

УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ

Ана Д. Митровић

**ПРИМЕНА СКАЛЕ У
ПАРАЛЕЛИЗАЦИЈИ РАСПЛИНУТОГ
ТЕСТИРАЊА**

мастер рад

Београд, 2019.

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: 2019.

Садржај

| | | |
|----------|---|-----------|
| 1 | Увод | 1 |
| 2 | Програмски језик Скала | 3 |
| 2.1 | Опште карактеристике | 4 |
| 2.2 | Функционални део језика | 6 |
| 2.3 | Паралелизам | 12 |
| 2.3.1 | Модел дељења меморије | 14 |
| 2.3.2 | Модел размене порука | 15 |
| 2.4 | Библиотека <i>Akka</i> | 17 |
| 2.4.1 | Структура | 18 |
| 3 | Расплинуто тестирање | 24 |
| 3.1 | Основе тестирања софтвера | 25 |
| 3.1.1 | Нивои тестирања | 26 |
| 3.1.2 | Стратегије тестирања | 27 |
| 3.2 | Основе расплинутог тестирања | 29 |
| 3.2.1 | Историја | 31 |
| 3.2.2 | Ограничења | 32 |
| 3.3 | Врсте програма расплинутог тестирања | 33 |
| 3.4 | Расплинуто тестирање формата датотека | 36 |
| 4 | Структура датотека у формату <i>PDF</i> | 40 |
| 4.1 | Објекти датотека | 40 |
| 4.2 | Организација датотека | 42 |
| 4.3 | Ажурирање датотека | 44 |
| 5 | Имплементација програма за расплинуто тестирање читача датотека у формату <i>PDF</i> | 46 |

| | | |
|----------|--|-----------|
| 5.1 | Улазни параметри | 47 |
| 5.2 | Организација програма | 48 |
| 5.3 | Објекти Актер | 48 |
| 5.4 | Креирање датотека у формату <i>PDF</i> | 52 |
| 5.5 | Резултати тестирања | 57 |
| 6 | Закључак | 62 |
| | Библиографија | 64 |

Глава 1

Увод

У области рачунарства напредак нових технологија је веома брз због потреба за све захтевнијим системима који су присутни у нашем свакодневном животу. Константно се развијају и софтвер и хардвер који омогућавају развој сигурнијих, бржих и ефикаснијих апликација разних намена. Неке од најважних карактеристика данашњих рачунара су конкурентно и паралелно извршавање процеса. Данас се углавном програмира и ради на вишејезгарним процесорима, који могу значајно да допринесу побољшању перформанси програма.

Функционална парадигма програмирања има особине које су погодне за паралелно програмирање. Дељење меморије међу нитима и руковање променљивим подацима изискују синхронизацију и закључавање, што је подложно грешкама. Транспарентност референци и коришћење непроменљивих података су особине функционалних програмских језика које олакшавају паралелизацију задатака. Програмски језик Скала је објектно оријентисан језик који подржава функционално програмирање, због чега може искористити предности функционалне парадигме. Скала искоришћава и предности објектне парадигме, па је погодна за програме различитих намена. Поред тога, компатибилна је са Јавом што омогућава лакше привикавање програмера на нови језик.

Проблеми модела дељења меморије могу се превазићи другачијим стилем паралелног програмирања. У раду је обрађен модел размене порука под називом Актер. Коришћена је библиотека *Akka*, која је заједничка за Јаву и Скалу. Као задатак који треба паралелизовати одабрана је техника тестирања софтвера под називом расплинуто тестирање¹. Ова техника се базира на генерисању великог броја правилних и неправилних улаза у програм како би што

¹Термин „расплинуто” се не односи на расплинуте (енг. *fuzzy*) логике и скупове.

више различитих случајева било испитано. Због генерисања различитих улаза, расплинуто тестирање је веома погодно за примену паралелизације тако што свака компонента тестирања паралелно може генерисати улаз за програм. У овом раду су тестирани читачи датотека у формату *PDF*, тако да су улази у програм такве датотеке.

Још један разлог одабира технике тестирања за задатак који треба паралелизовати је велики значај тестирања софтвера. Поред саме имплементације програма јако је важно тестирати програм. Иако само тестирање не може доказати исправност софтвера, оно игра велику улогу у проналажењу и исправљању грешака чиме се повећава поузданост софтвера. Због тога су у раду обухваћене и основе тестирања софтвера.

Рад је организован на следећи начин. У глави 2 детаљно је описан програмски језик Скала. Описане су његове опште карактеристике (поглавље 2.1) са посебним акцентом на функционални део језика (поглавље 2.2). Детаљно је описан концепт паралелизма (поглавље 2.3) и библиотека *Akka* (поглавље 2.4). У глави 3 је описана техника расплинутог тестирања. Поглавље 3.1 садржи основе тестирања софтвера, а поглавље 3.2 основе расплинутог тестирања. Врсте програма расплинутог тестирања описане су у поглављу 3.3, а расплинуто тестирање формата датотека описано је у поглављу 3.4. Структура датотека у формату *PDF* описана је у глави 4. Описани су објекти датотека (поглавље 4.1), организација датотека (поглавље 4.2) и њихово ажурирање (поглавље 4.3). У глави 5 описана је имплементација програма за расплинуто тестирање. Описани су улазни параметри програма (поглавље 5.1), организација програма (поглавље 5.2), објекти Актер (поглавље 5.3), креирање улазних датотека (поглавље 5.4) и постигнути резултати (поглавље 5.5). У глави 6 дати су закључци рада.

