикасно коришћене без одговарајућих алата за генерисање великог броја тестних случајева које захтева статистичка процена.

Развој процесора са више језгара као и кластера ових процесора који омогућавају паралелно извршавање програма постављају нови изазов - развој паралелних програма састављених од слободно спојених или потпуно независних процеса. Стабло задатака је врста паралелног програма где су изведени чворови од истог родитељског чвора међусобно независни. Паралелна обрада података један је од кључних питања који мора бити решен код пројектовања програмских решења за управљање критичним инфраструктурама као што су дистрибуциони системи за гас и електричну енергију. Ови системи обезбеђују континуалан надзор и контролу базирану на прикупљању и обради података. Од њих се очекује веома велика расположивост, поузданост и сигурност, као и да подржавају различите оперативне активности. Они обухватају веома сложене скупове сервисних компоненти које омогућавају веома сложене прорачуне, коришћењем различитих модела система, који уобичајено имају форму стабла. Пројектовање ове врсте сервисних компоненти је веома захтеван, јер се ове врсте прорачуна врше на веома великим моделима у реалном времену.

Овај рад се бави посебном класом паралелних програма базираних на стаблу задатака које је названо TTE (TaskTreeExecutor) стаблом задатака, види (Поповић, 2009). TTE стабло задатака је стабло задатака код којег су изведени чворови од истог родитељског чвора међусобно независни. Пример програма базираног на стаблу задатака је систем велике скале за управљање дистрибуцијом електричне енергије (Поповић, 2009). Концептуално, TTE стабло задатака је веома слично са Intel Cilk Plus графовима без циклуса (Intel, 2011 a) и Intel Thread Building Blocks графова задатака (Intel, 2011 b), који су основа последњег Intel Parallel Studio 2011 који је укључен у Microsoft Visual Studio развојно окружење.

Тестирање и верификација паралелних програма базираних на стаблу задатака је нови изазов за истраживаче. Постојећи, добро афирмисани методи и пратећи алати за тестирање секвенцијалних програма су добра почетна основа, али нису директно применљиви на стабла задатака. Очито је да се методи за секвенцијалне програме могу применити само на нивоу једног TTE задатка. Са друге стране, постојећи вишенитни методи су програми са просто-разгранатим независним нитима базираним на pthreads API, док TTE стабло задатака садржи фино-разгранате независне задатке који користе TTE API. Алтернативно, програми за обраду порука су усмерени на модел програмирања један програм више података (SPMD) где процес извршења на географски дистрибуираним рачунарима извршава различите делове петљи итерационог простора, док TTE стабло задатака типично извршава паралелне процесе на географски раздвојеним великим низовима. Дакле,

постоји потреба за прилагођавање постојећих метода за ову, нову врсту паралелних програма, а можда и за истраживање новог приступа и стварања нових метода. Управо приступ коришћења статистичког тестирања и процене поузданости (Woit, 1993 a,b, 1994 a,b; Broakman, 2002) је прилагођен за тестирање великог скупа стабла задатака базираног на оперативним профилима (Поповић, 2010). У раду су представљене две методе за тестирање стабла задатака, које су развијене прилагођавањем традиционалне методе за тестирање секвенцијалних програма.

Прво је прилагођена метода потпуног тестирања секвенцијалних програма, који је најистакнутији детерминистички, познат као контролисан, тестни метод. Прилагођен метод за потпуно тестирање је у суштини коришћен као основа за упоредну анализу. Корисност овог метода је да можемо поново користити механизме контроле извршења стабла задатка за било коју врсту контролног тестирања. Друго, узет је пре добро установљен генерички метод за статистичко тестирање секвенцијалних програма, који промовише интересантну идеју коришћења случајно генерисане комбинаторне структуре, као граф, стабло, речи, путеви, итд. за статистичко тестирање секвенцијалних програма (Denise, 2004; Gouraud, 2005) прилагођених у генеричку методу за статистичко тестирање стабла задатака. Треће, поређена је прилагођена метода за потпуно тестирање и генеричка метода за статистичко тестирање стабла задатака са претходно прилагођеном статистичком методом тестирања. Напоменимо да су све методе коришћене на истом нивоу апстракције, тј. на нивоу TTE стабла задатака.

Циљ рада је прилагођавање генеричког метода за статистичко тестирање секвенцијалних програма на класу паралелних програма базираних на стаблу задатака. Такав адаптирани генерички метод (GMST) може деловати на фамилију стабла задатака, а не само на једно стабло задатака, као и да уважава различите развоје појединачног стабла задатака. Резултати тестирања овако прилагођеног модела биће поређени са резултатима добијеним прилагођеном методом потпуног тестирања (ET) и претходно прилагођеном статистичком методом тестирања (SUT). Тестирање је базирано на експерименталном мерењу времена извршења тестних случајева (тј. напора уложеног у тестирање) и покривености путева преко стабла задатака. При томе, постоји посебан интерес за откривање дубоко скривених грешака у чешће извршаваним путевима, јер је чињеница да присуство грешака у често извршаваним путевима драстично смањује поузданост производа.

Рад ће на основу резултата експерименталних мерења покривености путева и времена извршења свих тестних случајева дати

препоруке о коришћењу поменута три метода. Резултати истраживања ће се моћи користити за статистичко тестирање широке класе паралелних програма базираних на стаблу задатака и представљати солидну основу за даља истраживања у будућности.

У раду се користе подаци тестова као и ранијих истраживања на ФТН Нови Сад, Департман за рачунарство и аутоматику. Докторска теза се реализује у оквиру истраживачких активности на пројекту Министарства за просвету и науку бр. ТР-32031 за период 2011-2014.

У наредном поглављу биће дат преглед релевантних приступа области која је предмет изучавања на основу присутне литературе. Након тога биће извршена анализа оригиналних метода за статистичко тестирање програма као и анализа TTE стабла задатака. Након представљања нових метода за потпуно тестирање и статистичко тестирање паралелних програма, биће дат приказ резултата мерења са дискусијом и закључцима.

ПРЕГЛЕД ОБЛАСТИ ИСТРАЖИВАЊА

Неки приступи тестирању паралелних програма покушавају да их редукују на тестирање секвенцијалних еквивалената. Тако нпр. Sung (1998) користи предпроцесорски транслатор и проналазач пута да сведе тестирање паралелних програма на транслирани серијски програм. Алтернативно, Bocchino и други (2009) установљава за паралелни модел програмирања да је детерминистички у основи: детерминистичко понашање је гарантовано све док програмер експлицитно не користи недетерминистичке конструкције, и сматрају да се исти тестни комплети развијени за секвенцијални код, могу користи за такав паралелни код. Супротно томе, у овом раду је представљено директно тестирање паралелних програма писаних у актуелним програмским језицима.

Структурно тестирање или метода беле кутије (white-box), секвенцијалних програма је добро установљена техника тестирања која је изведена из логичке структуре програма који се тестира (Rapps, 1985). Тестни случајеви су изведени тако да неки елементи у програмској структури морају бити покривени током теста. Типичан критеријум структурног тестирања је да сви чворови, све гране и све дефинисане промењљиве користе парове дефиниције - коришћене у програму. Вреди напоменути да је структурно тестирање дефинисано као обавезно у постојећим прописима у сигурносно критичним системима (Zhu, 1997).

Недавно су Augustio и остали (2007) предложили фамилију тестних критеријума заснованих на току управљања и току података за аспектно орјентисане програме. Алтернативно, Jee и остали (2009) су пронашли класичне критеријуме покривености, базиране на графу тока управљања, који су неодговарајући када се користе за језик тока података, као што је функционални блок дијаграм (FBD). Као решење, они предлажу основну покривеност, покривеност улазних услова и покривеност сложених услова. Оба рада (Augustio, 2007; Jee, 2009) представљају интересантно проширење традиционалног структурног тестирања секвенцијалних програма.

Yang и Chung (1990) предлажу модел који представља извршење које укључује конкурентне путеве (Ц-путеве) и конкурентне правце (Ц-правце) конкурентних програма, описују стратегије извршења теста за откривање различитих грешака у конкурентном програму, али примењену методологију за избор Ц-путева и Ц-праваца нису представили у раду. Даље, Yang и Pollock (2009) представљају тестни оквир за све коришћене тестове покривености паралелних програма, који су примењени у вишенитним програмима, али они нису погодни за примену на TTE базиране програме. Алтернативно, у овом раду се користи дато стабло задатака као модел тока управљања и сви задаци развоја путева свих стабала задатака као критеријум покривености. Пошто овај метод покрива све путеве, он се назива потпуним тестним методом (ET).

Генерички метод за статистичко тестирање секвенцијалних програма (GMST-SP) је други тестни метод који је коришћен у овом раду. GMST-SP је иницијално био инспирисан радом Theveond-Fosse и Waeselynck (1991), где је избор расподеле улазног домена био вођен неким критеријумом покривености програма (структурно статистичко тестирање) или спецификације (функционално статистичко тестирање). Други извор инспирације за GMST-SP је био рад Flajolet, Zimmermann и Van Cutsem (Flajolet, 1994) који је генерализовао и систематизовао ефикасан алгоритам за генерисање, равномерно и случајно, облик различитих комбинаторних структура, оригинално развијене од Wilf и Nijenhus (Wilf, 1997; Nijenhus, 1979).

GMST-SP (Denise, 2004) је метод за статистичко тестирање према датом опису понашања система који се тестира. Користи хомогене случајне рутине генерисања путева из скупа путева извршења система. GMST-SP користи технике линеарног програмирања да би повећао вероватноћу да елементи буду покривени тестирањем. Проналазачи GMST-SP су демонстрирали примељивост њиховог метода и пратећих алата у истраживању где су конструисали више од 10.000 експеримената на програмској подршци коришћеној у индустрији, за више детаља видети (Gouraud, 2005).

Статистичка метода тестирања (SUT) је трећи тестни метод коришћен у овом раду. Denise M. Woit је урадила много у овој области како у својој докторској дисертацији тако и у сродним радовима (Woit, 1993, а,б; 1994, а,б). Вредно је напоменути да је приступ Woit-ове, аутоматизацији генерисања тестних сценарија и процени поузданости програмске подршке базираних на оперативним профилима, признат као de facto стандард и образац прихваћен у индустрији (Broakman, 2002).

Рад Woit-ове је био праћен у области тестирања базираног на моделима коришћењем генеричког окружења за моделовање (Поповић, 2005, 2006 а,б, 2007). Управо сада је Woit-ов приступ прилагођен за тестирање широке класе стабала задатака (Поповић, 2010). Наравно и многи други истраживачи су активни у области SUT. Као пример је обећавајући приступ заснован на расплинутој логици, који је презентован од Kumar и други (2007). Уопште, тестирање на бази модела ће и даље бити активна област за многе истраживаче.

Вреди напоменути да у овом раду није коришћен прилагођени случајни тестни метод (ART) (Chen, 2004 а,б, 2007) зато што је познато да он укључује значајне додатне покушаје у генерисању захтеваних тестних сценарија. Проналазач ART метода у његовом почетном раду (Chen, 2004 б) приказује како ART надмашује обично случајно тестирање (ORT) за фактор од 1 до 50% на упоредном тесту на програмима са 12 посејаних грешака. Цена достизања ових резултата је била прилично велика - ефективно је било потребно генерисати 10 пута више тестних случајева него што је било потребно за ORT.

ОРГИНАЛНЕ МЕТОДЕ

У овом поглављу биће представљене оригиналне методе за статистичко тестирање секвенцијалних програма: статистичка метода тестирања и генерички метод за статистичко тестирање.

Статистички метод тестирања секвенцијалних програма

Статистичка метода тестирања секвенцијалних програма (SUT-SP) је изведена из оригиналних дефиниција на нивоу модула (Woit, 1994), њиховом генерализацијом на ниво производа и укратко се може представити следећим дефиницијама (Поповић, 2010).

