УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ

Ана Д. Митровић

ПРИМЕНА СКАЛЕ У ПАРАЛЕЛИЗАЦИЈИ РАСПЛИНУТОГ ТЕСТИРАЊА

мастер рад

Београд, 2018.

Ментор:

др Милена Вујошевић Јаничић, доцент
Универзитет у Београду, Математички факултет

Чланови комисије:

др Саша Малков, ванредни професор
Универзитет у Београду, Математички факултет

др Александар Картељ, доцент
Универзитет у Београду, Математички факултет

Датум одбране: 15. јануар 2016.

Наслов мастер рада: Примена Скале у паралелизацији расплинутог тестирања

Резиме:

Кључне речи:

Садржај

Глава 1

Увод

Глава 2

Програмски језик Скала

Скала је језик опште намене настао 2003. године са циљем да превазиђе ограничења програмског језика Јава комбиновањем објектно оријентисане и функционалне програмске парадигме. Мотивација иза овакве идеје је не ограничавати се на једну од ових парадигми и њених предности већ искористити најбоље из оба света. Управо због овакве комбинације парадигми Скала је веома погодна за решавање различитих врста проблема, почевши од малих незахтевних скриптова па све до великих компликованих система. На основу ове прилагодљивости је и добила своје име: реч „скала" означава „скалабилан језик" (енг. *scalable language*), односно језик који ће се прилагођавати и расти заједно са потребама система [20].

Творац Скале је Мартин Одерски (нем. *Martin Odersky*) (1958-), немачки научник и професор на универзитету EPFL (École Polytechnique Fédérale de Lausanne) у Лозани, Швајцарској. Још док је био студент, имао је жељу да напише језик који ће подржавати објектну и функционалну парадигму, говорећи да су ове две парадигме само две стране истог новчића. Желео је да се тај језик преводи у Јава бајт кôд али и да превазиђе ограничења језика Јава. Први резултат оваквог његовог рада је био језик Funnel, који због свог минималистичког дизајна није заживео. Затим је настала Скала, коју је професор Одерски развијао од 2001. године заједно са својом групом на универзитету EPFL. Познат је и по другим радовима, као што је имплементација GJ (Generic Java) компајлера који је постао основа javac компајлера [2, 19].

2.1 Опште карактеристике

Ово су неке од најважнијих особина Скале [20]:

Компатибилност са Јавом Скала није сама по себи продужење Јаве али је потпуно компатибилна са њом: њен изворни кôд се преводи у Јава бајт кôд који се извршава на Јава виртуелној машини. Из кода писаног у Скали је омогућено коришћење Јава библиотека, класа, интерфејса, метода, поља и типова. Омогућено је и обрнуто, позивање Скала кода из Јаве, мада се оно ређе користи. Такође, Скала омогућава лакшу и лепшу употребу Јава типова уз помоћ имплицитне конверзије омогућавајући употребу својих метода за манипулацију типовима. Компатибилност олакшава програмерима лакши прелазак из Јаве у Скалу јер нису принуђени да се одједном одрекну написаног кода у Јави.

Корен Скале није само језик Јава, иако је Јава имала највећи утицај у њеном стварању. Идеје и концепти из разних језика, како објектно оријентисаних тако и функционално оријентисаних, су инспирисали развој Скале. Међу њима су језици: C# од кога је Скала преузела синтаксне конвенције, Erlang чије идеје су сличне идејама конкурентности базиране на моделу Actors и други: C, C++, Smalltalk, Ruby, Haskell, SML, F# итд.

Објектно оријентисана парадигма Писање програма и смештање података и њихових својстава у класе и објекте тих класа је веома популарно и интуитивно програмерима. Скала је то задржала притом мало изменивши концепт. У многим језицима, укључујући и Јаву, дозвољене су вредности које нису објекти или које нису у склопу објекта. То могу бити примитивне вредности у Јави или статичка поља и методи. Скала то не дозвољава и зато је објектно оријентисан језик у *чистој форми*: „Свака вредност је објекат и свака операција је позив метода” [20]. Елегантан начин коришћења метода налик на операције је један од начина на који се Скала брине да програмерима буде пријатно њено коришћење.

Концизност Скала кôд има тенденцију да буде краћи од Јава кода, чак се процењује да има барем два пута мање линија кода од Јаве. Постоје и екстремни случајеви где је број линија и десет пута мањи. Ова особина није значајна само због тога што знатно олакшава програмирање, већ олакшава читање кода и откривање грешака којих ионако има мање јер

има мање простора за њих. Ово је нешто што одликује саму синтаксу језика, а веома помажу и разне библиотеке које имају већ имплементиране многе функције које врше послове са којима се често сусрећемо. Лако се и имплементирају и касније употребљавају библиотеке које сами напишемо. Такође, Скала подржава аутоматско закључивање типова (енг. *type inference*) што омогућава изостављање понављања већ наведених типова, што резултује читљивијим кодом.

Висок ниво Како је Скала погодна и за велике и комплексне системе она се труди да се прилагоди њиховим захтевима подижући ниво апстракције у свом коду. Омогућава много једноставнији и краћи начин кодирања разних проблема. Рецимо, уместо да пролазимо кроз ниску карактера карактер по карактер користећи петљу, у Скали се то може урадити у једној линији кода помоћу *предиката* тј. *функцијских литерала* који су детаљније објашњени у поглављу 2.2. Скала кôд тежи да буде разумљивији и мање комплексан како би олакшао већ комплексан систем који се имплеметира.

Статичка типизираност Скала је статички типизиран језик што значи да се типови података који су коришћени знају у време компилације. Неки сматрају ово маном као и да је навођење типова сувишно поред техника тестирања софтвера као што је нпр. тестирање јединица кода (енг. *unit testing*). Ипак, у Скали је статичка типизираност напреднија јер нам дозвољава да изоставимо тип на местима где би он био поновљен и вероватно би нам само сметао. Због оваквих понављања која су неопходна у неким језицима, многи се одрекну предности статички типизираних језика као што су: детектовање разних грешака у време компилације, лакше рефакторисање кода и коришћење типова као вид документације.

Својства Скала уводи појам својстава (енг. *trait*) који деле карактеристике са интерфејсима и наслеђивањем класа у Јави. У својствима као и у интерфејсима можемо декларисати методе, али и дефинисати поља и конкретне методе што није могуће у интерфејсима. Променљиве омогућавају чување стања а методе дефинисање одређеног понашања. То чини својства богатијим од интерфејса. Кажемо да се својства „миксују" у класе (енг. *mix in*). У једну класу је могуће миксовати више својстава.

Миксовање се може остварити кључном речју **extends** и у том случају класа наслеђује суперкласу својства. У случају да желимо да класа наследи неку другу класу, онда помоћу **extends** дефинишемо суперкласу а својства миксујемо помоћу **with** као што је илустровано у наредном примеру [20]:

```
class Animal
trait HasLegs
trait Philosophical
/* Superklasa "Animal" i miksovana svojstva "HasLegs" i "Philosophical" */
class Frog extends Animal with HasLegs with Philosophical {
       /* ... */
}
```

Највећа разлика између класе и својства односи се на позивање метода надкласе помоћу поља *super*. На пример, позивом метода *super.toString()* у случају класе, тачно се зна који ће метод *toString()* бити позван јер се зна надкласа дате класе. У случају својства, не можемо знати на шта се овај позив односи јер то директно зависи од класе у коју ће дато својство бити миксовано, као и од претходно миксованих својстава. То значи да се *super* позиви одређују *динамички*. Свако миксовано својство може позвати метод претходно миксованог својства. Ово омогућава класама да постигну различито понашање у зависности од редоследа миксованих својстава. Помоћу малог броја дефинисаних својстава може се постићи много различитих циљева комбиновањем редоследа миксовања.