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 [29].

Творац Скале је Мартин Одерски (нем. *Martin Odersky*) (1958-), немачки научник и професор на универзитету *EPFL* (École Polytechnique Fédérale de Lausanne) у Лозани у 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 код 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 компајлера *GJ* (*Generic Java*) који је постао основа компајлера *javac* [3, 28].

2.1 Опште карактеристике

Прилагођавање потребама система је у Скали лако и веома удобно. Томе доприноси могућност програмера да дефинише библиотеке које су веома лаке за коришћење и које се користе једнако интуитивно као и библиотеке које се већ налазе у самом језику. У наставку су наведене неке од најважнијих особина Скале које доприносе њеној „скалабилности” [29]:

Компатибилност са Јавом Скала није сама по себи продужење Јаве, али је потпуно компатибилна са њом: њен изворни код се преводи у Јава бајт код који се извршава на Јава виртуелној машини. У коду написаном у Скали је омогућено коришћење библиотека, класа, интерфејса, метода, поља и типова језика Јава. Омогућено је и обрнуто, позивање кода написаног у Скали из Јаве, мада се оно ређе користи. Такође, у Скали је омогућена лакша и лепша употреба типова језика Јава уз помоћ имплицитне конверзије. Тиме је омогућена употреба метода за манипулацију типовима које има Скала. 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 Актер 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” [29]. Елегантан начин коришћења метода налик на операције је једна од особина Скале које је чине пријатном за коришћење.

Концизност Кôд написан у Скали има тенденцију да буде краћи од кода написаног у Јави. Чак се процењује да Скала 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. 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 имплементирају и касније употребљавају библиотеке које сами напишемо. Такође, у Скали је подржано аутоматско закључивање типова (енг. *type inference*), што омогућава изостављање навођења већ наведених типова, што резултује читљивијим кодом.

Висок ниво Скала је погодна за велике и комплексне системе и може се прилагодити њиховим захтевима подижући ниво апстракције у коду. Омогућен је 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. Кôд написан у Скали је разумљивији и мање комплексан, што је важно јер је систем који се имплементира сам по себи комплексан.

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. Довољно је навести тип само једном уместо више пута. Понављање нарушава читљивост кода, али је ипак у неким језицима неопходно. Због оваквих понављања, многи се одрекну предности статички типизираних језика као што су: детектовање разних грешака у време компилације, лакше рефакторисање кода и коришћење типова као вида документације.

Способности Скала уводи појам способности (енг. *trait*) које деле карактеристике са интерфејсима и наслеђивањем класа у Јави. У способностима као и у интерфејсима можемо декларисати методе, али и дефинисати поља и конкретне методе што није могуће у интерфејсима. Променљиве омогу-

ћавају чување стања а методе дефинисање одређеног понашања. То чини способности богатијим од интерфејса. Кажемо да се способности „спајају” у класе (енг. *mix in*). У једну класу је могуће спојити више способности. Спајање се може остварити кључном речју **extends** и у том случају класа наслеђује наткласу способности. У случају да желимо да класа наследи неку другу класу, онда помоћу *extends* наводимо наткласу а способности спајамо помоћу **with** као што је илустровано у наредном примеру [29]:

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

Највећа разлика између класе и способности односи се на позивање метода наткласе помоћу поља *super*. На пример, позивом метода *super.toString()* у случају класе, тачно се зна који ће метод *toString()* бити позван јер се зна наткласа дате класе. У случају способности, не можемо знати на шта се овај позив односи јер то директно зависи од класе у коју ће дата способност бити спојена, као и од претходно спојених способности. То значи да се позиви *super* одређују *динамички*. Свака спојена способност може позвати метод претходно спојене способности. Ово омогућава класама да стекну различито понашање у зависности од редоследа спојених способности. Помоћу малог броја дефинисаних способности може се постићи много различитих циљева комбиновањем редоследа спајања.

Иако је Скала објектно оријентисани језик, она има елементе функционалне парадигме и подржава чисто функционално програмирање. Функционална природа Скале је детаљно објашњена у следећем поглављу.

2.2 Функционални део језика

Функционална парадигма се развија од 1950-тих година. Чисто функционални програмски језик може да се посматра и као математичка формализа-

ција алгоритма (може и обрнуто). Примери таквих формализама су *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 експресивност 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 [18, 22, 26, 10].

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#*, *Python* [18, 10].

Разлика између императивне и функционалне парадигме се може описати разликом у начину дефинисања појма програма и начину писања програма [22]:

- 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 математичком начину размишљања због чега се и називају *функционалним* језицима: њихове функције се понашају као функције у математичком смислу. О томе говоре следеће две карактеристике функционалних језика [29]:

Прва карактеристика се односи на „**статус**” **функција**. У функционалним језицима функције су вредности *првог реда*, тј. *грађани првог реда*. То значи да се променљиве чије су вредности функције могу користити као променљиве других типова као што су нпр. цели, реални бројеви, ниске карактера итд. 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 (угњежене) 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 и прослеђиване као обичне вредности. Функцијски литерали се у другим језицима називају „анонимним” или чешће „ламбда” функцијама. 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 [29]:

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

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

Друга карактеристика 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. Уместо променљивих чија се стања ажурирају, користе се *именоване вредности*. Именоване вредности се иницијализују само једном, а у Скали се дефинишу помоћу кључне речи **val**. Покушај промене именоване вредности након што је иницијализована резултирао би грешком. Поред именованих вредности, у Скали је дозвољено

коришћење променљивих јер она није чист функционални језик. Променљиве се дефинишу кључном речју **var**.