Дефиниција 1. *Производ* је пакет програма који скрива детаље програма. Сваки производ се сматра аутоматом са коначним бројем стања (FSM - Finite State Machine). Производ комуницира са околином преко спрега.

Дефиниција 2. *Догађај E* је комуникација производа са околином.

Дефиниција 3. *Улазни простор I* производа је скуп који обухвата све могуће јединствене догађаје.

Дефиниција 4. *Извршавање производа* је редослед догађаја $E_1\ E_2 \cdots E_t$ издат производу, који почиње са догађајем одмах после иницијализације производа и завршава се непосредно пре завршетка извршења производа.

Дефиниција 5. *Тестни случај* је извршење призвода, означен као $E_1\ E_2 \cdots E_t$.

Дефиниција 6. *Оперативни профил* је опис расподеле улазних догађаја који се очекују да се појаве у току рада производа. Конкретније, оперативни профил је модел понашања производа у његовој редовној

експлоатацији, који указује на оперативна стања производа и стања прелаза са датом вероватноћом. Спецификација оперативног профила је матрица S са m врста одговарајућих стања и n врста одговарајућих догађаја.

Дефиниција 7. *Стопа отказа fr* производа, је вероватноћа да ће извршење производа, бирано случајно за дати оперативни профил, изазвати грешку.

Дефиниција 8. У овој тези се разматра *оперативна поузданост* (у даљем тексту означена као *поузданост*) програмског производа као вероватноћу да извршење производа, бирано случајно за дати оперативни профил, неће изазвати грешку. Тако имамо:

$$r = 1 - fr$$

Дефиниција 9. *Ниво поверења M*, како је дефинисан од Woit у (Woit, 1994), као вероватноћа да производ има поузданост мању од r и још увек пролази дати тестни скуп са N тестних случајева.

За дату жељену поузданост производа r и ниво поверења M, захтевани број тестних случајева N се израчунава као:

$$N = \log_r M$$

где је N број тестних случајева, r жељена поузданост и M жељени ниво поверења.

Дефиниција 10. *Тестни скуп T* је скуп од N тестних случајева произведен за дату спецификацију оперативног профила S.

Означимо са F матрицу учестаности догађаја током формирања тестног скупа T. Тада је елеменат f_{ij} од F посматрана учестаност догађаја E_j у стању s_i. Одговарајућа очекивана учестаност e_{ij} је базирана на израчунавању вероватноће P_{ij} специфициране у S.

Дефиниција 11. *Одступање* између посматране и очекиване учестаности догађаја E_j у стању s_i се рачуна као:

$$D_{ij} = \frac{\left(f_{ij} - e_{ij}\right)^2}{e_{ij}}$$

Укупно одступање за стање s_i је дато као збир појединачног одступања D_{ij} за то стање ($j = 1, 2, ..., n$):

$$D_{ij} = \sum_{j} D_{ij} = D_{i1} + D_{i2} + \cdots + D_{in}$$

Дефиниција 12. Означимо са d_i посматрану вредност од D_i. Нивои значајности $SL_1, SL_2,\ldots,SL_m$ су вероватноће да су одступања велика као она посматрана између T и S као случајна одступања (P означава вероватноћу):

$$SL_i = P(D_i \geq d_i)$$

Већа вредност SL_i означава бољи квалитет генерисаног тестног скупа T, тј. боље слагање тестног скупа са спецификацијом оперативног профила.

Дефиниција 13. *Просечан ниво значајности E(SL)* је средња вредност свих нивоа значајности $SL_1, SL_2,\ldots, SL_m$.

Традиционална метода извођења оперативне поузданости производа за секвенцијалне програме састоји се од следећих корака:

1. Дефинисање оперативног профила.
2. За дати жељени ниво поузданости производа r, израчунати потребан број тестних случајева N.
3. Генерисање тестног скупа T који обухвата N тестних случајева.
4. Провера просечног нивоа значајности $E(SL)$.
5. Уколико је просечан ниво значајности мањи од 20%, повратак на корак 3.
6. Извршење на производу који се тестира у тестном окружењу као што је прописано у T.
7. Извештавање о неочекиваном понашању пројектантском и импелментационом тиму.

Тестни случајеви се генеришу преласком стања оперативног профила према вероватноћи појединачних стања прелаза. Типично, ово ради генератор тестних случајева, који одржава бројач учестаности f_{ij} посматраног стања прелаза. Тада се матрица F користи за израчунавање $E(SL)$ и генерисање укупног статистичког извештаја.

Уколико је просечан ниво значајности мањи од 20%, мора се генерисати нови тестни скуп. Иначе, генерисани тестни скуп се користи за рад производа у тестном окружењу, који обезбеђује снабдевање улазима и прихвата излазе из тестираног производа. Процес снабдевања улазима и прихватање излаза производа који се тестира је означен као аутоматско управљање тестирањем (test harness). Илустративан пример примене традиционалне методе се може наћи у (Поповић, 2006) у поглављу 5.

Генерички метод за статистичко тестирање секвенцијалних програма

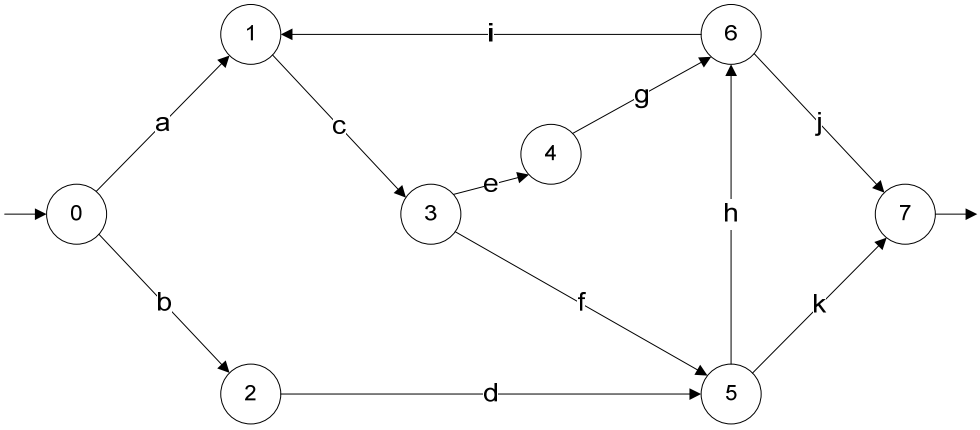
Генерички метод за статистичко тестирање секвенцијалних програма (GMST-SP) настао је као потреба за унапређењем технике статистичког тестирања, метод који комбинује статистички приступ са структурном или функционалном методом (Thevenod-Fosse, Waeselynck, 1991). Овај метод пружа могућност комбинације случајног тестирања и захтева покривености и има за циљ да превазиђе ограничења оба приступа. Примена критеријума покривености одговара разлагању улазног домена у поддомене који веома често нису хомогени. Ова нехомогеност уједно је и главни недостатак, неки улази могу изазвати грешку, док неки други могу давати коректне резултате. Случајно тестирање смањује ове недостатке, све док постоје интензивне тестне кампање, зато што се један елеменат програма извршава неколико пута са различитим подацима.

Метод је базиран на конструкцији вероватноће дистрибуције улазног домена, који за дати скуп елемената предвиђа неке критеријуме покривености, максимизирајући вероватноћу за елементе са најмањом вероватноћом, који ће бити активирани током извршења. Инспирисан овим радом Gouraud (Gouraud, 2005) је предложио нови приступ структурном статистичком тестирању према било ком графички описаном понашању система који се тестира (граф тока управљања, дијаграм стања, итд.). Најзначајнија оригиналност је комбинација резултата и алата из случајно генерисане комбинаторне структуре са решавањем ограничења, доносећи потпуно аутоматски метод генерисања теста. Уместо цртања улазних вредности, као код класичне методе тестирања, користи се униформна случајна рутина за генерисање путева из скупа могућих извршења путева система који се тестира, или много ефикасније, међу неким подскуповима који задовољавају неке услове покривености (Denise, 2004). Уколико је систем који се тестира описан на програмском нивоу, метод се користи за структурно статистичко тестирање а уколико је описан на нивоу спецификације, користи се за функционално статистичко тестирање.

Генерички метод статистичког тестирања секвенцијалних програма укратко се може представити следећим дефиницијама:

Дефиниција 14. Означимо са v_s почетни чвор графа тока управљања (CFG) и v_e крајњи чвор графа тока управљања, такав да за било који чвор v, постоји пут од v_s до v и пут од v до v_e у графу тока управљања. Пример

таквог графа приказан је на слици 1 (Denise, 2004), где су чворови означени бројевима од 0 до 7, а гране су означене словима од 'a' до 'k'; чворови 0 и 7 су почетни и крајњи чвор респективно.



Слика 1. Граф са почетним и крајњим чвором

Дефиниција 15. Уколико је n позитиван цео број, P_n (респ. $P_{\leq n}$) означава скуп путева дужине n (респективно, дужина мањих или једнаких n) у графу тока управљања од v_s до v_e, и $P_{\leq \infty}$ означава цео скуп путева од v_s до v_e.

Циљ је да се за дати цео број n, равномерно случајно генерише један или више путева дужине $\leq n$ од v_s до v_e. Равномерно означава да сви путеви у $P_{\leq n}$ имају исту вероватноћу да буду генерисани. Уколико се фокусирамо на мало другачији проблем: генерисање путева чија је дужина n, може се видети да мала промена у графу дозвољава генерисање путева $\leq n$. Генерално се може запазити да број путева дужине n расте експоненцијално са n.

Дефиниција 16. За дати било који чвор v, функција $f_v(m)$ означава број путева дужине m, која повезује чвор v са крајњим чвором v_e.

Претпоставимо да неки кораци за генерисање пута, који пролазе чвором v, који имају k наследника v_1, v_2, …, v_k. Такође претпоставимо да је $m > 0$ грана остаје да буде пређено да би добила пут дужине n до v_e. Тада је вероватноћа избора чвора v_i једнака $f_{vi}(m\text{-}1)/f_v(m)$ да би задовољила услов униформности, који захтева да вероватноћа да се дође до било код наследника од v мора бити пропорционална броју путева одговарајуће дужине од тог наследника до v_e. Вредности $f_v(i)$, за сваки $0 \leq i \leq n$ и сваки чвор v од графа тока управљања, у пракси се израчунавају коришћењем следећег правила рекурзије:

$f_v(0) = 1$, ако је $v = v_e$

$f_v(0) = 0$, иначе

$f_v(i) = \sum_{v \to v'} f_v(i - 1)$, за $i > 0$

значи $v \to v'$ да има грана од v до v'. Запазите да је v једнако v' у случају петље у графу тока управљања.

У следећој дефиницији користимо следеће означавање:

1. D је опис система који се тестира, који је базиран на комбинаторној структури, нпр. граф тока управљања.
2. C је дати критеријум покривености, такав као сви чворови, све гране, сви путеви, итд. (у овој тези од интереса је критеријум свих путева)
3. $E_C(D)$ је скуп елемената графа D, који мора бити прећен најмање једном, у одговарајућем скупу тестних случајева, такав да задовољи дате критеријуме C.

Дефиниција 17. Означимо $q_{C,N}(D)$ као квалитет метода у односу на C, који је дефинисан као минимална вероватноћа покривености било ког елемента $E_C(D)$ када представљамо N тестних случајева:

$$q_{C,N}(D) = 1 - \left(1 - q_{C,1}(D)\right)^N$$

где је $q_{C,1}(D)$ квалитет за један тестни случај.