Карактеристика која највише раздваја Скалу од Јаве је њена **функционалност** - Скала у потпуности подржава функционално програмирање. Ова особина је детаљно објашњена у следећем поглављу.

2.2 Функционални део језика

Функцинална парадигма се развија од 1960-тих година. У њеној основи леже *ламбда рачун* (енг. *λ-calculus*) и комбинаторна логика. Ламбда рачун представља математичку апстракцију и формализам за описивање функција и њихово израчунавање. Њега је увео Алонзо Черч (енг. *Alonzo Church*) 1930-тих година а Алан Тјуринг (енг. *Alan Turing*) је 1937. године показао да је експресив-

ност ламбда рачуна еквивалентна експресивности Тјурингових машина. Иако је ламбда рачун првобитно развијен само за потребе математике, данас се он сматра првим функционалним језиком. Други модел рачунања који је утицао на функционалне програмске језике, комбинаторну логику, створили су Мојсеј Шејнфинкел (рус. *Моисей Шейнфинкель*) и Хаскел Кари (енг. *Haskell Curry*) 1924. године са циљем да елиминишу употребу променљивих у математичкој логици [10, 5, 17, 5].

Најстарији виши функционални програмски језик је LISP који је пројектовао Џон Макарти (енг. *John McCarthy*) на институту МИТ (Масачусетски технолошки институт) 1958. године. Након тога се полако развијају и други функционални језици као што су ISWIM, FP, Scheme, ML, Miranda, Erlang, SML, Haskell, OCAML, F#, Elixir итд. Поред функционалних језика постоје и језици који нису функционални али у одређеној мери подржавају функционалне концепте или могу да их остваре на други начин, нпр. Java, C, C++, C#, Phyton [10, 5].

Разлика између императивне и функционалне парадигме се може описати дефинисањем самог програма [14]:

- Императивна парадигма:
  Програм представља формално упутство о томе шта рачунар треба да ради да би урадио неки посао
  Програм представља одговор на питање КАКО се нешто РАДИ

- Функционална парадигма:
  Програм представља формално објашњење онога што рачунар треба да израчуна
  Програм представља одговор на питање ШТА се РАЧУНА

Функционални језици су веома блиски математици због чега се и називају *функционалним* језицима: њихове функције се понашају као функције у математичком смислу. О томе говоре следећа два концепта којима се воде функционални језици [20]:

Први концепт се односи на **„статус" функција**. У функционалним језицима функције су вредности *првог реда*, тј. *грађани првог реда*. То значи да се оне могу користити као вредности типова као што су нпр. цели, реални бројеви,

ниске карактера итд. Статус оваквих типова се не разликује од статуса функција, што у многим језицима није случај. Ово омогућава коришћење функција на природан, концизан и читљив начин: као аргументе других функција, повратне вредности функција, њихово чување у променљивим, угњеждавање и слично. Другим речима, функције се креирају и прослеђују без икаквих рестрикција на које смо навикли у императивним језицима.

У Скали можемо користити локалне (угњеждене) функције које су често веома мале и лако се комбинују како би заједно заокружиле један већи посао. Ово одговара функционалном стилу који промовише следећу идеју: програм треба изделити на много мањих функција (целина) од којих свака има јасно дефинисан задатак.

Посебно су корисни и важни *функцијски литерали* који представљају функције које могу бити неименоване (анонимне функције) и прослеђиване као обичне вредности. Назив „функцијски литерал" се користи за функције у изворном коду, а у време извршавања оне се називају *„функцијским вредностима"* (разлика слична разлици између класа и објеката у објектно оријентисаним језицима). Функцијске вредности су објекти које можемо чувати у променљивама али су истовремено и функције које можемо позивати. Пример функцијског литерала који инкрементира цео број изгледа овако [20]:

```
var increase = (x: Int) => x + 1
increase(10)
```

Променљива **increase** је објекат који садржи литерал, али је и функција коју позивамо са аргументом 10.

Други концепт се односи на **непостојање стања** које се иначе представља променљивама у програму. То значи да извршавање метода нема *„бочне ефекте"*, односно да функције које позивамо не мењају податке „у месту" већ враћају нове вредности као резултате свог израчунавања. Избегава се (у чисто функционалним језицима се избацује) кад год је то могуће коришћење променљивих које мењају своје стање, а за промену се користе променљиве код којих то није дозвољено. То су променљиве које се иницијализују само једном. Оне се у Скали дефинишу помоћу кључне речи **val** и покушај промене вредности ове променљиве након што је иницијализована резултирао би грешком. Променљиве које су стандардне у императивним језицима и које можемо мењати по потреби се дефинишу кључном речју **var**.

Непостојање стања има за последицу избацивање итеративних конструкција. Овакав резон може испрва деловати чудно, међутим све што се може остварити кроз итерације се може остварити и *рекурзијом*. Рекурзија је у функционалним језицима потпуно природна и неизбежна. Некада се решења проблема који захтевају стање у оваквим језицима превише закомпликују, али оно се може направити по потреби на друге начине тако да се ово не сматра недостатком. Предност је у томе што не морамо пратити вредности променљивих и њихове промене што олакшава разумевање програма. Један програм у функционалном језику представља низ дефиниција и позива функција а његово извршавање је евалуација тих функција [10].

Као пример метода који нема бочне ефекте може послужити метод **replace** који као аргументе добија ниску карактера и два карактера. Његов задатак је да у датој ниски замени сва појављивања једног карактера другим карактером. Он неће променити ниску коју је добио као аргумент, већ ће вратити нову која више нема појављивања датог карактера који смо заменили новим.

Ова особина метода без бочних ефеката назива се *референтна прозирност* (енг. *referential transparency*): „за било које вредности аргумената позив метода се може заменити својим резултатом без промене семантике програма" [20]. Пример:

```
val p = previous(90)
```

Свако појављивање променљиве **p** можемо заменити изразом **previous(90)**.

Позив метода можемо заменити његовим резултатом јер смо сигурни да ће при сваком позиву са истим вредностима аргумената вратити исти резултат. Вредност резултата не зависи од бочних ефеката и тренутног стања променљивих, већ само од вредности прослеђених аргумената. Због тога, редослед позива таквих метода није важан и може бити произвољан. За овакве методе кажемо да су референтно прозирни или транспарентни. Они су концизнији, читљивији и производе мање грешака јер немају пропратне ефекте. Ипак, у пракси су нам углавном неопходни пропратни ефекти због чега они могу бити присутни на контролисан начин. У том случају, програмски језик више није чисто функционалан [10].

Чисто функционални језици (нпр. Haskel, Miranda) захтевају коришћење имутабилних структура података, референтно прозирних метода и рекурзије. Други језици, као што су Скала, Python, Ruby итд. охрабрују овакво про-

грамирање али не условљавају програмера. Скала омогућава бирање начина за решавање проблема и нуди функционалне алтернативе за све императивне конструкције. На овај начин програмер се полако и без притиска навикава на другачији начин размишљања [10, 20].

Неке од карактеристика које Скала подржава а које су заједничке и другим функционалним језицима су [20]:

Функције вишег реда Функције вишег реда су оне које имају друге функције као своје аргументе. Оне поједностављују кôд и смањују његово понављање. Ово је пример функције која као аргументе има ниску карактера **query** и другу функцију **matcher**:

```
def filesMatching(query: String,
                  matcher: (String, String) => Boolean) = {
        for (file <- filesHere; if matcher(file.getName, query))
                  yield file
}
```

Функција **filesMatching** треба да међу фајловима **filesHere** пронађе фајлове који испуњавају неки критеријум. Да се кôд не би понављао за сваки критеријум појединачно, он се прослеђује функцији као аргумент у виду функције **matcher**. Сам критеријум представља функцију која има две ниске карактера као аргумент, и враћа логички тип - тачно ако је критеријум испуњен и нетачно у супротном.