Непостојање стања има за последицу избацавање 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. Проблеми који захтевају чување и преношење променљивог стања некада имају компликовано решење у оваквим језицима. Ипак, решење постоји 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 [18]. Такође треба напоменути да је употреба рекурзије уобичајена за класичан приступ функционалном програмирању. Са друге стране, савремен приступ тежи да све рекурзије сведе на употребу стандардизованих функција вишег реда.

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 ниску 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 **транспарентност референци** (енг. *referential transparency*): позив метода можемо заменити његовим резултатом јер смо сигурни да ће при сваком позиву са истим вредностима аргумената вратити исти резултат. Пример:

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

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

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 па може бити произвољан. За овакве методе кажемо да су референцијално транспарентни. Они су концизнији, читљивији и производе мање грешака јер немају бочне ефекте. Ипак, у Скали су неопходни и бочни ефекти јер она није функционалан језик [18].

Чисто функционални језици (нпр. *Haskell*, *Miranda*) захтевају коришћење непроменљивих структура података, референцијално транспарентних метода и рекурзије. Други језици, као што су Скала, *Python*, *Ruby* 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 [18, 29].

Неке од карактеристика које Скала подржава а које су заједничке и другим функционалним језицима су [29]:

Функције вишег реда Функције вишег реда су оне које имају друге функције као своје аргументе. Оне поједностављују код 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) = {
    val filesHere = (new java.io.File(".")).listFiles
    for (file <- filesHere; if matcher(file.getName, query))
        yield file
}
```

Функција *filesMatching* треба да међу датотекама *filesHere* пронађе датотеке које испуњавају неки критеријум. Да се код 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 стандардне библиотеке Скале:

```
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 су функције које уместо једне листе аргумената имају више листи које садрже по један аргумент. То омогућава 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 datoteka */  
    if file.getName.endsWith(".scala") /* prvi uslov */  
    line <- fileLines(file) /* iteriranje kroz linije datoteke */  
    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 датотека помоћу услова задатог у *if* изразу. Онда следи једна угњеждена петља где се врши итерација кроз линије тренутне датотеке (која је испунила услов, а ако није, прелази се на следећу датотеку). 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 стално расте у складу са захтевнијим апликацијама: „Направи десет пута бржи процесор, и софтвер ће убрзо наћи десет пута више посла” [34]. Потребно је све брже и ефикасније извршавати захтевна израчунавања. Зато су годинама откривани и развијани разни начини побољшања перформанси процесора. Неки од начина побољшања су повећавање брзине откуцаја часовника, оптимизација тока извршавања и повећавање

простора кеш меморије на чипу. Ова побољшања се односе и на секвенцијалне као и на конкурентне апликације. Ипак, ови начини подизања перформанси полако достижу свој максимум. Због тога се развијају нове идеје и данас се напредак процесора све више односи на број *централних процесорских јединица* тј. *језгара*. Процесори који имају више од једног језгра се називају *multi-core* процесори. Сматра се да је конкурентност нова главна револуција у писању софтвера [34, 27].

Програм се сматра **конкурентним** уколико може да има више од једне активне нити извршавања (енг. *thread*). Нит извршавања представља компоненту процеса која се извршава секвенцијално. Другим речима, конкурентност значи да се може десити да су различите нити истовремено у току. Ако су нити истовремено у току, то не значи да су обавезно истовремено физички активне. Можемо имати више нити којима распоређивач (енг. *scheduler*), као нпр. Јава виртуелна машина, наизменично додељује „парчиће” времена (енг. *time-slice*). Због овога имамо привид да се нити извршавају дословно истовремено. Ипак, истовремено извршавање захтева више од једног процесорског језгра. Уколико је то испуњено, нити извршавања могу да буду истовремено физички активне, па се конкурентан програм сматра **паралелним**. Као и нити, и процеси се могу извршавати и конкурентно и паралелно.

До појаве нити је дошло због високе цене промене контекста (енг. *context switch*). Промена контекста 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 при конкурентном извршавању процеса дешава јако често, до неколико стотина хиљада пута у секунди по процесорском језгру. То је незгодно јер узима доста процесорског времена. Такође, подаци о стању процеса заузимају много меморије. Нити могу делити податке између себе, тако да је конкурентно извршавање нити једног процеса ефикасније од конкурентног извршавања више процеса. Прелазак са једне на другу нит истог процеса је бржи од преласка са једног на други процес. То доводи до уштеде меморије и времена [32, 24, 27].

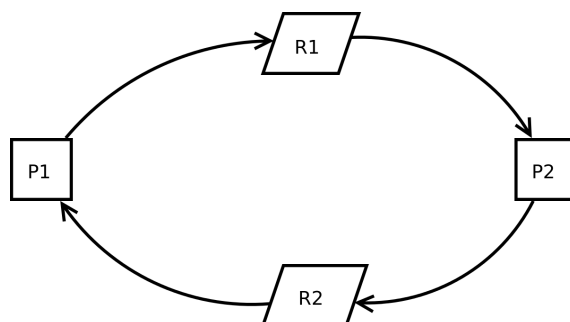
Постоје различити модели тј. стилови паралелног програмирања. Ови стилови говоре о начинима комуникације процесорских језгара, нити, и начинима на који је то остварено. У наставку су објашњена два модела, модел дељења меморије и модел размене порука [27].