Дефиниција 18. Означимо $AP{\leq}n$ као критеријум покривености (C) "свих путева дужине $\leq n$".

Квалитет критеријума $AP{\leq}n$ изражава се као:

$$q_{AP \leq n,N} = 1 - \left(1 - \frac{1}{|P_{\leq n}|}\right)^N$$

За дати жељени квалитет $q_{AP \leq n,N}$ и број путева $P_{\leq n}$, можемо израчунати потребни број тестних случајева као:

$$N \geq \frac{\log\left(1 - q_{AP \leq n,N}\right)}{\log\left(1 - \frac{1}{|P_{\leq n}|}\right)}$$

Када је спроведена за критеријум $AP{\leq}n$, оригинална GMST-SP метода садржи следеће кораке:

1. Конструисати граф тока управљања за дати програм.

2. Израчунавање вредности $f_v(i)$ за сваки $0 \leq i \leq n$ и сваки чвор v графа тока управљања коришћењем правила рекурзије како је представљено.
3. Израчунавање потребног броја тестних случајева N за дати захтевани квалитет q и броја путева чија је дужина мања или једнака n.
4. Униформно генерисање N путева (то значи да сви путеви имају исту вероватноћу).
5. Генерисање улазних података за генерисане путеве.
6. Извршавање N тестних случајева вођених генерисаним улазним подацима.

ИЗВРШИЛАЦ СТАБЛА ЗАДАТАКА

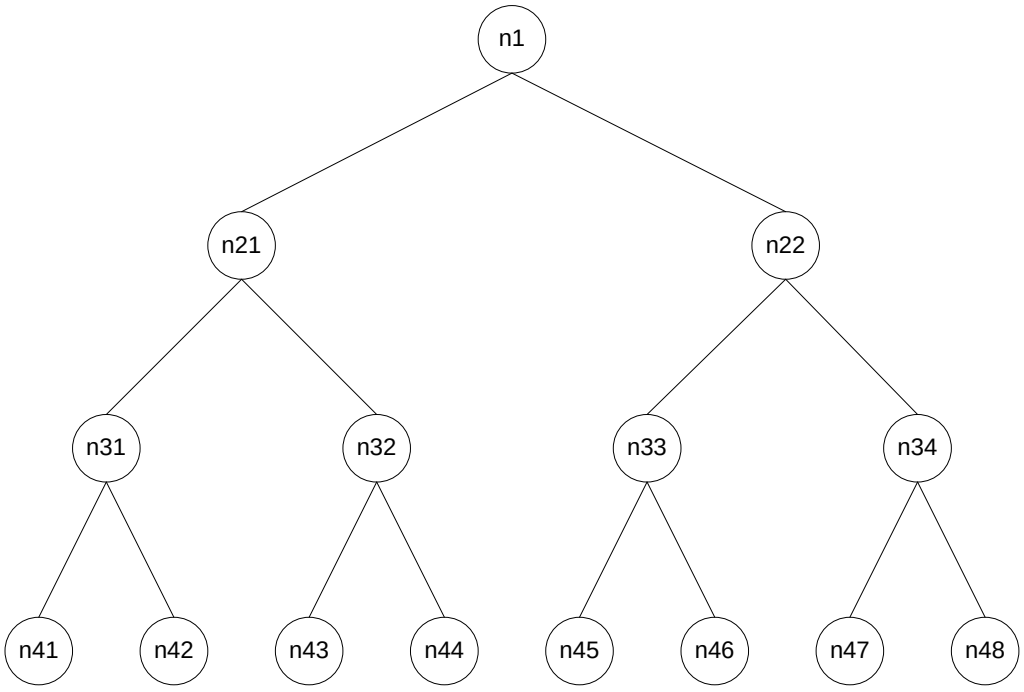
Извршилац стабла задатака (TTE - Task Tree Executor) је извршни модул настао као потреба реструктурирања наслеђеног секвенцијалног кода и извршавања стабла задатака на комерцијално расположивим оперативним системима (Поповић, 2009). Изградња веома великих дистрибутивних система, као што су системи за дистрибуцију електричне енергије, гаса, воде, итд. увек је посматран као веома захтеван подухват у области инжењеринга система заснованих на рачунару. Такви системи данас управљају са десетинама милиона улазних променљивих који се користе у веома сложеним математичким прорачунима а имплементирани су као секвенцијални програми. Један од највећих изазова је паралелизација овако наслеђених система као и истраживања нових могућности технологије које доносе модерни вишепроцесорски и вишенитни системи.

Представљени концепт паралелизације (Поповић, 2009) базиран је на дељењу података и конструкцији одговарајућег стабла задатака. Уколико посматрамо дистрибутивну мрежу електричне енергије концепт је базиран на дељење графа мреже и прављењу графа задатака, где појединачни задаци у графу задатака одговарају посебним деловима дистрибутивне мреже. У таквом груписању имамо групе задатака где сваки задатака делује само на делу графа мреже а не на читавом графу. Као подршка овом концепту развијен је извршни модул за прављење и извршавање стабла задатака на комерцијално расположивим оперативним системима, као што су Windows, Solaris и Linux. TTE може извршавати дати граф задатака од-дна-до-врха или од-врха-до-дна.

Сваки систем за управљање дистрибутивном мрежом представљен је моделом дистрибутивне мреже који има изглед стабла. У случају радијалне дистрибутивне мреже, модел је типично граф са једним кореном, где чворови графа одговарају чворовима дистрибутивне мреже, а лукови графа одговарају линијама преноса енергије које спајају чворове дистрибутивне мреже са изворима енергије. Пример таквог стабла приказан је на слици 2. Типичано, таква мрежа је дистрибутивна мрежа

електричне енергије где се израчунавање тока оптерећења обавља у две фазе. У првој фази за сваки чвор мреже се израчунавају струје коришћењем I Kirchhoff-овог правила. У другој фази се израчунава напон у сваком чвору мреже коришћењем II Kirchhoff-овог правила. Примена I Kirchhoff-овог правила у првој фази израчунавања тока оптерећења подразумева од-дна-до-врха прелажење графа мреже, док примена II Kirchhoff-овог правила у другој фази подразумева од-врха-до-дна прелажење графа мреже.

Гасоводна мрежа је, такође, пример такве мреже. У гасоводној мрежи чворови представљају тачке спајања цеви а гране означавају саме цеви (Купрешанин, 2002). Код гасоводне мреже израчунавање тока оптерећења, односно одређивање притисака у чворовима и протока кроз цеви, се обавља у две фазе. Фазе подразумевају израчунавање протока и притисака уз задовољавање једначина струјања гаса у цевима као и I и II Kirchhoff-ово правило. I Kirchhoff-ово правило односи се на биланс протока у сваком чвору, тј. мора бити задовољен услов да је збир протока у сваком чвору једнак нула. II Kirchhoff-ово примењује се на пад притиска при обиласку контура.



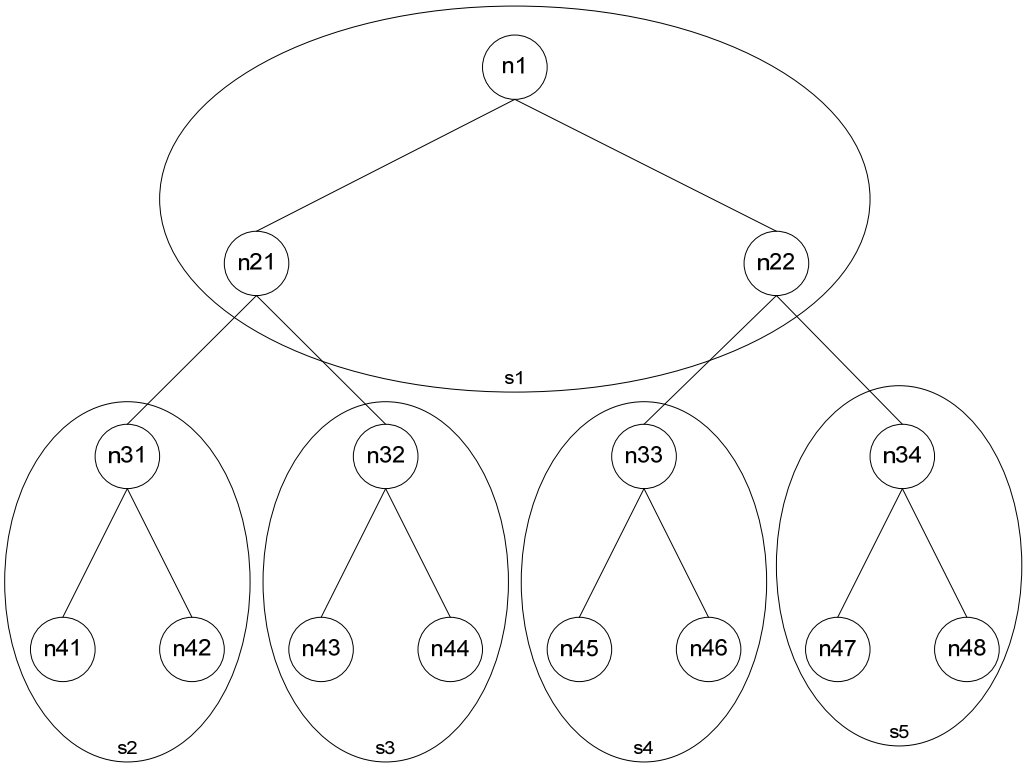
Слика 2. Пример графа мреже

Дакле, традиционално израчунавање тока оптерећења дистрибутивне мреже електричне енергије подразумева секвенцијално

прелажење графа мреже два пута, прво од-дна-до-врха да се израчунају струје у чворовима, и друго, од-врха-до-дна да се израчунају напони у чворовима. Коришћење секвенцијалног решења на вишепроцесорским системима доприноси њиховој слабој искоришћености, док се сви прорачуни извршавају на једном процесору, остали процесори остају бескорисни. Прорачуни тока оптерећења се могу паралелизовати дељењем графа мреже у делове графа и додељивањем једног задатка сваком делу. Зависност података утиче на конструкцију графа задатка, тако да редослед задатака прати редослед делова.

На пример, граф мреже на слици 2 може бити подељен у 5 делова, означених као *s1, s2 ... s5*, као што је приказано на слици 3.

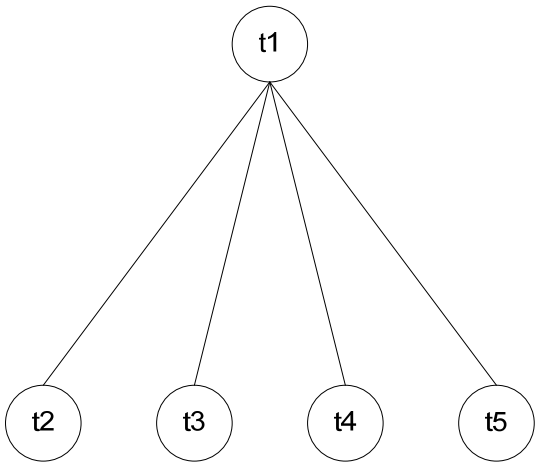
Сваки део графа мреже на слици 3 обухвата 3 чвора графа, нпр. део *s1* обухвата чворове графа *n1*, *n21* и *n22*. Доделом задатака делова графа мреже конструише се граф задатака који одговара деловима графа мреже (слика 4).



Слика 3. Подељени граф мреже

Граф задатака одређује извршење појединачних задатака. Пошто је интересантна паралелизација израчунавања тока оптерећења, гледамо

који задаци се могу извршавати паралелно. Веома је лако уочити да у случају од-дна-до-врха извршењу графа задатака, који одговара првој фази прорачун тока оптерећења, сви наслеђени задаци датог задатка *t* могу бити извршавани паралелно пре него што задатак *t* буде извршен. Слично, у случају од-врха-до-дна извршењу графа задатака, који одговара другој фази прорачуна тока оптерећења, задатак претходник датог задатка *t* мора бити извршен паралелно са другим задацима пре извршења *t* задатка.