Следећи пример демонстрира коришћење функције вишег реда из Скала библиотеке:

```
def containsOdd(nums: List[Int]) = nums.exists(_ % 2 == 1)
```

Метод **exists** проверава постојање елемента листе **nums** који задовољава наведени предикат који му је прослеђен као аргумент (дељивост са 2). Постоји доста метода сличних методу exists, попут: **find**, **filter**, **foreach**, **forall**, итд.

Каријеве функције Каријеве функције уместо једне листе аргумената узимају аргументе један по један, имајући више листи које садрже по један аргумент. На овај начин омогућују дефинисање функција помоћу већ дефинисаних на следећи начин:

```
def curriedSum(x: Int)(y: Int) = x + y
```

Ово је функција која сабира два цела броја. Када је позовемо на овај начин:

```
curriedSum(1)(2)
```

десиће се два позива функције. Први позив ће узети први параметар **x** и вратити другу функцију која узима параметар **y**, и враћа збир ова два параметра. Прва и друга функција би редом изгледале овако:

```
def first(x: Int) = (y: Int) => x + y
val second = first(1) /* poziv prve funkcije */
second(2) /* poziv druge funkcije */
```

Каријев поступак има пуно примена. Рецимо, можемо дефинисати функцију која додаје број 1 неком целом броју:

```
val onePlus = curriedSum(1)_
onePlus(2) /* --> vraca 3 */
```

Позивамо Каријеву функцију **curriedSum** користећи _ као знак да на том месту може бити било која вредност (енг. *wildcard*). Суштина је да је функција **onePlus** дефинисана помоћу Каријеве функције која сабира било која два цела броја, а код које је први аргумент везан за број 1.

Упаривање образаца Често је потребно препознати да ли неки израз има одређену форму тј. да ли одговара одређеном обрасцу. То је корисно уколико треба да се имплементира функција која ради различите ствари у зависности од типова аргумената. Уобичајени примери су аритметичке операције или секвенце одређеног типа. Ово је пример са листама:

```
expr match {
       case List(0, _, _) => println("found it")
       case _ =>
}
```

Помоћу кључне речи **match** проверавамо да ли израз **expr** одговара некој листи од три елемента којој је први елемент 0 а друга два било који бројеви. Пошто се израз **expr** пореди редом са случајевима како су наведени, након неуспешног поређења са трочланом листом успешно ће се упарити са _ који служи као образац који одговара свему (енг. *wildcard pattern*).

Comprehensions Конструкција **for expression** или **for comprehension** се често користи у Скали не само за стандардно итерирање кроз колекције већ и на другачије начине, на пример:

```
val forLineLengths =
       for {
              file <- filesHere  /* iteriranje kroz listu fajlova */
              if file.getName.endsWith(".scala")  /* prvi uslov */
              line <- fileLines(file)  /* iteriranje kroz linije fajla */
              trimmed = line.trim  /* dodela promenljivoj trimmed */
              if trimmed.matches(".*for.*")  /* drugi uslov */
       } yield trimmed.length   /* povratna vrednost */
```

Овај пример обухвата пуно могућности које пружају **for** изрази. Прво, итерирање кроз листу **filesHere** користећи <- метод. Друго, филтрирање тих фајлова помоћу услова задатог у **if** изразу. Онда следи једна угњеждена петља где се врши итерација кроз линије тренутног фајла (који је испунио услов, а ако није, прелази се на следећи фајл). Након тога, чување тренутне линије без вишка бланко карактера у променљивој **trimmed** и још један **if** израз. Најзад, помоћу **yield** наредбе враћа се дужина линије чиме се попуњава листа **forLineLengths**. Дакле, можемо користити угњеждене петље, филтрирати резултате, чувати податке у променљивама и попуњавати колекције помоћу **yield** наредбе. Све ове опције нам омогућавају конструкције које подсећају на дефинисање елемената скупова у математици.

Управо функционална својства Скале су кључни разлози због којих је она веома погодна за паралелизацију израчунавања.

2.3 Паралелизам

Потреба за све већом моћи процесора непрекидно расте у складу са захтевнијим апликацијама. „Направи десет пута бржи процесор, и софтвер ће убрзо наћи десет пута више посла” [23]. Другим речима, стандарди расту докле год расту и могућности. Потребно је све брже и ефикасније извршавати захтевна израчунавања. Зато су годинама откривани и развијани разни начини побољшања перформанси процесора као што су повећавање брзине откуцаја часов-

ника, оптимизација тока извршавања и повећавање простора кеш меморије на чипу. Ова побољшања се односе и на секвенцијалне, непаралелне апликације као и на конкурентне апликације. Ипак, ови начини подизања перформанси полако достижу свој максимум. Муров закон каже да се број транзистора на процесорском језгру дуплира на сваких осамнаест месеци, описујући експоненцијалан раст броја транзистора. Као и друге области које експоненцијално напредују, ни ова не може напредовати вечно и у једном тренутку раст више неће бити могућ. Поставља се питање шта ће бити када достигнемо ограничења свих побољшања на која смо навикли? Још увек има простора за раст, али јавила се потреба за новим идејама. Тако је напредак процесора почео све више да се односи на број *централних процесорских јединица* тј. *језгара*. Данас имамо *multi-core* процесоре који имају у просеку од 2 до 8 процесорских језгара, и *many-core* процесоре са стотинама, па и хиљадама језгара. Сматра се да је конкурентност нова главна револуција у писању софтвера [23, 18].

Како главне улоге у новим побољшањима имају конкуретност и паралелизација, следе њихове дефиниције и појашњења [18, 15, 21]:

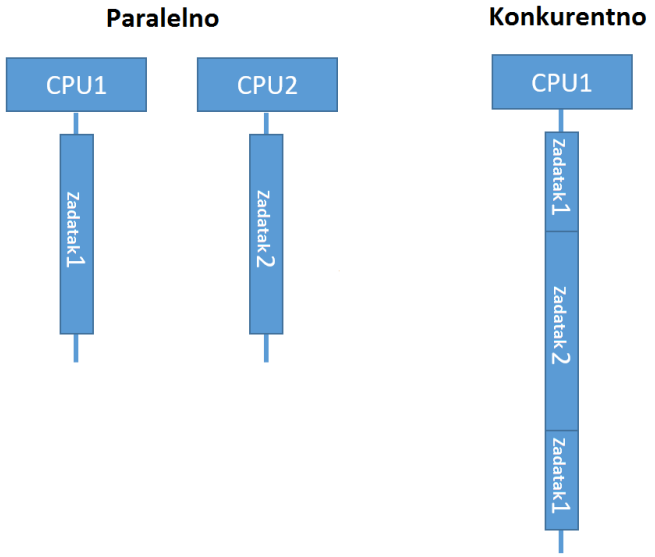
Дефиниција 1. *За два процеса кажемо да се извршавају* ***конкурентно*** *ако није унапред позната њихова међусобна временска лоцираност.*

Ово значи да имамо два процеса којима ће распоређивач (енг. *scheduler*), као нпр. Јава виртуелна машина, наизменично да додељује „кришке" времена (енг. *time-slice*). Због овога имамо привид да се ова два процеса извршавају дословно истовремено иако то није случај. Конкурентно извршавање се обично одвија када процеси деле једну процесорску јединицу.