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 приступају заједничким меморијским локацијама. 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 техникама закључавања као што су мутекси, катанци, монитори, семафори. Сврха ових техника 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 [30, 27].

Механизми закључавања решавају проблеме који се тичу приступа подацима, али уводе нове проблеме [30, 2]:

- **Број катанаца** Закључавање и откључавање катанаца представљају скупе операције. Због тога велики број катанаца може да успори рад програма. Са друге стране, мали број катанаца може значити да они закључавају велику количину података, што доводи до дужег чекања па и споријег рада.
- **Надметање око ресурса** Надметање око ресурса (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. То је разлог увођења синхронизације нити.
- **Мртва петља** 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 2.1: Процес P1 тражи ресурс R1 али не успева да му приступи јер процес P2 држи ресурс R1. Слично, процес P2 тражи ресурс R2 али њега држи процес P1. Два процеса су у застоју јер



Слика 2.1: Мртва петља

се међусобно чекају. Проблем мртве петље је последица лоше технике закључавања. Може се решити тражењем катанаца у унапред дефинисаном редоследу, али и даље представља оптерећење за програмере.

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

Писање конкурентних програма је јако тешко. Постоји мало програмера који су у стању да напишу конкурентан програм без грешака, 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. Веома је компликовано доказати да је програм *коректан* тј. да ради управо оно што се од њега очекује. 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 програма. То су разлози због којих је функционална парадигма погоднија од других за паралелизацију. Чим нема променљивих структура нема ни проблема које оне носе са собом [30].

2.3.2 Модел размене порука

Модел размене порука (енг. *message passing model*) је један од популарних „трендова“ који се тичу конкурентности. Овај модел подржава архитектуру **SN** тј. архитектуру *Shared Nothing*. Архитектура *SN* 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 посла. Због тога нема потребе за механизмима закључавања. Како се међу чворовима ништа не дели, отуда и назив ове архитектуре [33, 13].

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. Поруке које се шаљу су непроменљиве и могу се слати синхронно и асинхронно [29].

Модел размене порука се може имплементирати на више начина, а најуспешнији начин је помоћу такозваног модела Актер (енг. *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 Актер. 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 модела Актер представљају објекти које зовемо *Актери*. Они формирају хијерархијску структуру у којој сваки објекат има свог родитеља који сноси одговорност за њега. Сваки Актер 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 Актер од других објеката је способност реаговања на поруке одређеном акцијом. Као одговор на поруку, Актер може направити још компонената Актер (своју децу), слати поруке другим компонентама или зауставити рад своје деце или себе. Актер има своје локално стање које је променљиво, али са другима не дели ништа осим порука. Важна особина Актера је да се не блокирају када пошаљу поруку већ настављају са радом, односно слање порука је увек асинхронно.

Предности модела Актер и уопште архитектуре *SN* се огледају у избегавању свих поменутих проблема конкурентног програмирања, чији највећи узрок представљају дељени променљиви подаци. Модел Актер је погодан у апликацијама у којима је могуће посао поделити на што више мањих, независних послова. Тада сваки мањи посао представља задатак једне компоненте

Актер. 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 модел Актер 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. Модел Актер 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 [30]:

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, тј. о поруци којом је започета комуникација међу компонентама. 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. Модел Актер енкапсулира рад са нитима чиме се налази на нивоу изнад њих. Боље перформансе програма се могу постићи директним радом са нитима уместо са моделом Актер.

Разни језици имају библиотеке које имплементирају модел Актер. У следећем поглављу је обрађена одговарајућа библиотека у Скали [30, 16, 9].

2.4 Библиотека *Akka*

Akka представља имплементацију модела Актер на Јава виртуелној машини. Од верзије 2.10 језика Скала ова библиотека је подразумевана библио-

тека за коришћење модела Актер. Постојала је и библиотека *Actor*, али је она застарела и од верзије 2.11 се више не користи [15].

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

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

Кључне особине које одликују компоненте Актер су слање и примање порука. То се ради на следећи начин [30]:

- Слање порука се врши позивом метода !:

```
a ! msg
```

Објекту *a* се шаље порука *msg*.

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

```
receive {  
  case pattern1 =>  
    ...  
  case patternN =>  
}
```