Слика 4. Граф задатака

У случају стабла задатака приказаног на слици 4, веома је лако уочити да стабло може бити извршено од-дна-до-врха паралелним извршавањем *t2*, *t3*, *t4* и *t5* пре извршавања *t1*. Ако означимо секвенцијално извршавање τ_1 и τ_2 као $S(\tau_1, \tau_2)$ и паралелно извршавање τ_1 и τ_2 као $P(\tau_1, \tau_2)$, од-дна-до-врха извршење стабла задатака на слици 4, може бити формално записано као:

$$BUe = S(P(t2, t3, t4, t5), t1)$$

Уколико се стабло задатака извршава од-врха-до-дна паралелно извршавањем *t2*, *t3*, *t4* и *t5* паралелно после извршавања *t1*:

$$TDe = S\big(t1, P(t2, t3, t4, t5)\big)$$

Потпуно израчунавање тока оптерећења је низ од BUe и TDe дефинисан као:

$$LFe = S(BUe, TDe) \text{ или}$$

$$LFe = S\Big(S(P(t2, t3, t4, t5), t1), S\big(t1, P(t2, t3, t4, t5)\big)\Big)$$

Основна предност паралелног извршавања прорачуна тока оптерећења је скраћивање времена извршења. Паралелно извршавање стабла задатака приказаног на слици 4 почиње са паралелним извршавањем задатака *t2*, *t3*, *t4* и *t5* и завршава се извршењем задатка *t1*. Даље се исто стабло задатака извршава паралелно секвенцијалним извршавањем задатка *t1*, коме следи паралелно извршавање задатака *t2*, *t3*, *t4* и *t5*. Важно је напоменути да задаци у графу задатака могу извршавати различите функције када се извршавају од-дна-до-врха или од-врха-до-дна. Када се израчунава ток оптерећења дистрибутивне мреже електричне енергије, када се извршава од-дна-до-врха, извршавају се функције задатака специфичне за прву фазу прорачуна тока оптерећења (израчунавање струја - I Kirchhoff-ово правило), док извршење од-врха-до-дна, извршава функције задатака специфичне за другу фазу прорачуна тока оптерећења (израчунавање напона - II Kirchhoff-ово правило). Код израчунавања тока оптерећења дистрибутивне гасне мреже, када се извршава од-дна-до-врха, извршавају се функције задатака специфичне за прву фазу прорачуна тока оптерећења (израчунавање протока - I Kirchhoff-ово правило), док извршење од-врха-до-дна, извршава функције задатака специфичне за другу фазу прорачуна тока оптерећења (израчунавање притисака - II Kirchhoff-ово правило).

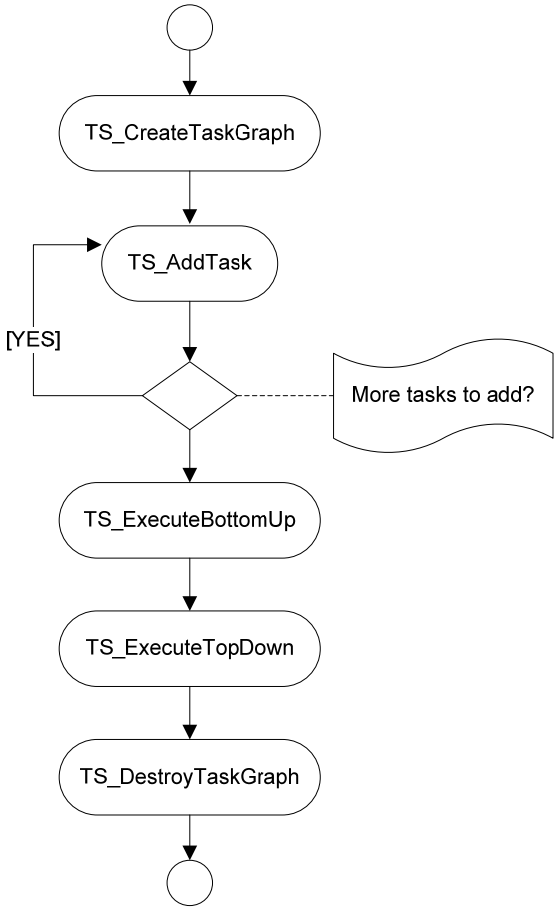
TTE програмска структура је конструисана и реализована коришћењем TTE API (Application Programming Interface) и садржи следеће C функције (Поповић, 2009):

1. *int TS_CreateTaskGraph(long troot, FNptr bu, FNptr td, long nthreads);*
2. *int TS_AddTask(long parent, long task);*
3. *int TS_DeleteTask(long task);*
4. *int TS_ExecuteBottomUp(void);*
5. *int TS_ExecuteTopDown(void);*
6. *void TS_DestroyTaskGraph(void);*

API функција *TS_CreateTaskGraph* ствара стабло задатака, где је *troot* идентификација (*id*) основног задатка, *bu* је показивач на од-дна-до-врха функцију обраде, *td* је показивач на од-врха-до-дна процесну функцију и *nthreads* је максималан број нити који се користи у паралелног извршавању стабла задатака. Функције обраде од-врха-до-дна и од-дна-до-врха су повратно-позиване функције којима се *id* задатка прослеђује као параметар. API функција *TS_AddTask* додаје нови задатак у стабло задатака, где *parent* представља *id* задатка претходника и *task* је *id* новог задатка. API функција *TS_DeleteTask* брише дати задатак и све његове наследнике у стаблу задатака. API функције *TS_ExecuteBottomUp* и

TS_ExecuteTopDown извршавају паралелно стабло задатака од-дна-до-врха и од-врха-до-дна респективно. На крају API функција *TS_DestroyTaskGraph* брише стабло задатака.

Нормалан животни циклус стабла задатака је приказан на слици 5. Почиње са стварањем стабла задатака, на почетку обухвата само један основни задатак. Сви остали задаци се додају поновним позивањем функције *TS_AddTask*. Једном, када је стабло задатака створено, оно се нормално извршава паралелно два пута, и то први пут од-дна-до-врха, и други пут од-врха-до-дна. На крају се цело стабло задатака обрише.



Слика 5. Нормалан животни циклус стабла задатака

Као пример представљамо *програм 1* који ствара и извршава једноставно стабло задатака. Програм 1 најпре ствара иницијално стабло задатака са основним задатком и дефинише од-дна-до-врха повратно-позивану функцију *CalcCurrents*, од-врха-до-дна повратно-позивану функцију *CalcVoltages*, и поставља број нити на 8 (види линију 2). Затим додаје наследнике основном задатку, названи задаци 1, 2 и 3 (линије 3-5); додаје наследника задатку 2, ово су задаци 4 и 5 (линије 6 и 7); извршава паралелно стабло задатака од-дна-до-врха (линија 8); извршава паралелно

стабло задатака од-врха-до-дна (линија 9) и на крају брише стабло задатака (линија 10). Када се задатак извршава, TTE позива одговарајуће повратно-позиване функције и преноси им одговарајућу идентификацију задатка. На пример, уколико се задатак 2 извршава од-дна-до-врха, TTE позива повратно-позивану функцију *CalcCurrents* и преноси јој идентификацију (ID) 2; слично уколико се задатак 3 извршава од-врха-до-дна, TTE позива повратно-позивану функцију *CalcVoltages* и преноси јој ID 3.

Програм 1:

```
1 void main(void) {
2       TS_CreateTaskGraph(0, CalcCurrents, CalcVoltages, 8);
3       TS_AddTask(0, 1);
4       TS_AddTask(0, 2);
5       TS_AddTask(0, 3);
6       TS_AddTask(2, 4);
7       TS_AddTask(2, 5);
8       TS_ExecuteBottomUp();
9       TS_ExecuteTopDown();
10      TS_DestroyTaskGraph();
11 }
```

Оригинално, TTE је уведен да дозволи паралелизам за наслеђени FORTRAN код за прорачуне дистрибутивне мреже електричне енергије. Ова мрежа је типично моделована као стабло мрежа, у коме је основни модел стабла трансформатор који је део високонапонског преносног система, средњи чворови стабла модел средњег и ниског напонског трансформатора, гране стабла модела преносне линије и лишће стабла модела крајњи корисници (индустрија, домаћинства, итд.). Секвенцијални прорачун тока оптерећења типично се обавља у два пролаза стабла мреже. Први пролаз је од-дна-до-врха и користи I Kirchhoff-ово правило за израчунавање свих струја у мрежи. Други пролаз је од-врха-до-дна и користи II Kirchhoff-ово правило за прорачуне свих напона у мрежи. Већина осталих прорачуна у дистрибутивној мрежи електричне енергије се обавља на исти начин.

TTE концепт паралелизма базиран је на подели стабла мреже у делове који дају дубину и стварају одговарајуће задатке који су одговорни за израчунавање на појединим деловима стабла мреже. Дубина одсечка стабла мреже дефинисана је као број средњих грана стабла, која почиње на врху одсечка а завршава се на дну одсечка. Резултат дељења на одсечке је стабло задатака које одговара стаблу мреже. Предност овог приступа је да неки задаци унутар стабла задатака могу бити извршени

паралелно, што води паралелној обради одговарајућих података смештених у моделу стабла мреже.

Мада TTE стабло задатака може изгледати као специјализована врста паралелних програма, она је уствари само врста програмских структура која постоји у многим сличним апликацијама, нпр. гасна и водоводна дистрибуциона мрежа, итд. Дакле, метод који је приказан у овом раду може бити директно примењен у широкој класи приказане инфраструктуре.

НОВЕ ПРИЛАГОЂЕНЕ МЕТОДЕ

Статистички метод тестирања

За разлику од других производа, који су у основи аутомати са коначним бројем стања који рукује са серијом догађаја, апликације базиране на стаблу задатака у принципу морају да раде на различитим стаблима задатака и локални оперативни систем мора распоредити паралелне нити (представљају задатке) у произвољан распоред. На пример, паралелни Систем за управљање дистрибуцијом (DMS - Distribution Management System) користи прост паралелизован систем FORTRAN кода који се извршава на врху TTE да прорачуна ток оптерећења за различите градове и регионе, нпр. град Београд или Болоња у Италији, који даје различита стабла задатака (Башичевић, 2009). Други пример је паралелна програмска структура TaskTreeExecutor, која је у основи извршна библиотека која извршава дато стабло задатка (Поповић, 2009). Она мора бити извршена у паралели од-дна-до-врха или од-врха-до-дна, за било које дато стабло задатака, за било који конкретан распоред паралелних нити, изабран од локалног оперативног система, нпр. MS Windows или Linux. Дакле, статистичко тестирање такве апликације мора да обезбеди и тестирање различитих стабала задатака и различитог распореда задатака. Да би се то обезбедило неопходно је решити два проблема (Поповић, 2010):

Проблем 1. Прилагодити традиционалне методе за апликације базиране на стаблу задатака.

Прављење оперативних профила је прилично осетљив и мукотрпан посао. Резултујући оперативни профил никада није апсолутно тачан модел оперативног окружења производа који се сертификује. Дакле, некада је пожељно извршити статистичко тестирање директно на оперативном окружењу, практично без матрице оперативног профила *S*. На пример, може бити корисно извршити тестирање статистичким

методом апликација базираних на стаблу задатака без познавања *S* за оперативни систем MS Windows. У том случају не можемо да израчунамо просечан ниво значајности E(*SL*).

Проблем 2. Дефинисати алтернативни индикатор квалитета тестног скупа који ће бити коришћен уместо E(*SL*), када је *S* непознат.