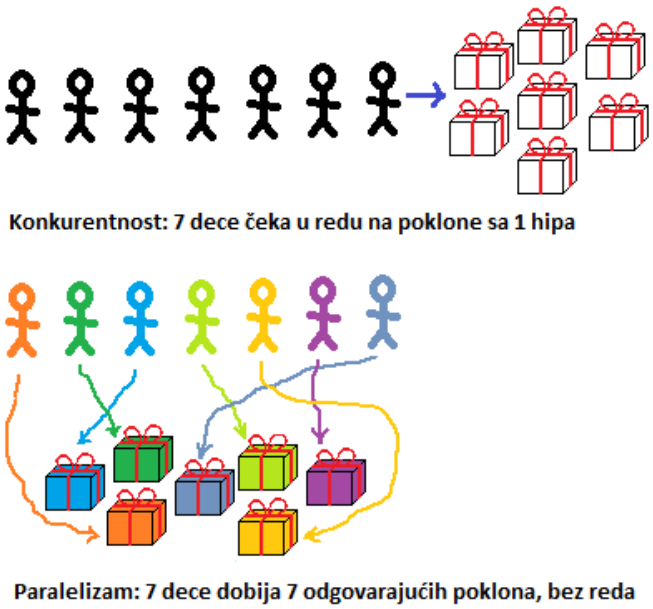
Дефиниција 2. *Два процеса се извршавају* ***паралелно*** *ако постоји период времена у коме су оба процеса истовремено активна.*

У случају паралелног израчунавања процеси се извршавају дословно истовремено на различитим процесорским јединицама. Паралелно израчунавање је подскуп конкурентног израчунавања, тј. може се представити као још један од начина на који се постиже конкурентност. Можемо написати конкурентну апликацију која има више просеца или нити, а са додатним језгрима процесора она може постати и паралелна.

Људи често мешају конкурентност и паралелизам и користе их као синониме, иако постоји разлика између њих. Ове појмове не треба тек тако мешати. Њихова различитост је демонстрирана сликама 2.1 и 2.2.



Слика 2.1: Разлика између конкурентног и паралелног израчунавања



Слика 2.2: Разлика између конкурентног и паралелног израчунавања

Оно што је скупо код конкурентног извршавања је промена контекста (енг. *context switch*) а то је промена стања процесора која је неопходна у случају када процесор са извршавања кода једног процеса прелази на извршавање кода другог процеса. Оваква промена се дешава јако често, до неколико стотина хиљада пута у секунди по процесорском језгру, што је незгодно јер узима до-

ста процесорског времена. Такође, подаци о стању процеса заузимају много меморије.

То је довело до појаве **нити** (енг. *thread*). Нит је компонента процеса која се извршава секвенцијално, тако да један процес може имати више нити. Нити омогућавају штедњу што се тиче ресурса ако постоје подаци које те нити деле. Самим тим, добија се и на времену. Нити се, као и процеси, могу извршавати конкурентно и паралелно.

Постоје различити модели тј. стилови паралелног програмирања. Ови стилови говоре о начинима комуникације процесорских језгара, нити, и начинима на који је то остварено. У наставку су објашњена два модела, модел дељења меморије и модел размене порука [18].

2.3.1 Модел дељења меморије

Рад са нитима подржава модел дељења меморије (енг. *shared memory model*). То значи да нити имају део меморије који им је заједнички, тако да могу асинхроно да читају и пишу на те меморијске локације. Ту се јављају разни проблеми. Треба омогућити безбедно дељење ресурса, комуникацију и синхронизацију нити. Случајеви када две или више нити не смеју истовремено имати приступ истим подацима су веома чести и зато се прибегава решењима закључавања као што су мутекси, катанци, монитори, семафори. Сврха ових решења је спречити недозвољене приступе подацима који се деле. Некада је дозвољено истовремено читање података, а некада је строго забрањен приступ свим нитима осим оне која тренутно чита и/или мења податке. Такве одлуке се доносе у зависности од конкретног случаја и потреба [21, 18].

Механизми закључавања решавају проблеме који се тичу приступа подацима, али уводе нове проблеме [21]:

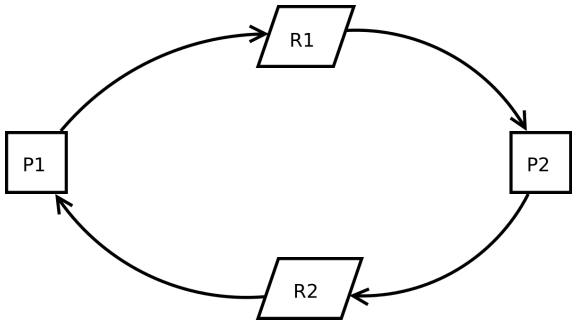
- **Катанци спречавају састављање** Као што је поменуто, функционални језици охрабрују писање мањих целина од којих свака има јасно дефинисан задатак. Ове мање целине на крају „саставе" цео програм. Ово није могуће ако користимо катанце.

- **Превише или премало катанаца** Закључавање и откључавање катанаца представљају скупе операције. Често није јасно колико је катанаца потребно да програм ради како треба а да не утиче на перформансе. Пре-

мало катанаца повећава шансу за недозвољеним приступом а са превише њих долази до пада перформанси и мртвих петљи.

- **Мртва петља и надметање за ресурсима** Ово су неки од проблема карактеристичних за конкурентно програмирање.

  Мртва петља (енг. *deadlock*) или *узајамно закључавање* је ситуација када више процеса или нити уђе у стање чекања због тражења ресурса који није доступан. Све ове нити чекају да им нека друга ослободи ресурс, али је то немогуће јер нит коју чекају такође чека неки ресурс. Пример је дат на слици 2.3: Процес P1 тражи ресурс R1 али не успева да му приступи



Слика 2.3: Мртва петља

  јер процес P2 држи ресурс R1. Слично, процес P2 тражи ресурс R2 али њега држи процес P1. Два процеса су у застоју јер се међусобно чекају. Проблем мртве петље се може решити тражењем катанаца у унапред дефинисаном редоследу, али и даље представља оптерећење за програмере [1].

  Надметање за ресурсима (енг. *race conditions*) представља ситуацију када два или више процеса или нити истовремено читају и мењају исти податак. Један од тих процеса поништи измене другог процеса, па резултат који се добије на крају зависи од редоследа којим процеси мењају дати податак. Редослед је немогуће предвидети тако да се приликом покретања оваквог програма углавном сваки пут добије другачији резултат [1].

- **Тежак опоравак од грешака** Не постоји одређени механизам за опоравак програма са више нити од грешака. Углавном је корисно проћи кроз *.log* фајл у коме је у различитим тренуцима записано стање стека са циљем праћења позивања функција и промена вредности променљивих.

Писање конкурентних програма је јако тешко. Нити су ниског нивоа, веома блиске хардверу и представљају начин на који сам процесор обавља послове. Постоји мало програмера који су у стању да напишу конкурентан програм без багова тако да тај посао треба препустити стручњацима у тој области. Један од главних разлога што је ово тежак посао је непредвидивост конкурентних и паралелних програма. Немогуће је утврдити којим редоследом ће се послови обављати, а веома је компликовано доказати да је програм *коректан* тј. да ради управо оно што се од њега очекује. Програмери се ослањају на своје логичко размишљање и надају се најбољим исходима. Друга ствар која је кључни разлог тежине конкурентног програмирања је коришћење променљивих података. Дељени подаци који се временом мењају захтевају синхронизацију и механизме закључавања, отежавајући паралелизацију конкурентног програма. То су разлози због којих је функционална парадигма погоднија од других за паралелизацију. Чим нема мутабилних тј. променљивих структура нема ни проблема које оне носе са собом [21].

2.3.2 Модел размене порука

Модел размене порука (енг. *message passing model*) је један од нових „трендова" који се тичу конкурентности. Овај модел подржава **SN** тј. *Shared Nothing* архитектуру. SN архитектура се односи на дистрибуиране системе и подразумева да се систем састоји из неколико независних чворова. Сваки чвор (тј. независна машина) има своју меморију, дискове и интерфејсе за улаз и излаз. Расположиви подаци се поделе овим чворовима тако да сваки од њих има одговорност искључиво за своје податке. Подаци се међу чворовима никада не деле што значи да сваки чвор има потпуну слободу над својим делом па нема потребе за механизмима закључавања. Како се међу чворовима ништа не дели, отуда назив ове архитектуре [22, 7].