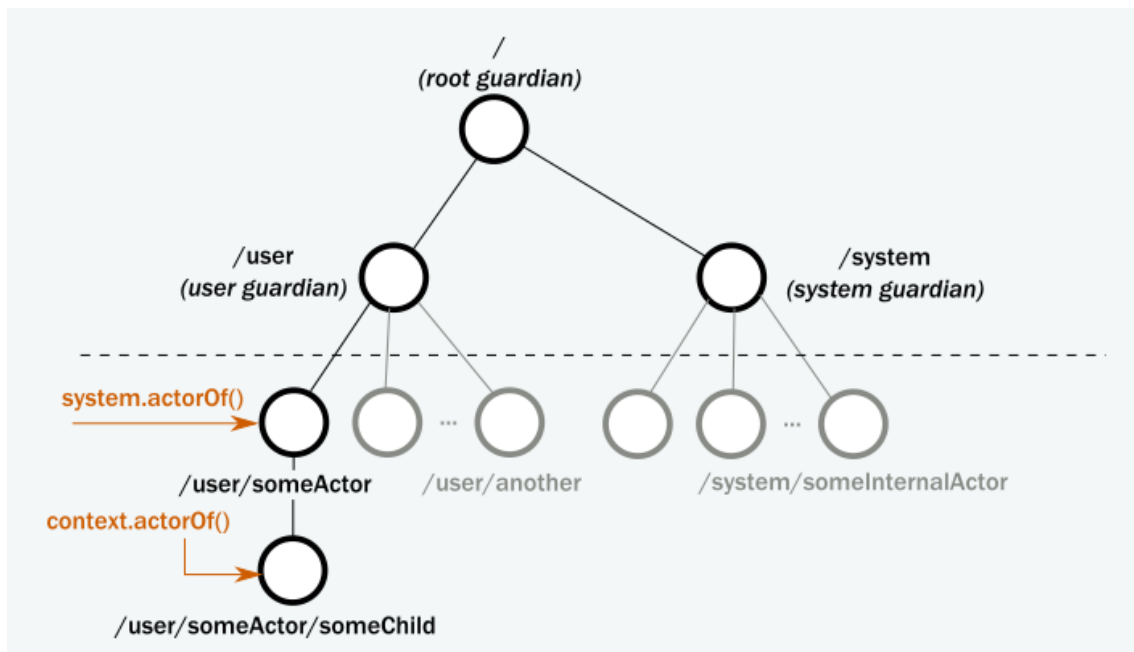
Све компоненте система Актер су хијерархијски распоређене и организоване тако да се свакој приступа на јасно дефинисан начин. То је описано у следећем поглављу.

2.4.1 Структура

Све компоненте Актер 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 [30, 6].

Све компоненте Актер једног система *ActorSystem* су смештене у **стабло** које је приказано на слици 2.2. Свака компонента има родитеља коме припада. На самом врху хијерархије налазе се три предефинисане компоненте [6]:

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



Слика 2.2: Стабло Actor компонената

Оваква хијерархијска структура је слична структури система датотека. Прву компоненту правимо помоћу метода *actorOf* самог система *ActorSystem*. Све компоненте направљене на овај начин ће постати деца предефинисане компоненте *user guardian*. Овакве компоненте су на врху хијерархије корисничких компоненти, па за њих кажемо да су на највишем нивоу иако постоје предефинисане компоненте изнад њих. Компонента Актер која произведе нову компоненту представља њеног родитеља (енг. *parent actor*), а нова компонента је њено дете (енг. *child actor*). Ова радња се постиже методом *actorOf* атрибута

context компоненте Актер. Овај атрибут 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 компоненте Актер 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 атрибута *self*. Слично, свака компонента преко атрибута *sender* има приступ референци на компоненту која јој је послала поруку.

У систему *ActorSystem* свака компонента Актер има путању **ActorPath** која је јединствено идентификује. Различите компоненте се могу налазити на различитим чворова на мрежи тј. извршавати се на различитим Јава виртуелним машинама. Због тога први део путање садржи протокол и локацију на којој се налази компонента (идентификацију система *ActorSystem* у коме се налази). Надаље путању чине надовезани елементи у стаблу раздвојени са „/”, почевши од корена. На пример [6]:

```
"akka://my-sys/user/service-a/worker1"    // локална компонента
"akka.tcp://my-sys@host.example.com:5678/user/service-b" // удаљена
компонента
```