Ова два проблема се сматрају одвојеним, па тако требају и бити решена. Проблем 1 се сматра главним, док је проблем 2 додатни, мањи проблем. Циљ је да се реши проблем са минималним прилагођавањем традиционалног метода, зато што се он сматра стандардом.

Предложени и реализовани метод (Поповић, 2010) прилагођавања традиционалне методе извршен је увођењем нових дефиниција које представљају решење поменута два проблема.

Дефиниција 19. *Задатак* τ је повратно-позивна функција која се извршава као локална нит оперативног система.

Дефиниција 20. *Стабло задатака* је неусмерени радијални граф задатака *TG* чији су чворови задатака повезани релацијом веза које представљају релацију претходник-наследник. Стабло задатака које обухвата *k* задатака $TK=\{\tau_1, \tau_2, ..., \tau_k\}$ и скуп од (*k-1*) веза $L=\{l_1, l_2, ..., l_{(k-1)}\}$.

Дефиниција 21. *Корен rt* је претходник свих чворова у стаблу задатка.

Дефиниција 22. За било која два директно спојена чвора у стаблу задатака, чвор који је најближи корену је претходник осталих чворова, а остали чворови су његови наследници.

Паралелно извршење стабла задатка може бити формално описано као композиција два оператора, названих P() и S(), где први означава паралелно извршење његових параметара а други одговарајуће секвенцијално извршење (Поповић, 2009). Стабло задатака може бити извшено паралелно од-врха-до-дна или од-дна-до-врха.

Дефиниција 23. *Пут извршења стабла задатка*, назван *пут* у стаблу задатака или линија је низ крајева појединачних задатака $\tau_1, \tau_2, ..., \tau_k$ током извршења стабла задатка. Дужина овог низа за стабло задатака састављеног од *k* задатака је увек једнак *k*.

Дефиниција 24. *Шума задатака* је серија стабала задатака исте сложености (који обухвата исти број чворова) који је генерисан као тестни скуп.

Дефиниција 25. *Тестни случај* је појединачно извршење стабла задатка описаног одговарајућим путем.

Проблем 1 је стварно редуциран на дефинисање потребног броја тестних случајева N као функција жељене поузданости r и нивоом поверења M. Нова процедура за израчунавање N изведена је у наставку. Ако E_e означава успешно извршење производа, r је вероватноћа да је E_e истинито:

$$r = P(E_t)$$

Означимо са E_t и E_p успешно извршење произвољног стабла, преко произвољног пута, респективно. E_e је истинито ако су E_t и E_p истинити тј. $E_e = E_t\, E_p$. Тада заменом добијамо:

$$r = P(E_e) = P(E_t E_p) = P(E_t)P(E_p)$$

Означимо P(E_t) као *поузданост стабла производа* r_t и P(E_p) као *поузданост пута* r_p. Тада се поузданост производа добија множењем ова два:

$$r = r_t\, r_p$$

Ако претпоставимо да је $r_t = r_p$, тада имамо:

$$r_t = r_p = r^{\frac{1}{2}}$$

Слично је изведена и формула за ниво поверења M. Неопходно је запазити да је M уствари вероватноћа да је цела потврда једног E_{tn} *истинито негативна* (потврда је успешна, али у стварности поузданост је мања од r). Поставимо да U_t и U_p означавају сертификацију производа која није покрила стабло и да није покрила пут, који изазивају грешку, респективно. E_{tn} је истинито U_t или U_p истинито, тј. $E_{tn} = U_t + U_p$, тада следи да је:

$$M = M(E_{tn}) = P(U_t + U_p) = P(U_t) + P(U_p)$$

Означимо са P(U_t) као *ниво поверења стабла* M_t и P(U_p) *ниво поверења пута* M_p. Укупан ниво поверења M је збир ова два:

$$M = M_t + M_p$$

Ако претпоставимо да је $M_t = M_p$, тада имамо:

$$M_t = M_p = {}^{M}/_{2}$$

На крају, за дато r и M израчунавамо захтевани број стабала N_t и путева N_p за свако стабло као:

$$N_t = N_p = log_r^{1/2}(M/2)$$

Укупан број тестних случајева N се добија множењем N_t и N_p:

$$N = N_t N_p = \left(log_r^{1/2}(M/2)\right)^2$$

Сада се може поновно употребити традиционални метод - просто генеришемо N_t стабла задатака и сваки од њих извршимо N_p пута. Ово је ефикасно решење за раније изложен проблем 1.

Сада се враћамо на проблем број 2. Пошто традиционални инжењеринг програмске подршке користи метрику покривености улазног домена као индикаторе квалитета тестног скупа, чини се одговарајућим њихово коришћење као замена за просечан ниво значајности у ситуацијама када је S непознато. Две метрике које су одговарајуће за апликације базиране на стаблу задатака су *проценат различитих стабала задатака PDT* и *проценат различитих путева PDP*, који су покривени током сертификационог процеса. Ове метрике су пројектоване тако да добијају вредност 100% када су сва покривена стабла и сви покривени путеви јединствени, респективно. Нормално, ово може бити случај када је проверавана апликација базирана на класи стабла задатака велике скале.

Означимо N_{dt} као број различитих стабала задатака у шуми задатака од N_t стабала задатака. *PDT* је дефинисан као:

$$PDT = 100 \times \left(N_{dt}/N_t\right)[\%]$$

Бољи квалитет генерисаног тестног скупа подразумева већу вредност *PDT*, идеално је да буде 100%.

Означимо N_{dp} као број различитих путева током N_p извршења датог стабала задатака. *PDP* је дефинисан као:

$$PDP = 100 \times \left(N_{dp}/N_p\right)[\%]$$

Већа вредност *PDP* указује на бољи квалитет тестног скупа. *Просечна вредност PDP* за дату шуму задатака E(*PDP*) је такође идеално када је 100%.

Изложени метод за статистичко тестирање и процену поузданости апликација базираних на стаблу задатака, који решава оба изложена проблема, састоји се од следећих корака:

1. За дати жељени ниво поузданости производа, израчунати N_t и N_p.
2. Генерисати N_t стабло задатака.
3. Извршити свако стабло задатака N_p пута.
4. Проверити извештај о метрици покривености.
5. Ако је *PDT* или E(*PDP*) неприхватљиво мало, повратак на корак 2.
6. Извештавање о неочекиваном понашању система пројектантском и имплементационом тиму.

Утврђивање прагова за неприхватљиву вредност *PDT* и E(*PDP*) је зависно од будућег искуства и део је на коме се и даље ради. Предности коришћене методологије су двоструке. Прво, нема потребе за моделирањем експлицитних оперативних профила, који су најосетљивији део процеса статистичког тестирања. Друго *PDT* и E(*PDP*) индикатори квалитета тестног скупа су много значајнији за ову област него генерички E(*SL*). Ограничење предложеног метода је да се статистички модел тестирања мора обављати на циљној платформи.

Метод потпуног тестирања

Да би се могао применити на стабло задатака неопходно је прилагодити метод потпуног тестирања (ET метод) за ову врсту проблема (Поповић, 2012).

Дефиниција 26. *Лист* стабла задатака је чвор стабла који нема наследника.

Дефиниција 27. *Граф еволуције* стабла задатака садржи све путеве узетог стабла задатака са заједничким чвором корена.

Преласком графа еволуције од-врха-до-дна, нпр. следећи пут од корена према неком листу одговара једној посебној од-врха-до-дна еволуцији узетог стабла задатака, а преласком развоја графа од-дна-до-врха, нпр. у супротном смеру, одговара једној посебној од-дна-до-врха еволуцији узетог стабла задатака. Све док су од-дна-до-врха и од-врха-до-

дна еволуције узетог стабла задатака потпуно симетричне, можемо размотрити само једну, без икаквог губљења општости.

Дефиниција 28. *Фамилија стабала* задатака је шума стабала задатака која садржи сва стабла задатака исте сложености.

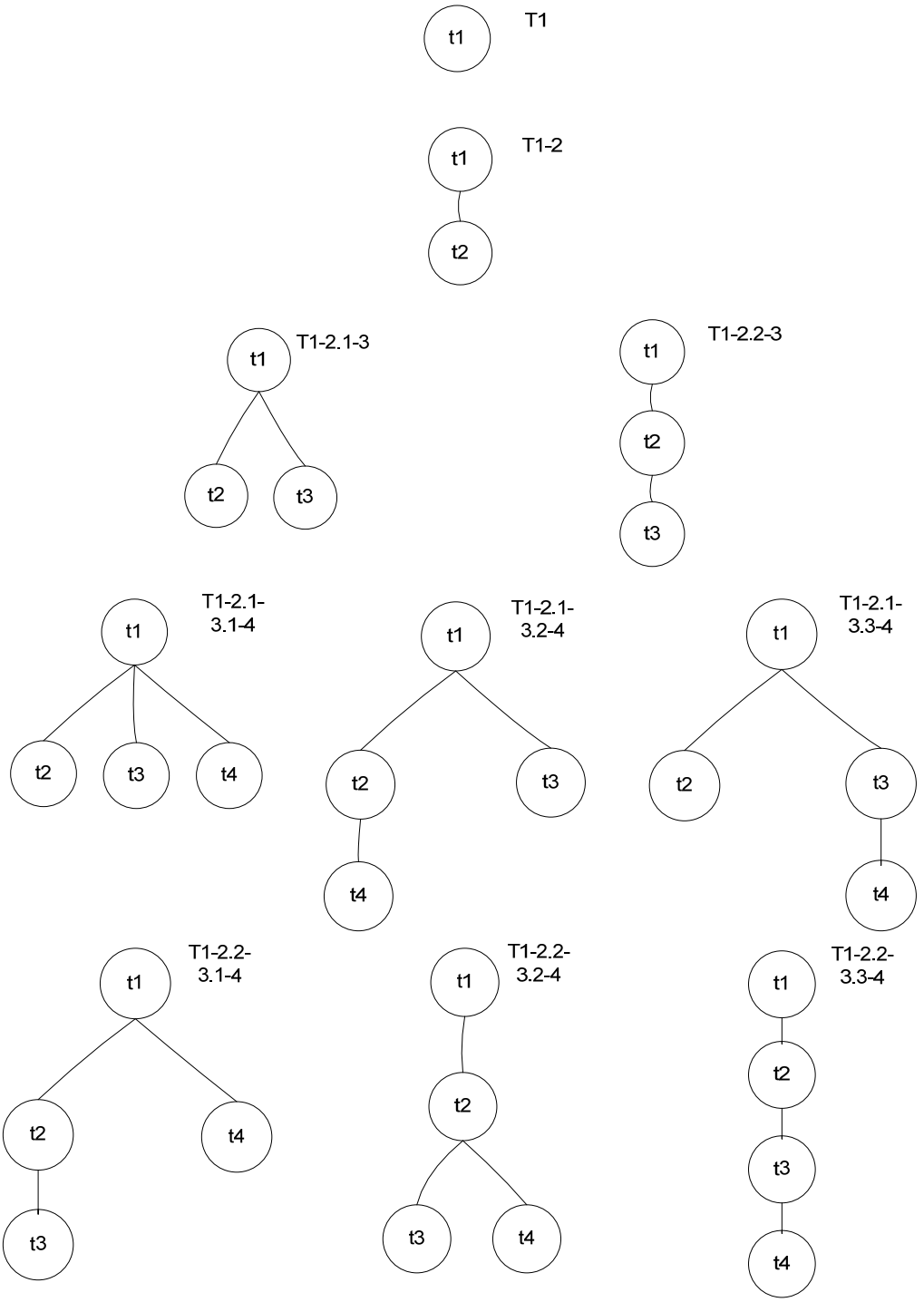
Када се конструише за дати број задатака, ET метод обухвата следеће кораке:

1. Генерише фамилију стабла задатка (коришћењем алгоритма 1-3).
2. За свако стабло задатака у генерисаној фамилији стабала задатака изврши кораке од 2.1. до 2.4.
   2.1. Ствара граф еволуције (коришћењем алгоритма 4).
   2.2. Сакупи све путеве (коришћењем алгоритама 5-7).
   2.3. Претвори све путеве у одговарајући план извршења.
   2.4. Изврши све планове на изабраном OS симулатору.