Оваква логика недељивости стоји и иза модела размене порука. Компоненте овог модела међусобно комуницирају искључиво размењујући поруке. Свака компонента овог модела има своје стање (које јесте променљиво) али га никада не дели са другима, а поруке које шаље су непроменљиве. Поруке се могу слати синхроно и асинхроно [20].

Модел размене порука се може имплементирати на више начина, а најуспешнији начин је помоћу такозваног модела **Actors**. Њега је осмислио амерички научник Карл Еди Хјуит (енг. *Carl Eddie Hewitt*) који је 1973. године заједно

са другим ауторима објавио рад који представља увод у модел Actors. Иако се развија годинама, тек је од скоро доказано да је овај модел ефикасан у решавању проблема савремених рачунарских система. Он енкапсулира тежак рад са нитима и олакшава програмирање конкурентности.

Компоненте Actor модела представљају објекти које зовемо *Actors*. Они формирају хијерархијску структуру у којој сваки објекат има свог родитеља који сноси одговорност за њега. Сваки Actor има своје „поштанско сандуче" и његов задатак је да обради сваку поруку коју добије. Поруке се чувају у FIFO (енг. *first-in first-out*) структури тј. *реду* (енг. *queue*) па се поруке обрађују редом којим су стигле. Оно што разликује објекте Actor од других објеката је способност реаговања на поруке одређеном акцијом. Као одговор на поруку Actor може направити још Actor компоненти (своју децу), слати поруке другим компонентама или зауставити рад своје деце или себе. Actor има своје локално стање које је променљиво, али са другима не дели ништа осим порука. Важна особина Actor објеката је да се не блокирају када пошаљу поруку већ настављају са радом, односно слање порука је увек асинхроно.

Предности Actor модела и уопште SN архитектуре се огледају у избегавању свих поменутих проблема конкурентног програмирања чији највећи узрок представљају дељени променљиви подаци. Actor модел је погодан у апликацијама у којима је могуће посао поделити на што више мањих, независних послова. Тада сваки мањи посао представља задатак једне Actor компоненте. Родитељ решава проблеме своје деце и прикупља резултате њихових послова. Дизајн такве апликације личи на организацију послова у компанијама где се послови деле по одељењима, све док ти послови не постану довољно једноставни да их може обавити један радник.

Иако Actor модел у многим случајевима олакшава посао, постоје ситуације када је неопходно да компоненте деле податке између себе. Не треба на силу користити један модел ако природа проблема намеће неки други. Actor модел није универзално решење за све проблеме већ треба препознати случајеве када га није погодно користити. Неке од ситуација када треба прибећи другачијем решењу су следеће [21]:

Дељени променљиви подаци Неки проблеми природно захтевају дељење података. Тада је погодније изабрати рад са нитима, поготово ако се подаци деле само за читање. Рад са базама података и трансакцијама је пример када се програмери обично одлучују за неко друго решење.

Цена асинхроног програмирања Модел дељења порука са собом носи одређену сложеност. Дебаговање и тестирање великих апликација које имплементирају модел дељења порука може бити веома тешко. Наиме, компликовано је праћење асинхроно послатих порука што доводи до тешког проналаска извора проблема. У овом случају је корисна информација о првој послатој поруци. Такође, многим програмерима је тешко да се навикну на нову парадигму што изискује труд и време.

Перформансе Неке апликације захтевају највећу могућу брзину и ефикасност. Иако је Actor модел у већини случајева примењив, нити се налазе на нижем нивоу и самим тим омогућавају боље перформансе.

Разни језици имају библиотеке које имплементирају Actor модел. У следећем поглављу је обрађена одговарајућа библиотека у Скали [21, 9, 4].

2.4 Библиотека Akka

Akka представља имплементацију Actor модела на Јава виртуелној машини. Од верзије 2.10 језика Скала ова библиотека је подразумевана библиотека за коришћење Actor модела. Постојала је и библиотека *Actor*, али је она застарела и од верзије 2.11 се више не користи [8].

Све што се тиче Actor објеката је смештено у својству *Actor* које се уводи *import* наредбом:

```
import akka.actor.Actor
```

Кључне особине које одликују Actor компоненте су слање и примање порука. То се ради на следећи начин [21]:

- Слање порука се врши позивом метода **!**:

  ```
  a ! msg
  ```

  Објекту **a** се шаље порука **msg**.

- Примање порука се остварује помоћу **receive** блока и упаривања образаца које је објашњено у поглављу 2.2:

  ```
  receive {
          case pattern1 =>
  ```

```
            ...
        case patternN =>
}
```

Све компоненте Actor система су хијерархијски распоређене и организоване тако да се свакој приступа на јасно дефинисан начин. Ово је тема следећег поглавља.

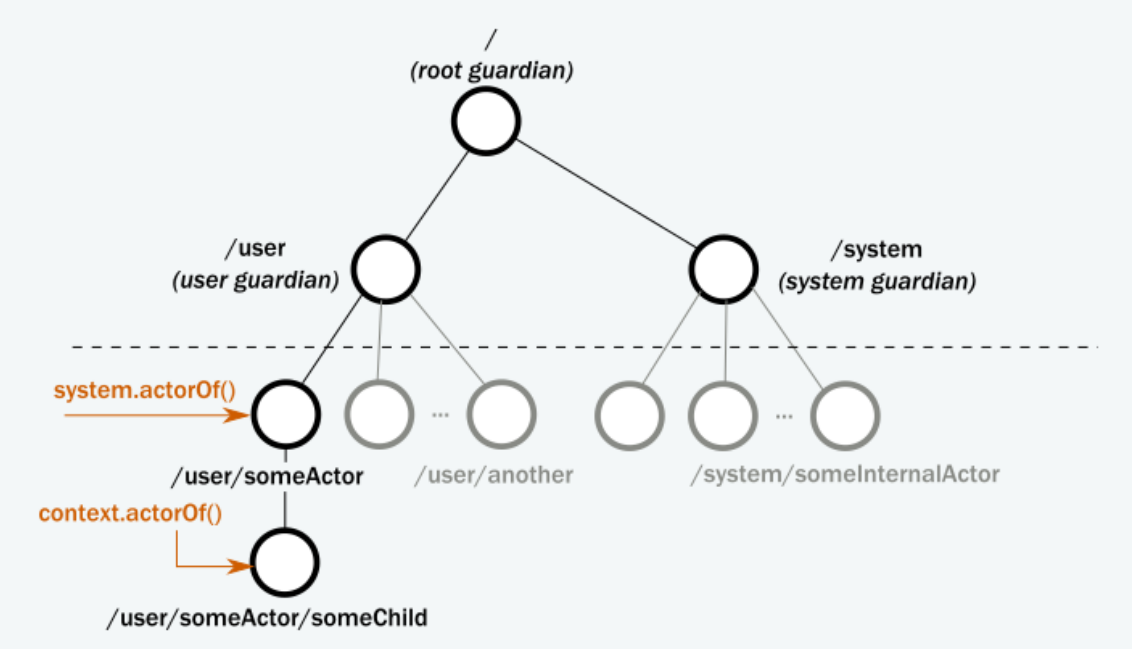
2.4.1 Структура

Све Actor компоненте које припадају једној логичкој компоненти апликације (нпр. једна за управљање базом података а друга за реаговање на захтеве корисника) припадају заједничком систему који се назива **ActorSystem**. То је хијерархијски уређена група компонената које деле исту конфигурацију. На ActorSystem можемо гледати као на менаџера својих компоненти који прави нове и претражује постојеће компоненте. Такође, ActorSystem је задужен за алоцирање 1...N нити које ће бити коришћене у апликацији [21, 3].