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 датотека је начин на који претражујемо њене елементе. На пример [6]:

```
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 (енг. *wildcard*), добијамо сву децу свог родитеља, укључујући и себе. На овај начин лако шаљемо поруку различитим компонентама истовремено.

Сваки систем *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 блока *receive* компоненте Актер. 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 компоненту Актер. 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 компоненте Актер докле год те податке не делимо [30, 6].

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 [6].

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.

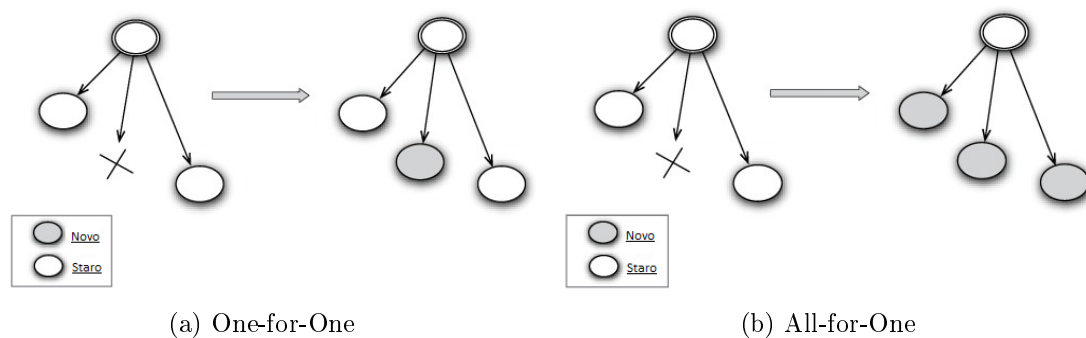
Други разлог постојања хијерархије је управљање грешкама и изузецима. *Akka* подржава програмирање у коме се стање грешке посматра као било које друго валидно стање. Немогуће је спречити све грешке, тако да је циљ при-

премити 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, које 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 [30, 6]:

- 1) Настављање са радом са акумулираним стањем
- 2) Трајно заустављање детета
- 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. Стратегије су илустроване на сликама 2.3a и 2.3b [30, 6]:

- **One-for-One** је подразумевана стратегија у којој током рестартовања одређене компоненте, друге компоненте истог родитеља 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.



Слика 2.3: Стратегије рестартовања

Глава 3

Расплинуто тестирање

Грешке су саставни део сваког посла. Развојем софтвера развијају се нове врсте пропуста. „Постоје стотине слабости софтвера које само чекају да буду откривене, и биће откривене када прође довољно времена” [36]. Неке грешке немају велики значај, док друге могу оставити озбиљне последице. Постоје програми који користе грешке других програма уметањем вируса, црва, злонамерних скриптова и слично. Циљ напада на софтвер 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 могу угрозити квалитет и безбедност система. [36, 23].

Тестирање се може применити на разне аспекте система, због чега постоје различите врсте тестова. На пример, неким тестовима је циљ провера тачности израчунавања вредности израза или функције. Другачијом врстом тестова можемо проверити на који начин независне јединице система раде једна са другом, итд. Због тога је важно напоменути да је у овом раду фокус на тестирању робу-сности софтвера. Циљ тестирања је откривање неисправног или непредвидивог понашања програма у случају неисправних улазних података.

Расплинуто тестирање (енг. *fuzz testing/fuzzing*) је приступ тестирању софтвера који се бави генерисањем великог броја улаза у програм, након чега за сваки од улаза посматра његов ток извршавања. Циљ ових тестова је да открију необичне грешке и изузетке у програму. Улази треба да буду неочекивани и генеришу се или потпуно случајно или уз коришћење одређених *хеуристика*. Узимају се у обзир и неисправни улази јер је важно да програм препозна и одбаци овакве улазе без прекида или отежавања даљег рада.

3.1 Основе тестирања софтвера

Тестирање софтвера представља најчешћи вид верификације софтвера. *Верификација* софтвера се бави испитивањем исправности програма тј. провером жељеног понашања програма задатог *спецификацијом*. Постоје *статичка* и *динамичка* верификација. Статичка испитује исправност програма без извршавања кода тј. његовом анализом, док динамичка подразумева испитивање извршавањем кода [20].

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 тестирање као процес који покушава да открије грешке у програму посматрајући његово извршавање у контролисаним условима [25].

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 грешака, али никада њихово одсуство” [25]. 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 изграђеног система.

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. Редослед је важан јер постоје случајеви када се одређени тест примери не могу извршити пре извршавања неке друге радње. Рецимо, не можемо тестирати брисање податка

из базе података ако је она празна тј. ако претходно није успео тест уметања податка у базу. Након извршавања тестова прегледају се 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 али не мора да ради у договору са програмером [12, 19].

„Покривеност кода (енг. *code coverage*) тестовима је однос броја неких елемената програма испитаних тестовима у односу на укупан број тих елемената” [19]. 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 висок 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 [19].

3.1.1 Нивои тестирања

Тестирање се врши на више нивоа, у зависности од сложености компоненте 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 компоненти, 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 да 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 [12, 19].

3.1.2 Стратегије тестирања

На основу доступних информација о имплементацији софтвера који се тестира разликујемо три основне стратегије тестирања [35, 36, 19]:

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 код, 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 код 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 високу покривеност кода (можемо проћи кроз велики број путања кроз програм, кроз много грана, итд.). Ипак, велики број линија кода повлачи и велику сложеност његовог анализирања. Алати који анализирају код 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 непримењива уколико изворни код 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 може применити када информације о унутрашњој структури нису доступне, али је 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.

Једна од предности ове стратегије је примењивост. Без обзира на доступне информације, тестирање црне кутије је увек могуће. Погодно је јер га могу примењивати и тестери који нису програмери. 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. Једна могућност је да се тестови базирају на обема техникама беле и црне кутије. Друга могућност је да у некој мери постоје информације о унутрашњој структури софтвера, али не као код тестирања беле кутије. Изворни код 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 код 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 може искористити предности стратегија црне и беле кутије. Уколико изворни код није доступан