У тексту који следи описујемо детаље алгортима 1-7. Извор за алгоритам 1-3 долази од посматрања порекла фамилије стабала задатака (види слику 6). Слика 6 приказује четири стабла задатака који садрже 1, 2, 3 и 4 стабла задатака, респективно. Фамилија која обухвата један задатак има једно стабло задатака T1. Фамилија која обухвата два задатка, има такође једног члана T1-2. Фамилија која обухвата три задатка има два члана, чије је стабло задатака T1-2.1-3 and T1-2.2-3. Коначно, фамилија која садржи четири задатка има шест чланова, чија су стабла задатака T1-2.1-3.1-4, T1-2.1-3.2-4, T1-2.1-3.3-4, T1-2.2-3.1-4, T1-2.2-3.2-4, и T1-2.2-3.3-4. Означавање стабла задатака је извршено са намером да прикаже везе између чворова, који су кључ за разумевање начина изградње свих стабала задатака.

Почнимо са првим задатком фамилије стабла задатака. Један задатак можемо конструисати веома лако, постављањем тог задатка у стабло које ће одмах бити извршено. Одмах имамо потпуну фамилију стабала задатака који обухвата један задатак. Пошто смо конструисали прву фамилију, веома једноставно конструишемо другу. Како? Простим узимањем другог задатка и повезивањем са првим. На овај начин добијамо другу фамилију, која као и прва фамилија има само једног члана. Ово је генерално идеја како да конструишемо следећу фамилију из претходне. Сваки пут просто узимамо нови задатак и правимо ново стабло, повезивањем новог задатка у један задатак у претходном стаблу. Понављањем ових корака, системски, за све задатке у претходном стаблу,

добијамо сва стабла у новој фамилији, и тако ефикасно градимо нову фамилију. Овај приступ проверићемо за трећу фамилију задатака.



Слика 6. Порекло фамилије стабла задатака

Узмимо другу фамилију, која има само једно стабло T1-2. Сада узмемо задатак три и спојимо га са задатком један, и конструисали смо

стабло T1-2.1-3. Након тога узмемо задатак три, повежемо га са задатком два, конструишемо стабло T1-2.2-3, и завршили смо. Трећа фамилија је успешно конструисана. Слично, узимањем задатка четири и спајањем са сваким од задатака 1, 2 и 3 једном у реду, и са два стабла у трећој фамилији, конструишемо четврту фамилију, и тако даље за све фамилије. У наставку описујемо псеудо код за појединачне алгоритме (Поповић, 2012).

Функција *GenerateTreeFamily* у алгоритму 1 почиње са фамилијом празног стабла задатака (линије 2 и 3). Сваки следећи рекурзивни позив функције *GenerateTreeFamilyR* (линија 15) генерише следећу фамилију стабла задатака увођењем следећег задатка (почевши од задатка број 1), све док се жељена фамилија не конструише (линија 16). Следећа фамилија је конструисана из претходне на прилично једноставан начин. Функција *GenerateTreeFamilyR* у алгоритму 1 узима стабло задатака из претходне фамилије један по један (линија 11) и позива функцију *AddTaskToEachNode* у алгоритму 2 (линија 12), која један за другим прави нова стабла задатака увођењем новог задатка и повезивањем са задатком који је већ повезан у претходном стаблу задатака (линије 19-22). Функција *GenerateTreeFamilyR* и *AddTaskToEachNode* понавља овај процес за сва стабла задатака у претходној фамилији и за све задатке који су већ повезани у стабло задатака.

Алгоритам 1:

```
1 GenerateTreeFamily(noTasks)  =
2       TreeFamily empty
3       GenerateTreeFamilyR(noTasks,1,empty)
4
5 GenerateTreeFamilyR(noTasks, curTask,TreeFamily curFamily) =
6       TreeFamily fam
7       if curTask = 1
8             t1 ← new Task(1)
9             fam ← fam ∪ t1
10      else
11            for each task tree tr in task family fam
12                  fam ← AddTaskToEachNode(curTask, tr, fam)
13            DestroyTreeFamily(curFamily)
14      if curTask < noTasks
15            fam = GenerateTreeFamilyR(noTasks, curTask+1, fam)
16      fam
```

Функција *AddTaskToEachNode* у алгоритму 2 користи функцију *CloneTree* у алгоритму 3 да клонира дато стабло задатака. Такође се користи као метод стабла задатака *locateTask*(*id*) за лоцирање задатака са идентификацијом *id* (линија 20), метод *addNode*(*n*) за повезивање задатка са новим задатком *n* (линија 21), и конструкција *Task*(*taskid*) за конструисање новог задатка са идентификацијом *задатак* (линија 21).

Алгоритам 2:

```
17 AddTaskToEachNode(taskid, Tree tree, TreeFamily family) =
18     for each id ∈[1, taskid)
19         clone ← CloneTree(tree)
20         tk ← clone.locateTask(id)
21         tk.addNode(new Task(taskid))
22         family ← family ∪ clone
23     family
```

Функција *CloneTree* у алгоритму 3 ефектно клонира дато стабло задатака. Чини то стварањем корена задатка (линија 25) и додавањем његових наследника, наследникових наследника, кроз рекурзивни позив функције *CloneSubTree* (линија 35). Користи метод стабла задатака *getSuccessors* да би сакупила све наследнике задатка за дати задатак (линије 29 и 33).

Алгоритам 3:

```
24 CloneTree(Tree tree) =
25     clone ← new Task(1)
26     clone ← CloneSubTree(tree, clone)
27
28 CloneSubTree(Task task, Task clone) =
29     ttsucc ← task.getSuccessors()
30     for each task tt in ttsucc
31         clone.addNode(new Task(tt.getId()))
32
33     ctsucc ← clone.getSuccessors()
34     for each ti in ttsucc && each corresponding ci in ctsucc
35         CloneSubTree(ti, ci)
```

Функција *TDET_CreateEvolutionTree* у алгортиму 4 се извршава од-врха-до-дна. Почиње стварањем неименованог корена графа еволуције који је виртуални задатак са интерном (*id*) и екстерном (*eid*) идентификацијом једнаком 0 (линије 39-40). Променљива *enode* садржи

прву, још увек неозначену екстерну идентификацију чвора графа еволуције (линија 41). Екстерна идентификација је означена методом *setEid* (линије 40 и 50). Функција *TDET_CreateEvolutionTree* почиње са серијом рекурзивних позива функције *CreateEvolutionTreeR* у алгоритму 4, који један за другим клонира појединачне задатке из оригиналног стабла задатака (линије 49-50), и спаја ове клонове према свим могућим од-врха-до-дна развојима датог стабла задатака (линије 55-60).

Алгоритам 4:

```
36 TDET_CreateEvolutionTree(Task task) =
37      tasklist e ← {}
38      enode ← 0
39      tdegraph ← new Task(0)
40      tdegraph.setEid(enode)
41      enode ← enode + 1
42      CreateTDEvolutionTreeR(tdegraph, task, e)
43      tdegraph.deleteNode(0)
44      tdegraph
45
46 CreateEvolutionTreeR(Task parent, Task task,
47                              tasklist aliveSuccessors) =
48      tasklist ts ← {}
49      clone = new Task( task.getId() )
50      clone.setEid(enode)
51      enode ← enode + 1
52      parent.addNode(clone)
53      aliveSuccessors.remove(clone)
54
55      ts ← task.getSuccessors()
56      for each np in ts
57              aliveSuccessors ← aliveSuccessors ∪ {np}
58
59      for each np in aliveSuccessors
60              parent ← CreateEvolutionTreeR(clone, np, aliveSuccessors)
61      parent
```

Функција *TDET_GetPaths* у алгоритму 5 користи функцију *updateNoPaths* у алгоритму 6 за ажурирање броја путева, прелазећи сваки чвор за дати граф еволуције (линија 65), као и функција *getNextPath* у алгоритму 7, која враћа следећи пут (линија 67), почевши од првог и завршава са последњим.

Алгоритам 5:

```
62 TDET_GetPaths(Task tdegraph) =
63      paths pts ← {}
64      path pt ← null
65      nopaths ← updateNoPaths(tdegraph)
66      for i ∈ [0, nopaths)
67              pt ← getNextPath(tdegraph, pt)
68              pts ← pts ∪ {pt}
69      pts
```

Функција *updateNoPaths* у алгоритму 6 је саморекурзивна функција, која користи промењљиву *acc* да сабере број путева прелаза датог чвора у графу еволуције (линије 74 и 75) и метод *setNoPaths* да постави вредност у одговарајући атрибут чвора (линија 77).

Алгоритам 6:

```
70 updateNoPaths(Task root) =
71      acc ← 0
72      s ← root.getSuccessors()
73      if s = {}
74              acc ← 1
75      elese for each t in s
76              acc ← acc + updateNoPaths(t)
77      root.setNoPaths(acc)
78      acc
```

Функција *getNextPath* у алгоритму 7 конструише путеве кроз серију саморекурзивних позива (линија 89). Она користи метод *getNoPaths* и *setNoPaths* да смањи преостали број пређених путева датог развоја чвора графа (линија 86).

Алгоритам 7:

```
79 getNextPath(Task r, Path ip) =
80      Path path ← ip
81      path.append( r.getId() )
82
83      s ← r.getSuccessors()
```

```
84      if s = {} return path
85
86      r.setNoPaths( r.getNoPaths() – 1 )
87      for each t in s
88              if( t.getNoPaths() > 0 )
89                      path = getNextPath(t, path)
90                      break
91      path
```

Генерички метод за статистичко тестирање

Како би се генерички метод за статистичко тестирање секвенцијалних програма могао применити на стабло задатака, неопходно га је проширити са следећом дефиницијом:

Дефиниција 29. Означимо $\{T_1, T_2, \ldots, T_F\}$ као фамилију стабла задатака, где T_i представља i^{th} стабло задатака за фамилију ($0 \leq i \leq F$), означимо h_i број путева за стабло задатака T_i, и H као број путева читаве фамилије.

Претпоставимо да је интервал целобројних $R = [1, H]$ подељен тако да првих h_1 бројева означава путеве стабла задатка T_1, следећих h_2 бројева означава путеве за стабло задатака T_2, итд., последњих h_F бројева означавају путеве стабала задатака T_F. Избором случајног броја r из R, униформно бирамо пут z из свих путева фамилије стабла задатака. Једном када изаберемо r можемо користити функцију *getTaskTreeAndPathNo* у алгоритму 8 да одредимо стабло задатака T_i коме пут z припада, као и редни број j пута z са каталогом свих путева у стаблу задатака T_i.

Алгоритам 8:

```
92 getTaskTreeAndPathNo(r) =
93      k ← r
94      i ← 1
95      while k > 0
96              j ← k
97              k ← k - h_i
98              i ← i + 1
99      i ← i - 1
100     i, j
```

Генерички метод за статистичко тестирање примењен на стабло задатака садржи следеће кораке:

1. Генерише фамилију стабла задатака (користи алгоритам 1-3).
2. За свако стабло задатака T_i у фамилији генерисаних стабала задатака уради корак од 2.1 до 2.3.
   2.1. Ствара граф еволуције (користи алгоритам 4).
   2.2. Сакупља све путеве (користи алгоритам 5-7).
   2.3. Постави h_i на број путева у T_i.
3. Израчуна $H = \sum h_i$
4. Израчуна N коришћењем H и жељени квалитет q (обележимо као $H=P_{\leq n}$).
5. За сваки од N тестних случајева урадимо кораке 5.1 до 5.4.
   5.1. Генеришемо случајни број r.
   5.2. Одредимо одговарајући број стабала задатака i и број путева j коришћењем алгоритма 8.
   5.3. Конвертујемо пут z_j у одговарајући распоред задатака s_j.
   5.4. Извршимо распоред s_j на циљном OS симулатору.