Све Actor компоненте једног система ActorSystem су смештене у **стабло** које је приказано на слици 2.4. Свака компонента има родитеља коме припада. На самом врху хијерархије налазе се три предефинисане компоненте [3]:

- *root guardian* је родитељ свих других компонената. Чак и он интерно има свог родитеља који се назива *„Bubble-Walker”*, али је он невидљив корисницима.
- *user gardian* је родитељ свих корисничких компоненти (компоненти које су направили сами корисници) и сва његова деца испред свог назива имају префикс „*/user/*”.
- *system gardian* је родитељ свих интерних компоненти. Интерна компонента, на пример, може бити компонента коју направи сама конфигурација оног тренутка када се креира систем ActorSystem.

Прву компоненту правимо помоћу метода *actorOf* самог система ActorSystem. Све компоненте направљене на овај начин ће постати деца предефинисане компоненте *user gardian*. Овакве компоненте су на врху хијерархије корисничких компоненти, па за њих кажемо да су на највишем нивоу иако постоје предефинисане компоненте изнад њих. Када се из једне Actor компоненте



Слика 2.4: Стабло Actor компонената

направи нова она постаје њено дете (енг. *child actor*) а родитеља називамо (енг. *parent actor*). Ова радња се постиже методом *actorOf* поља *context* Actor компоненте. Ово поље је типа **ActorContext** који омогућава једној компоненти да има приступ самој себи, свом родитељу и својој деци. Свако дете добија име свог родитеља као префикс свог имена.

Прављењем Actor компоненте или добијањем постојеће претрагом система не добијамо директну референцу на компоненту, већ добијамо показивач на референцу **ActorRef**. Ова референца је задужена за слање порука својој компоненти и штити је од директног приступа корисника. Свака компонента има приступ својој референци преко поља *self*, а током обраде примљене поруке преко поља *sender* има приступ референци компоненте која јој је поруку послала.

Оваква хијерархијска структура је слична структури фајлова у систему. У систему ActorSystem свака Actor компонента има путању **ActorPath** која је јединствено идентификује. Пошто се компоненте могу налазити и на више чворова на мрежи тј. извршавати се на различитим Јава виртуелним машинама, први део путање садржи протокол и локацију на којој се налази компонента (идентификацију ActorSystem система у коме се налази). Надаље путању чине надовезани елементи у стаблу раздвојени са „/”, почевши од корена. На пример

[3]:

```
"akka://my-sys/user/service-a/worker1"    // purely local
"akka.tcp://my-sys@host.example.com:5678/user/service-b"  // remote
```

Код примера локалне путање протокол је „akka" а ActorSystem је „my-sys". Међутим, нелокална путања дефинише другачији протокол и уз то име домаћина (енг. *host*) и број порта. У обе путање присутне су поменуте компоненте „root guardian" и „user guardian" што чини део путање „/user". Након тога следе корисничке компоненте које добијамо проласком кроз стабло све до жељене компоненте.

Још једна сличност ове хијерархије са хијерархијом система фајлова је начин на који претражујемо њене елементе. На пример [3]:

```
context.actorSelection("../*") ! msg
```

Метод **actorSelection** враћа референце компоненти које тражимо путем аргумента. Користећи „**..**" пењемо се један ниво изнад у стаблу тј. добијамо свог родитеља, а како „*****" представља знак за било шта (*wildcard*), добијамо сву децу свог родитеља, укључујући и себе. На овај начин лако шаљемо поруку више различитих компоненти.

Сваки ActorSystem има „диспечера" тј. елемент који се назива **MessageDispatcher**. Овај диспечер је задужен за слање порука у одговарајуће поштанско сандуче као и за позивање одговарајућег receive блока Actor компоненте. У позадини диспечера налази се складиште нити која се назива *thread pool*. Складиште садржи одређени број покренутих али неактивних нити које чекају да им се додели задатак. Сваки пут када је потребно извршити нови задатак, проверава се постојање слободне нити у групи. Ако постоји, бира се прва слободна нит. У случају да су све нити тренутно заузете, прва нит која се ослободи ће преузети нови задатак. Оваква идеја се примењује уколико знамо да ће бити пуно нити које ће извршавати кратке задатке, па је ефикасније имати унапред спремне нити него их изнова и изнова правити. На овај начин се такође смањује број нити које се креирају током рада апликације.

Thread pool даје важну гаранцију у конкурентној апликацији, а то је да у сваком тренутку само једна нит може извршавати одређену Actor компоненту. То значи да никада две или више нити не могу истовремено извршавати исту компоненту. Једну компоненту могу извршавати различите нити али

само у различитим временским интервалима. То нам омогућава да безбедно мењамо податке унутар Actor компоненте докле год те податке не делимо [21, 3].

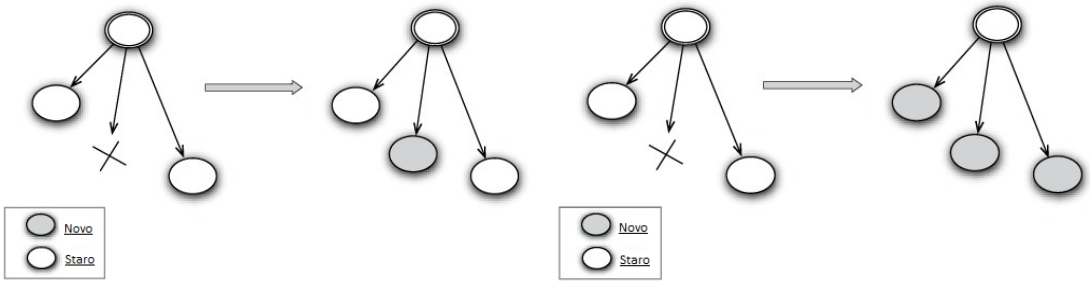
Компонента која је задужена за примање порука које су послате обустављеној или непостојећој компоненти назива се *dead letter actor*, а њено сандуче *dead letter mailbox*. Након што се одређена компонента угаси, није добра пракса направити нову са истом путањом. Разлог томе је то што не можемо претпоставити редослед догађаја и може се десити да нова компонента прими поруку која је била намењена старој [3].

Један од разлога због којих су компоненте смештене у хијерархијску структуру је управљање њиховим животним циклусима. Компоненте живе до момента када их корисник заустави и тада се прво рекурзивно заустављају сва њена деца. Ниједно дете не може надживети свог родитеља. То је корисно јер се на тај начин спречава цурење ресурса. Такође, препоручује се заустављање компоненте једино из ње саме. Добра је пракса да компонента као одговор на одређену поруку заустави саму себе док се заустављање произвољне компоненте из неке друге избегава.

Други разлог постојања хијерархије је управљање грешкама и изузецима. Akka подржава програмирање у коме се стање грешке посматра као било које друго валидно стање. Немогуће је спречити све грешке, тако да је циљ припремити начине на које ћемо се изборити са њима. У томе помаже структура компонената. Када компонента наиђе на проблем она привремено суспендује сву своју децу и саму себе, и обавештава родитеља о проблему. Њен родитељ одлучује како даље наставити са радом. Другим речима, сваки родитељ је задужен за решавање грешака и изузетака своје деце и тиме представља њиховог *супервизора*. Акција коју родитељ одлучи да предузме назива се *стратегија супервизора* (енг. *supervisor strategy*). Подразумевана стратегија је заустављање и поново покретање (рестартовање) детета и оно подразумева брисање акумулираног стања и враћање на почетак. Ову акцију је могуће предефинисати, тако да поред рестартовања постоје још три опције које родитељ може предузети [21, 3]:

1) Настављање са радом са акумулираним стањем

2) Трајно заустављање детета



(a) One-for-One (b) All-for-One

Слика 2.5: Стратегије рестартовања

3) Пропагирање грешке на ниво изнад тј. обавештавање свог родитеља о њој и тиме суспендовање себе

Што се тиче самог рестартовања, разликујемо две стратегије: *One-for-One* и *All-for-one*. Ове стратегије говоре о томе шта се дешава са осталом децом супервизора тј. родитеља компоненте која се рестартује и илустроване су на сликама 2.5a и 2.5b [21, 3]:

- **One-for-One** је подразумевана стратегија у којој током рестартовања одређене компоненте остале поменуте компоненте живе и настављају са радом независно од ње. Ова стратегија се примењује када компонента врши посао који не утиче на послове других компоненти.
- **All-for-One** стратегија се примењује када компоненте заједно учествују у неком послу тако да рестартовање једне повлачи рестартовање и осталих компоненти. У овом случају рестартовање једино компоненте у којој се јавила грешка не би било валидно.