а бинарне датотеке јесу, често се користи обрнути инжењеринг као допуна чистом тестирању црне кутије. 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.



Слика 3.1: Разлике између стратегија тестирања

3.2 Основе расплинутог тестирања

Развојем расплинутог тестирања је откривено доста начина генерисања улаза који у великом броју случајева откривају грешке. Најчешће се улази или делови улаза генеришу случајно, праве се улази који су мањи него што би требало (нпр. датотека са мање података), користе се негативне и граничне вредности нумеричких типова података, предугачке ниске карактера, ниске карактера са специјалним карактерима, замењују се суседни битови, итд. Не постоје правила која дефинишу радње које се примењују на подацима, већ само савети до којих се дошло дугогодишњим испробавањем.

Савети који помажу при генерисању тест примера представљају *хеуристике*. Хеуристика је приступ решавању проблема који прави изборе на основу искуства, тј. изборе који ће највероватније довести до решења проблема. То су избори за које не постоји доказ да доводе до решења, али се у пракси показало да дају добре резултате. Хеуристике се користе у случају да је немогуће или

је јако тешко испитати све могуће случајеве неког проблема. Зато се испитају познати случајеви који углавном у кратком временском периоду нађу решење. Решење често не буде најбоље могуће решење, али буде прихватљиво [31].

Кључна предност расплинутог тестирања наспрам других техника тестирања је могућност аутоматизације овог процеса. Ручно креирање тест примера и покретање програма је веома напорно, споро, неефикасно и захтева да га обавља стручњак у области тестирања. Зато се прибегава аутоматизацији што је могуће већег дела посла које тестирање захтева. Тестирање може покренути особа без великог знања о томе. Такође, многе апликације за тестирање су примењиве на велики број различитих програма [35, 36].

Расплинуто тестирање је један од приступа тестирања црне кутије. Програм за тестирање не зна ништа о структури програма који тестира. Проблем који се јавља оваквим тестирањем је тај што формирани улази често откривају грешке које је лако открити. То су грешке до којих долази извршавањем кода који се најчешће извршава. Са друге стране, семантичке грешке које се налазе дубоко у програму остају неоткривене. Зато су развијени приступи алтернативног расплинутог тестирања који одговарају тестирању црне, беле и сиве кутије [36, 8, 14]:

Расплинуто тестирање црне кутије (енг. *black box fuzzing*) је приступ на који се обично мисли под појмом „расплинуто тестирање”. Он је најлакши за имплементацију и омогућава тестирање великог броја тест примера за кратко време. Показује се да је често ефикаснији од других приступа уколико њима треба превише времена за генерисање тест примера.

Расплинуто тестирање беле кутије (енг. *white box fuzzing*) пре самог тестирања користи технике које идентификују делове програма који могу да имају грешке. На тај начин ова врста тестирања повећава покривеност кода, открива грешке до којих се теже долази него другим врстама тестирања, али је и компликованија од њих.

Расплинуто тестирање сиве кутије (енг. *gray box fuzzing*) је приступ који има предности претходне две технике а труди се да избегне њихове мане. Тежи се да ефикасност буде што ближа ефикасности приступа црне кутије, али ова техника не генерише улазе „на слепо” већ користи неки паметнији начин, као приступ беле кутије. Уместо анализе кода, користе се технике инструментације да се добије увид у структуру програма.

3.2.1 Историја

Примена тестирања налик на расплинуто је почело још 1980-их година са алатом званим „мајмун” (енг. *The monkey*). Он је развијен за тестирање програма на *Macintosh* рачунарима, који су имали проблема због мањка меморије. Мајмун је симулирао правог мајмуна који неартикулисано удара у тастатуру и миш рачунара и на тај начин тестира рад покренутог програма. Ипак, до 1990-их су интересовања и потребе за тестирањем биле много мање него данас, највише због недостатка интернета и идеја напада на други софтвер. Истраживање безбедности софтвера је почело крајем 1980-их година, када су почели да се развијају напади који се односе на прекорачење бафера. Један од познатих напада је настао 1988. године и назива се „Morris Internet Worm” [35, 36].

Први пројекат развијен за тестирање је написан 1989. године. Идеју за овај пројекат је добио професор Бартон Милер (енг. *Barton Miller*), за кога неки сматрају да је „отац” расплинутог тестирања. Заједно са својим ученицима са курса о оперативним системима тестирао је квалитет и поузданост *UNIX* апликација. Програм је случајно генерисао ниске карактера и прослеђивао их апликацијама. Иако је тестирање имало веома једноставну идеју, у то време је то био велики помак јер се о расплинутом тестирању знало јакo мало. На Универзитету у Оулу у Финској је 1999. године почело развијање пакета тестова под називом *PROTOS*. Ови тестови су проверавали безбедност разних протокола. Компанија *Microsoft* је 2002. године новчано помогла овој иницијативи тако да је тим који је развијао *PROTOS* 2003. године засновао компанију за производњу пакета тестова, под именом *Codenomicon*. Након тога, објављен је програм за тестирање отвореног кода имена *SPIKE*. Он је био напреднији од програма који је развио професор Милер. Брзо је почело и развијање других пројеката попут *sharefuzz*, *mangleme*, *Hamachi*, *CSSDIE*, *FileFuzz*, *SPIKEfile*, *notSPIKEfile*, итд. Компанија *Mu Security* је 2005. године почела да развија хардверски уређај коме је циљ мењање података који се шаљу протоколима путем мреже. Контроле *ActiveX* су такође постале мета програма *COMRaider* и *AxMan* 2006. године [35, 36]. Временом је потреба за сигурношћу софтвера постајала све већа, тако да су се развијали алати за најразличитије врсте апликација. Методе тестирања, као и апликације које врше тестирање, се и данас развијају и побољшавају. Доступан је велики број алата који примењују најновије технике тестирања и врше разне оптимизације ради веће ефикасности. Оптимизације омогућавају краће време и цену тестирања, једноставност кори-