РЕЗУЛТАТИ МЕРЕЊА

Генерално, грешка у недетерминистичком извршавању паралелног програма не може бити откривена чак и ако се извршава више пута (Yang, 1990). Као последица тога, у овом раду, наш циљ су скривене грешке, и посебно дубоко скривене грешке. Грешке су скривене у путевима уколико нису изложене приликом првог прелажења пута. Скривена дубина d скривених грешака је број потребних прелаза преко путева ради откривање скривених грешака. Грешку класификујемо као видљиву ако је d =1 и као скривену ако је d >1. Грешка је дубоко скривена ако је d >>1. Објашњење услова d >>1 зависи од апликације коју испитујемо. У нашем проучавању d >>1 интерпретирамо као d =10.

Ми смо посебно заинтересовани за откривање дубоко скривених грешака у чешће извршаваним путевима на циљној платформи. Мотивација за ово интересовање је чињеница да одсуство грешака у ретко извршаваним путевима не унапређује превише поузданост, али присуство грешака у чешће извршаваним путевима драстично смањује поузданост производа.

Тестирање је спроведено мерењем времена извршења тестних покушаја, покривености путева и способности откривања грешака за сваки од три презентована метода. Тестни објекат је TTE фамилија стабла задатака која је конструисана од 3, 4, 5, 6 и 7 задатака. Дубоко скривене грешке дубине 10 (d=10) су посејане у најчешћим путевима за свако стабло задатака у фамилији. Најчешћи путеви су претходно утврђени на основу статистике на стварном паралелном извршењу сваког стабла задатака на тестној платформи, која је била симетрични вишепроцесорски систем са два језгра, Intel® Core(TM) i5 CPU M 520 @ 2.4 GHz, 4 GB RAM, са Windows 7 Professional® 64-bit OS (Поповић, 2012).

Табела 1 приказује тестне покушаје (време извршења у секундама) за три презентоване методе. Колоне у табели 1 представљају:

1. Колоне *Број задатака*, *Број стабала* и *Број путева* приказују број задатака, стабала и путева, респективно.
2. Колоне *ET време*, *GMST време* и *SUT време* приказују време потребно за извршење тестних случајева ET, GMST и SUT методом, респективно. Колона *GMST време* је подељена у три подколоне за три карактеристичне вредности захтеваног квалитета q (0,9; 0,95 и 0,99). Слично, колона *SUT време* је подељена у три различите вредности поузданости r (0,9; 0,95 и 0,99; у сва три случаја M је изабран да буде M=1,4%).

Табела 1. Време извршења теста

Број задатака	Број стабала	Број путева	ET време [s]	GMST време [s]			SUT време [s]		
				q=0,9	q=0,95	q=0,99	r=0,9	r=0,95	r=0,99
3	2	3	0	0	0	1	2	10	273
4	6	18	0	1	1	1	4	14	298
5	24	180	1	3	6	6	4	15	365
6	120	2700	18	18	20	31	5	17	426
7	720	56700	230	534	605	874	5	20	485

Табела 2 приказује број непокривених путева за све три представљене методе. Колоне у табели 2 су следеће:

1. Колоне *Број задатака*, *Број стабала* и *Број путева* приказују број задатака, стабала и путева, респективно.
2. Колоне *ET*, *GMST* и *SUT* приказују број непокривених путева. Последње две колоне су подељене у три подколоне за различите вредности жељеног квалитета q и поузданости r, исто као у табели 1.

Табела 2. Број непокривених путева

Број задатака	Број стабала	Број путева	ET	GMST			SUT		
				q=0,9	q=0,95	q=0,99	r=0,9	r=0,95	r=0,99
3	2	3	0	0	0	0	0	0	0
4	6	18	0	1	1	0	3	2	0
5	24	180	0	17	9	0	138	115	53
6	120	2700	0	276	120	30	2532	2410	1832
7	720	56700	0	5588	2796	591	56360	55933	52921

Табела 3 приказује ниво дубине покривености за три представљене методе. Табела 3 је организована као и табела 2.

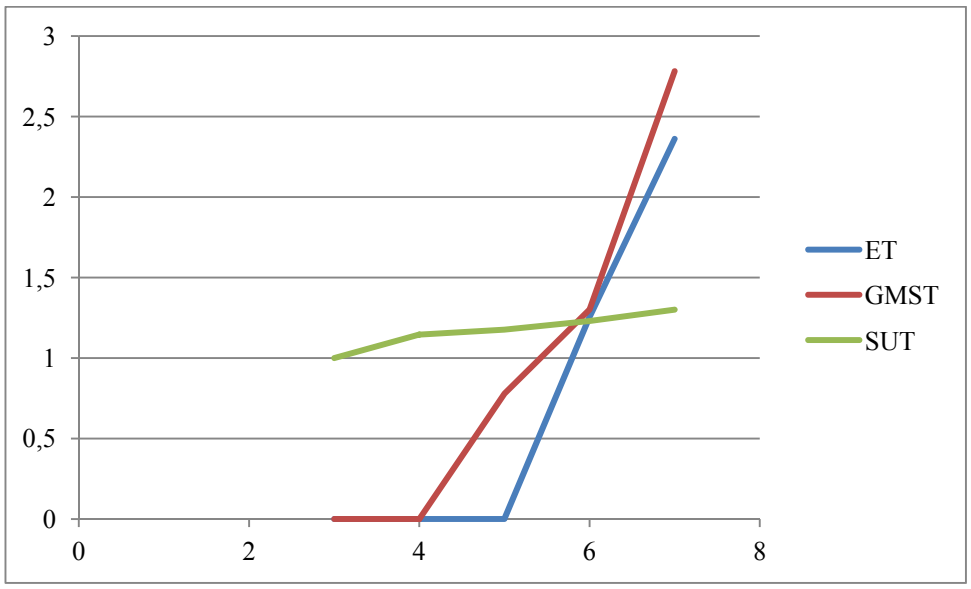
Табела 3. Ниво дубине покривености путева

Број задатака	Број стабала	Број путева	ET	GMST			SUT		
				q=0,9	q=0,95	q=0,99	r=0,9	r=0,95	r=0,99
3	2	3	1	2	3	4	3333	13333	333333
4	6	18	1	2	3	4	667	2500	55555
5	24	180	1	3	3	5	238	615	7874
6	120	2700	1	3	3	5	59	138	1152
7	720	56700	1	3	3	5	29	52	265

ДИСКУСИЈА

Резултати у табели 1 приказују експоненцијални раст времена тестирања ET метода са повећањем броја задатака, коришћених за конструкцију фамилије стабла задатака. GMST време тестирања приказује такође експоненцијални раст, који је чак и већи од ET времена тестирања, зато што GMST метода захтева много више тестних покушаја него ET метода. Насупрот томе, SUT време тестирања приказује логаритамски раст са растом броја задатака, почиње са већом вероватноћом али расте знатно спорије, и касније постаје мањи и од GMST і ET времена (за *r* мање од 0,99).

Овај тренд постаје много очигледнији после прегледа графикона 1 који приказује логаритам базе 10 и време извршења теста као функције броја задатака за ET метод, GMST метод (за q=0,95) и SUT метод (за r=0,95). Може се запазити мали раскорак између табеле 1 и графикона 1 који је настао услед замене нуле са јединицом у табели 1 да би се омогућило приказивање криве (зато што је $\log_{10}0$ недефинисано).

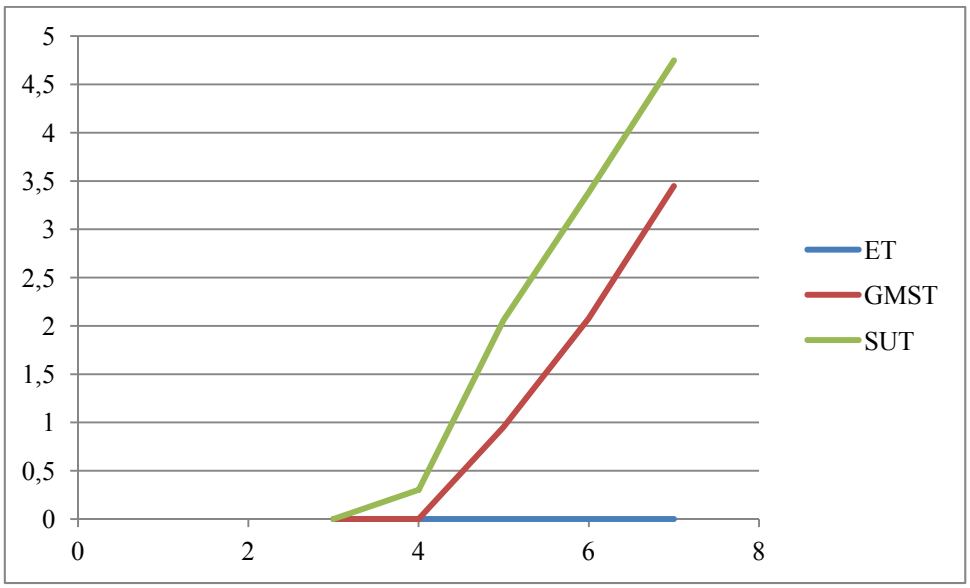


Графикон 1. Логаритам времена извршења према броју задатака

Из табеле 1 и графикона 1 постаје очигледно да је SUT скалабилнији са повећањем броја задатака, док методи ET и GMST нису скалабилни. Практично, SUT може бити коришћен чак и на нормалном desktop и laptop PC-ју. Са друге стране, апликација ET и GMST захтева дуже време извршења на много јачим серверима, који су типично расположиви за безбедносно критичне апликације.

Резултати у табели 2 показују да само ET метод пружа потпуну покривеност свих путева, пошто је број непокривених путева нула за све фамилије стабала задатака. GMST метод пружа парцијалну покривеност путева, где број непокривених путева расте експоненцијално са бројем задатака. Број непокривених путева за SUT такође показује сличан експоненцијални раст као и GMST метода, али је број непокривених путева знатно већи него код GMST метода.

Графикон 2 чини овај тренд очигледним са приказом логаритма базе 10 броја непокривених путева као функције броја задатака за ET метод, GMST метод (за q=0,95), и SUT метод (за r=0,95). Може се запазити да су све нуле у табели 2 замењене са јединицом да би се омогућио приказ логаритамске криве.