2.4.2 Future и Promise објекти

Глава 3

Тестирање софтвера

Грешке у раду било које природе су неминовне, па тако и у програмирању. Развојем софтвера развијају се нове врсте пропуста тј. багова. „Постоје стотине слабости софтвера које само чекају да буду откривене, и биће откривене када прође довољно времена" [25]. Неки багови немају велики значај док други могу оставити озбиљне последице. Такође је написано много програма којима је циљ да искористе багове и наштете другом софтверу разним нападима система, вирусима, црвима, злонамерним скриптовима и слично. Скоро сваки систем може врло лако бити хакован. Циљ напада може бити преузимање или пад система, крађа, злоупотреба и мењање постојећих података, производња нових и лажних података итд. Зато је јако важно софтвер тестирати са циљем елиминисања што већег броја грешака које су настале током самог програмирања, а и са циљем проналажења слабости које се односе на безбедност тј. на могућност упада другог злонамерног софтвера [25, 13]

Тестирање софтвера представља најчешћи вид верификације софтвера. *Верификација* софтвера се бави доказивањем исправности програма тј. провером жељеног понашања програма задатог *спецификацијом*. Постоје *статичка* и *динамичка* верификација. Статичка испитује исправност програма без извршавања кода већ његовом анализом, док динамичка подразумева испитивање у току извршавања кода. То се најчешће обавља тестирањем [12].

Тестирање се може описати на више начина. Често се описује као процес коме је циљ процена квалитета софтвера. Овакву дефиницију тестирања треба узети са резервом. Тестирање нам свакако даје увид у квалитет софтвера али прилично мали, јер углавном случајеви које тестирамо представљају само кап у океану свих могућих случајева. Мање амбициозна дефиниција представља те-

стирање као процес који покушава да открије грешке у програму посматрајући његово извршавање у контролисаним условима [16].

Технике тестирања софтвера константно напредују, али треба имати у виду њихова ограничења. Холандски информатичар и добитник Тјурингове награде Едсгер Дајкстра (хол. *Edsger Wybe Dijkstra*) је рекао: „Тестирање софтвера може показати присуство багова, али никада њихово одсуство" [16]. Тестирање не може доказати исправност софтвера, већ се за то користе друге, математичке технике. Ипак, тестирање има важну улогу у свим фазама развоја софтвера. Помоћу њега смањујемо број промаклих грешака и повећавамо поузданост у изграђени систем.

Због ефикасности процеса тестирања, често је пожељна његова аутоматизација. Генерисање тест примера је у већини случајева потребно извршити ручно, мада постоје неке врсте тестирања у којима је могуће аутоматизовати овај процес. Што се тиче извршавања тест примера, аутоматизација је углавном могућа и њу подржавају разни алати за развој софтвера.

Важно је на који начин се приступа тестирању софтвера. Некада је дизајн тестова једнако компликован као дизајн самог програма. Било да се тестирање врши ручно или аутоматизовано, пожељно је имати планове и идеје о тест примерима који су вероватнији да изазову грешку од других. Треба одредити и редослед којим ће се извршавати осмишљени тестови јер постоје случајеви када неки тест примери остављају систем у стању у ком је могуће извршити друге тест примере. Након извршавања се прегледају добијени резултати и утврђује се спремност тестираног система. Сав посао везан за тестирање може вршити сам програмер који је уједно и тестер. Друга могућност је постојање тестера који поред програмирања разуме технике тестирања, познаје грешке које се често јављају и необичне случајеве који их производе. Он може да ради заједно са програмером а може бити и раздвојен од њега [6, 11].

„Покривеност кода (енг. *code coverage*) је број неких елемената програма испитаних тестовима у односу на укупан број тих елемената" [11]. Уопштено, покривеност представља неку врсту метрике која говори о броју случајева који су испитани тестовима. Када сматрамо да тестови испитују добар део програма, кажемо да тестови „покривају" велики број случајева тј. да имају велики ниво покривености. У зависности од изабраних елемената програма разликујемо покривеност путања, наредби, грана/одлука, услова, вишеструких услова и функција. Постоје различити алати за рачунање нивоа покривености кода

[11].

3.1 Нивои тестирања

Тестирање се врши на више нивоа, у зависности од сложености компонената које се тестирају. Почиње се од **тестирања јединица кода** (енг. *unit testing*), тј. независних делова система. Јединице представљају најмање независне целине које обављају неку функцију (нпр. класа или функција). Пре спајања више независних јединица, важно је да оне саме раде правилно, изоловане од других јединица. Други ниво, **компонентно тестирање** (енг. *component testing*), је веома слично првом нивоу. Компоненте су изоловане од других компонената, али су мало веће и јединице које су у њима нису изоловане једне од других. Трећи ниво је **интеграционо тестирање** (енг. *integration testing*). Оно је задужено за тестирање компоненти које заједно чине целину система. То је важно јер се неретко дешава да компоненте које савршено раде самостално не успевају добро да раде заједно и комуницирају једне са другима. Следећи ниво тестира систем као целину и зато се назива **системско тестирање** (енг. *system testing*). На овом нивоу се тестира и хардвер, а тестирање не обухвата само функционалност програма већ и безбедност, капацитет, перформансе, преносивост, итд. На овом нивоу се примењује **истраживачко тестирање** (енг. *exploratory testing*) током кога тестер извршава тест случајеве који нису били у плану, са намером да открије непредвиђене начине коришћења система. Обављају се и **тестови прихватљивости** (енг. *acceptance testing*) које извршавају корисници, проверавајући да ли софтвер испуњава њихова очекивања и потребе. Након измена система врши се **регресионо тестирање** (енг. *regression testing*) које треба да покаже правилан рад неизмењених функција. Такође треба да покаже и да су перформансе новог система макар једнаке перформансама старог система, а пожељно је да су боље. Сви ови нивои се примењују уколико време дозвољава темељно тестирање. У случају мањка времена, тестирање се прилагођава могућностима [6, 11].

3.2 Стратегије тестирања

На основу доступних информација о имплементацији софтвера који се тестира разликујемо три основне стратегије тестирања [24, 25, 11]:

Тестирање беле кутије (енг. *white box testing*) или *структурно* (енг. *structural testing*) тестирање подразумева познавање интерне структуре и имплементације програма. Тестерима је доступан целокупан изворни кôд, тако да је њихов посао да га детаљно изуче како би нашли делове који су подложни грешкама. То је мукотрпан посао јер подразумева пролажење кроз све линије кода којих често има превише за људску обраду. Због тога се углавном користе алати који скенирају кôд и региструју потенцијалне слабости програма. Након тога их проверава програмер и одлучује да ли су пронађене слабости заиста претња. Алати много помажу али не могу да замене знање и искуство стручњака.