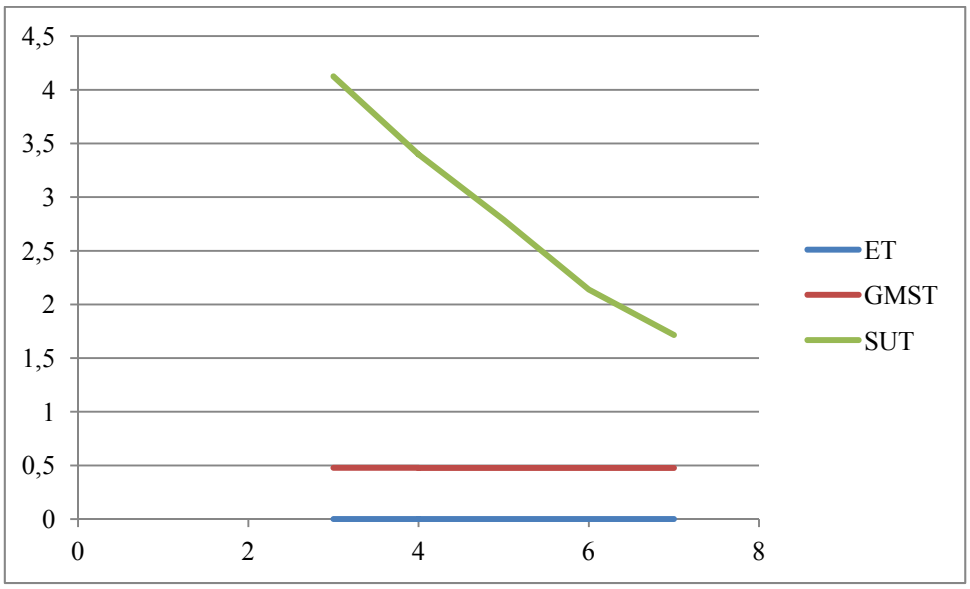
Графикон 2. Логаритам броја непокривених путева за дату дубину (d=10) према броју задатака

Графикон 2 показује да ET пружа тоталну покривеност путева, али се може запазити да је ова покривеност екстремно плитка, сваки пут је покривен само једном. Дакле, ET може открити само нескривене грешке (грешке дубине 1). Са друге стране, делимична покривеност путева у

случају GMST и SUT методе су објашњене случајним избором путева које користе. Ова случајност није хаотична, већ је она вођена значајном намером да фаворизује много вероватније путеве. Релативно сиромашна укупна покривеност за SUT метод је последица природе SUT методе - она фаворизује учесталије путеве а занемарује мање учестале путеве, али када је тестирање базирано на оперативним профилима производа, то је управо главни циљ тестирања. GMST пружа бољу укупну покривеност путева него SUT са равномерном фаворизацијом вероватнијих путева у односу на мање вероватне путеве, по цену повећања броја тестних покушаја.

Главна предност GMST у односу на SUT је да је покривеност путева GMST метода униформна, док покривеност путева SUT није униформна. Ова предност би могла бити препозната и интуитивно, с обзиром на природу GMST и SUT, али она постаје очигледна после анализе презентованих података у табели 3.

Резултати у табели 3 показују да ET метод има најмању дубину покривености пута, која је увек једнака 1, зато што је сваки пут покривен једном и само једном. SUT метод има значајнију дубину покривености пута него ET, која расте са поузданошћу r, али је експоненцијално смањује са бројем задатака. Дакле, ET-ева значајност покривености пута није скалабилна. Са друге стране, GMST метода такође има већи ниво дубине покривености пута него ET, што се такође повећава са захтеваним квалитетом q, али је главни квалитет GMST-овог нивоа покривености путева да је он константан у односу на број задатака, што значи да је скалабилан.



Графикон 3. Логаритам нивоа дубине покривености пута према броју задатака

Графикон 3 приказује логаритам базе 10 нивоа дубине покривености пута као функцију броја задатака за ET метод, GMST метод (за q=0,95), и SUT метод (за r=0,95). Заменом нула са један у табели 3 да би се омогућило цртање криве сада постаје значајан, зато што ове криве могу представљати ниво дубине покривености пута. Док ET не открива скривене грешке, ниво његове дубине покривености пута је нула, и заиста логаритам базе 10 значи пут покривености (који је 1) нула. SUT-ова стрма опадајућа крива линија приказује њену нескалабилност. GMST-ова крива је хоризонтална линија која доказује скалабилност у односу на број задатака, и такође приказује ниво дубине покривености путева. Заправо, табела 3 приказује да GMST-ов значај пута покривености за q=0,95 износи 3, што значи да је GMST у стању да открије скривене грешке дубине 3.

Чак је много значајније да се ниво GMST-ове дубине покривености путева веома лако регулише променом вредности q. На пример, табела 3 приказује да повећањем q на вредност 0,99 повећавамо ниво дубине покривености (одговарајућа хоризонтална линија на графикону 3 би ишла горе) тако да би значај пута покривености постао 5.

ЗАКЉУЧЦИ

Пошто ET метод све путеве покрива тачно једном, тј. не обезбеђује дубину покривености пута, захтева више тестних покушаја. GMST и SUT имају бољу дубину покривености пута него ET. SUT захтева мање тестних покушаја него GMST и ET, али његова дубина покривености путева опада са бројем задатака. Коначно GMST има предност над SUT зато што обезбеђује константно висок ниво дубине покривености пута, која не зависи од броја задатака, и који може бити регулисан захтеваним квалитетом теста q, тако да он постане дубљи са повећањем захтеваног квалитета q. Цена за ову предност GMST над SUT је више тестних покушаја, која може бити сматрана оправданом за безбедносно критичне пројекте.

Коначно, на основу резултата мерења препоручује се следеће:

1. Користити ET метод у случају када је циљ да се открију само нескривене грешке.
2. Користити SUT метод у случају када је доступан ограничен број тестних покушаја и када је циљ тестне кампање да открије дубоко скривене грешке само на најчешћим путевима извршења.
3. Користити GMST метод у случају када је доступан већи број тестних покушаја и када је циљ тестне кампање да открије скривене грешке у највероватнијим путевима, са униформном дубином покривености пута.

ЛИТЕРАТУРА

1. Augustio O, Lemos L, Vincenzi AMR, Maldonado JC, Masiero PC (2007). Control and data flow structural testing criteria for aspect-oriented programs, The Journal of Systems and Software. 80: 862-882.
2. Башичевић И, Јовановић С, Драпшин Б, Поповић М, Вртунски В (2009). An Approach to Parallelization of Legacy Software. Proc. 1st IEEE Eastern European Regional Conference on Engineering of Computer Based Systems held at Novi Sad, Serbia. pp. 42-48.
3. Башичевић И, Купрешанин И, Поповић М (2011). Operational Profiles for Statistical Testing of Distribution Management System. INFOCOMP Journal of Computer Science, Vol. 7, Num. 2, pp. 08-16.
4. Bocchino R, Adve VS, Adve SV, Snir M (2009). Parallel Programming Must Be Deterministic By Default. Proceedings of the First USENIX conference on Hot topics in parallelism held at Berkeley, CA. 22(1).
5. Broakman B, Notenboom E (2002). Testing Embedded Software, Addison-Wesley.
6. Chen TY, Huang DH, Kuo FC (2007). Adaptive Random Testing by Balancing. Proc. 2nd ACM International Workshop on Random Testing held at Atlanta, Georgia, USA. pp. 2-9.
7. Chen TY, Kuo FC, Merkel RG, Ng SP (2004). Mirror adaptive random testing. Journal of Information and Software Technology. 46:1001-1010.
8. Chen TY, Leung H, Mak IK (2004). Adaptive Random Testing. In Maher MJ (ed) LNCS 3321: ASIAN 2004. Springer-Verlag, Berlin / Heidelberg. pp. 320-329.
9. Denise A, Gaudel MC, Gouraud SD (2004). A generic method for statistical testing. Proceedings 15th IEEE International Symposium on Software Reliability Engineering held at Saint-Malo, Bretagne, France. pp. 25-34.
10. Flajolet P, Zimmermann P, Van Cutsem B (1994). A calculus for the random generation of labeled combinatorial structures. Theoretical Computer Science. 132:1-35.

11. Gouraud SD (2005). AuGuSTe: a tool for statistical testing – experimental results. Rapport de Recherche No 1400, CNRS and Universte Paris XI.
12. Intel (2011). Intel Cilk Plus. Online: http://software.intel.com/en-us/articles/intel-cilk-plus/. Last consulted: Dec., 2011.
13. Intel (2011). Threading Building Blocks. Online: http://threadingbuildingblocks.org/. Last consulted: Dec., 2011.
14. Jee E, Yoo J, Cha S, Bae D (2009). A data flow-based structural testing technique for FBD programs. Information and Software Technology. 51:1131-1139
15. Kumar K S, Misra R B (2007). Software operational profile based test case allocation using fuzzy logic, International Journal of Automation and Computing. 4(4):388-395.
16. Купрешанин И (2002). Један приступ вођењу гасоводног система коришћењем симулационог модела, Магистарски рад, Факултет техничких наука, Нови Сад
17. Nijenhuis A, Wilf H (1979). Combinatorial algorithms. Academic Press Inc.
18. Поповић М (2006). Communication Protocol Engineering. CRC Press.
19. Поповић М, Башичевић И (2010). Test case generation for the task tree type of architecture. Information and Software Technology. 52:697-706.
20. Поповић М, Башичевић И, Великић И, Татић Ј (2006). A Model-Based Statistical Usage Testing of Communication Protocols. Proceedings 13th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems held at Potsdam, Germany. pp. 377-386.
21. Поповић М, Башичевић И, Вртунски В (2009). A Task Tree Executor: New Runtime for Parallelized Legacy Software. Proc. 16th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems held at San Francisco, CA, USA. pp. 41-47.
22. Поповић М, Ковачевић Ј (2007). A Statistical Approach to Model-Based Robustness Testing. Proceedings 14th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems held at Tucson, Arizona, USA. pp. 485-494.
23. Поповић М, Купрешанин И, Башичевић И (2012). Generic Method for Statistical Testing of Parallel Programs Based on Task Trees. Scientific Research and Essays Vol. 7 (11), Academic Journals, pp. 1244-1255.
24. Поповић М, Великић И (2005). A Generic Model-Based Test Case Generator. Proceedings 12th IEEE International Conference and

Workshops on the Engineering of Computer-Based Systems held at Greenbelt, Maryland, Washington DC, USA. pp. 221-228.
25. Rapps S, Weyuker EJ (1985). Selecting software test data using data flow information. IEEE Transactions on Software Engineering. 11(4):367-375.
26. Souza SRS, Brito MAS, Silva RA, Souza PSL, Zaluska E (2011). Research in Concurrent Software Testing: A Systematic Review. Proceedings of the Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging held at Toronto, ON, Canada. pp. 1-5.
27. Sung AH (1988). Testing shared-memory parallel programs. Proceedings of the 2nd Symposium on the Frontiers of Massively Parallel Computation held at Fairfax, VA, USA. pp. 559-566.
28. Thevenod-Fosse P, Waeselynck H (1991). An investigation of software statistical testing. The Journal of Software Testing, Verification and Reliability. 1(2): 5-26.
29. Wilf H (1977). A unified setting for sequencing, ranking, and selection algorithms for combinatorial objects, Advances in mathematics. 24:281-291.
30. Woit DM (1993). Estimating Software Reliability with Hypothesis Testing. Technical Report CRL-263, McMaster University.
31. Woit DM (1993). Specifying Operational Profiles for Modules. Proceedings ACM International Symposium on Software Testing and Analysis held at Cambridge, MA, USA. pp. 2-10.
32. Woit DM (1994). A Framework for Reliability Estimation, Proceedings 5th IEEE International Symposium on Software Reliability Engineering held at Monterey, CA, USA. pp. 18-24.
33. Woit DM (1994). Operational Profile Specification, Test Case Generation, and Reliability Estimation for Modules. PhD thesis, Queen's University Kingstone, Ontario, Canada.
34. Yang CSD, Pollock LL (2003). All-uses Testing of Shared Memory Parallel Programs. Software testing, verification, and reliability. 13(1):3-24.
35. Yang RD, Chung CG (1990). A path analysis approach to concurrent program testing. Proceedings 9th IEEE Annual Phoenix Conference on Computers and Communications held at Phoenix, AZ. pp. 425–432.
36. Zhu H, Hall PAV, May JHR (1997). Software unit test coverage and adequacy. Computer survey. 29(4):367-427.