Добра страна ове стратегије је што доступност изворног кода омогућава потпуну покривеност кода (можемо проћи кроз све путање кроз програм, кроз све гране, итд.). Ипак, велики број линија кода повлачи и велику сложеност његовог анализирања. Алати који анализирају кôд често окарактеришу пуно делова кода као потенцијалне слабости. Тада програмери морају проћи кроз дугачак извештај о њима, а углавном се испостави да је већина записаних слабости лажна узбуна. Такође, ова стратегија је неприменљива уколико изворни кôд није доступан. Треба имати у виду да је доступност кода уобичајена код програма на Linux оперативним системима, што није случај код програма писаних за Windows оперативне системе.

Тестирање црне кутије (енг. *black box testing*) или *функционално* (енг. *functional testing*) не користи никакве информације о интерној структури и имплементацији програма. Ова стратегија се углавном примењује када информације о унутрашњој структури нису доступне, али може бити корисна и када јесу доступне, поред тестирања беле кутије. Једине информације које се користе су улаз и излаз из програма. Тестирање се углавном заснива на претпоставкама о тестираном програму, његовим спецификацијама и налажењу прихватљивог броја репрезентативних тест примера. Овај процес се најчешће обавља аутоматизовано због непрактичности ручног тестирања.

Једна од предности ове стратегије је примењивост. Без обзира на доступне информације, тестирање црне кутије је увек могуће и једноставније од осталих стратегија јер не изучава специфичности одређеног програма.

Такође, због тога што се тестови не ослањају на унутрашњу структуру софтвера, исти тест можемо употребљавати више пута за различите програме сличне намене. Једноставност овог приступа има и своје мане. Пошто се тестирање врши на основу претпоставки, никада не можемо тачно проценити ефикасност тестирања и ниво покривености кода као код тестирања беле кутије. Такође, ова стратегија није погодна за комплексне нападе где је потребно извршавати тестове одређеним редоследом ради изазивања рањивог стања програма.

Тестирање сиве кутије (енг. *gray box testing*) представља комбинацију претходне две стратегије у којој у некој мери постоје информације о унутрашњој структури софтвера, али не као код тестирања беле кутије. Изворни кôд није доступан али се увид у структуру софтвера добија преко датотека компајлираног програма тј. његових бинарних извршних датотека. Како су ове датотеке нечитљиве за људе, на њих се примењује процес обрнутог инжењеринга (енг. *reverse engineering*). Обрнути инжењеринг не може да открије изворни кôд програма, али препознавањем одређених инструкција може да произведе формат који се налази између изворног кода и машинског кода. За разлику од бинарних датотека, овај формат је читљив за људе и омогућава анализу сличну као код структурног тестирања.

Тестирање сиве кутије је хибридно решење које се често користи као допуна чистом тестирању црне кутије. То је могуће јер су бинарне датотеке софтвера увек доступне (осим Веб сервера и апликација). Мана ове стратегије је велика сложеност обрнутог инжењеринга који захтева постојање богатих ресурса.

Свака од поменутих стратегија има своје предности и мане. Ни за једну се не може рећи да је генерално боља од других, јер је свака од њих погодна за откривање различитих врста слабости. Најбоље решење је применити више стратегија ради откривања што више грешака. Различитост стратегија илустрована је сликом 3.1.

3.3 Расплинуто тестирање

Слика 3.1: Разлике између стратегија тестирања

Глава 4

Закључак

Библиографија

[1] Description of race conditions and deadlocks. https://support.microsoft.com/en-us/help/317723/description-of-race-conditions-and-deadlocks.

[2] Martin Odersky: Biography and current work. https://people.epfl.ch/martin.odersky/bio?lang=en&cvlang=en.

[3] Akka Documentation: Version 2.5.13. https://doc.akka.io/docs/akka/current/index.html, 2011-2018.

[4] Paul N. Butcher. *Seven concurrency models in seven weeks: when threads unravel.* The Pragmatic Bookshelf, Dallas, Tex., 2014.

[5] Felice Cardone and J. Roger Hindley. History of lambda-calculus and combinatory logic. 2006.

[6] Lee Copeland. *A Practitioner's Guide to Software Test Design.* Artech House, Norwood, MA, U.S.A., 2004.

[7] David J. DeWitt, Samuel Madden, and Michael Stonebraker. How to Build a High-Performance Data Warehouse. http://db.csail.mit.edu/madden/high_perf.pdf.

[8] Marius Herring. Concurrency made easy with Scala and Akka. http://www.deadcoderising.com/concurrency-made-easy-with-scala-and-akka/, March 2015.

[9] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. 1973.

[10] Milena Vujošević Janičić. Programske paradigme: Funkcionalna paradigma. http://www.programskijezici.matf.bg.ac.rs/ppR/2018/predavanja/fp/funkcionalno_programiranje.pdf.

[11] Milena Vujošević Janičić. Verifikacija softvera: Dinamička analiza softvera. http://www.programskijezici.matf.bg.ac.rs/vs/predavanja/02_testiranje/02_testiranje.pdf.

[12] Milena Vujošević Janičić. *Automatsko generisanje i proveravanje uslova ispravnosti programa.* PhD thesis, Matematički fakultet, Univerzitet u Beogradu, 2003.

[13] Saša Malkov. Informacioni sistemi: Bezbednost.

[14] Saša Malkov. Pojam funkcionalnih programskih jezika. http://poincare.matf.bg.ac.rs/~smalkov/files/fp.r344.2017/public/predavanja/FP.cas.2017.01.Uvod.p2.pdf.

[15] Saša Malkov. Razvoj softvera: Konkurentnost.

[16] Bertrand Meyer. Seven Principles of Software Testing. *Computer*, 41:99–101, 8 2008.

[17] John C. Mitchell and Krzysztof Apt. *Concepts in Programming Languages*, chapter 4.2, pages 57–67. Cambridge University Press, 2003.

[18] Cristóbal A. Navarro, Nancy Hitschfeld-Kahler, and Luis Mateu. A survey on parallel computing and its applications in data-parallel problems using gpu architectures. *Communications in Computational Physics*, page 285:329, February 2004.

[19] Martin Odersky. Scala's Prehistory. https://www.scala-lang.org/old/node/239.html, 2008.

[20] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala.* Artima Press, Walnut Creek, California, 2 edition, 2011.

[21] Martin Odersky, Lex Spoon, and Bill Venners. Concurrency programming in Scala. In *Programming in Scala*, chapter 9. Artima Press, Walnut Creek, California, 2016.

[22] Ben Stopford. Shared Nothing v.s. Shared Disk Architectures: An Independent View. http://www.benstopford.com/2009/11/24/understanding-the-shared-nothing-architecture/, 11 2009.

[23] Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal, C/C++ Users Journal*, December 2004.

[24] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Pearson Education, One Lake Street, Upper Saddle River, NJ 07458, 2007.

[25] Ari Takanen, Jared DeMott, and Charlie Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, 685 Canton Street Norwood, MA 02062, 2008.

Биографија аутора

Вук Стефановић Караџић (*Тршић, 26. октобар/6. новембар 1787. — Беч, 7. фебруар 1864.*) био је српски филолог, реформатор српског језика, сакупљач народних умотворина и писац првог речника српског језика. Вук је најзначајнија личност српске књижевности прве половине XIX века. Стекао је и неколико почасних доктората. Учествовао је у Првом српском устанку као писар и чиновник у Неготинској крајини, а након слома устанка преселио се у Беч, 1813. године. Ту је упознао Јернеја Копитара, цензора словенских књига, на чији је подстицај кренуо у прикупљање српских народних песама, реформу ћирилице и борбу за увођење народног језика у српску књижевност. Вуковим реформама у српски језик је уведен фонетски правопис, а српски језик је потиснуо славеносрпски језик који је у то време био језик образованих људи. Тако се као најважније године Вукове реформе истичу 1818., 1836., 1839., 1847. и 1852.