шћења, минималну људску интеракцију, итд. Неки од алата имају специфичну примену, док су други употребљиви у свакој врсти софтвера. Неки од познатијих алата су *AFL*, *Project Springfield*, *OSS-Fuzz*, *libFuzz*, *BFuzz*, *Fuddly*, *Honggfuzz*, *Peach*, *Radamsa*, итд [1, 35, 36].

3.2.2 Ограничења

Различите врсте тестирања откривају различите врсте слабости. Не постоји универзална техника за откривање свих грешака, већ у зависности од потреба бирамо одређену технику. Неке од врста проблема које расплинато тестирање обично не открива су [35]:

Права приступа Неки програми разликују обичне од привилегованих корисника који имају више права од обичних (рецимо поред читања података могу да их бришу или мењају). Програм за расплинато тестирање не зна ништа о логици програма који тестира, због чега не разликује типове корисника. Извршавање недозвољених функција од стране обичног корисника ће проћи неопажено код програма за расплинато тестирање. Могуће је уметнути логику програма у програм за тестирање, али је то веома сложен процес.

Дизајн програма Неки програми имају мане које се односе само на јако лоше осмишљен дизајн. Пример је апликација која дозвољава свим корисницима приступ одређеним подацима. Велика је грешка не узимање у обзир злонамерне кориснике. Проблем је непостојање аутентикације и ауторизације. Програм за расплинато тестирање не може да препозна овакав безбедносни проблем.

Безбедност меморије Једна од врста напада на софтвер може бити приступање недозвољеној меморији. Читањем и писањем на разне меморијске локације нападач може да стекне одређену контролу над програмом. Недозвољен приступ меморији може да се препозна на нивоу инструкције машинског кода, након чега се програму пошаље сигнал *SIGSEGV*. Проблем је у томе што неретко програми сами решавају овај проблем тако што „ухвате” сигнал и реагују одређеном акцијом. Реаговање на погрешан начин може бити опасно, као нпр. игнорисање сигнала. Програм за

расплинуто тестирање често не примети овакву врсту слабости јер је она решена (правилно или неправилно) од стране самог програма.

Сложени напади Расплинуто тестирање је корисно за детекцију индивидуалних слабости. Међутим, није се добро показало у случају сложених напада који искоришћавају више слабости истовремено.

3.3 Врсте програма расплинутог тестирања

Програми расплинутог тестирања се разликују по разним карактеристикама. Две основне поделе ових програма се односе на начин на који генеришу тест примере и на тип самог програма који тестирају.

Начин генерисања тест примера има два основна критеријума. Први критеријум представља постојање почетних тест примера од којих се генеришу нови тест примери [35, 36]:

- **Тестери засновани на мутацијама** (енг. *mutation-based fuzzers*) користе постојеће тест примере за генерисање нових, примењујући измене тј. мутације на улазним подацима. Квалитет тестирања зависи од квалитета постојећих тест примера. Ови тестери су обично тестери опште намене.
- **Тестери засновани на генерацијама** (енг. *generation-based fuzzers*) генеришу тест примере „од нуле” тј. независно од постојећих. Једине информације које могу да користе су информације о формату који тест примери треба да задовоље (нпр. формат датотека или одређени протокол). Обично су овакви тестери специфични за одређени проблем.

Програми за расплинуто тестирање, независно од тога да ли су засновани на мутацијама или генерацијама, у некој мери користе случајне податке за генерисање тест примера. Могуће је све податке генерисати случајно, што доводи до тежег увиђања тачног низа догађаја који је довео до грешке. Друга могућност је да се на основу спецификације програма креирају тест примери који ће тестирати граничне вредности података, које су често критичне. Најбоље резултате даје генерисање тест примера комбинацијом ове две технике.

Мутацијски и генерацијски тестери су основни типови тестера. Поред њих постоји још један тип тестера који се заснива на напредној техници која се назива **еволюционо расплинуто тестирање** (енг. *evolutionary fuzzing*). Ова

техника се заснива на „учењу”. На основу претходних корака, тестер закључује које тест примере би требало даље да конструише. Рецимо, за сваки тест пример тестер може да процени покривеност кода, и закључи на који начин да измени тест пример тако да покривеност буде већа. Еволуционо тестирање се ослања на друге технике, попут техника генетских алгоритама [36, 17].

Други критеријум који дели програме за тестирање на основу начина генерисања тест примера је присуство или одсуство знања о томе како треба да изгледају улази у програм који се тестира [35, 36]:

- **Интелигентни тестери** (енг. *intelligent fuzzers*) познају структуру улазних података програма који се тестира. На тај начин се избегава генерисање података које би програм одбацио пре било каквог рада над њима. Имплементација интелигентног тестера је компликована јер захтева проучавање структуре података који представљају улазе у програм.
- **Неинтелигентни тестери** (енг. *non-intelligent/dumb fuzzers*) конструишу улазне податке за програм „на слепо” тј. не знајући ништа о структури тих података. На тај начин тестер може генерисати велики број улаза који ће брзо бити одбачени од стране програма. Њихова имплементација је једноставнија од имплементације интелигентних тестера.

Ако улаз у програм представља датотека одређеног формата, неинтелигентни тестер може да генерише тест примере тако што инвертује произвољне битове те датотеке. Са друге стране, интелигентни тестер намерно оставља неке делове датотеке неизмењеним а друге делове мења. Рецимо, ако структуру чине називи поља и њихове вредности, називи ће остати исти док ће се вредности мењати.

Интелигентни тестери углавном имају већу покривеност кода и могу да укажу на већи број грешака од неинтелигентних, али то није правило. Најбоље је креирати тестер који не иде ни у једну крајност, јер то води до лоших резултата. Тестер који не зна ништа о формату улаза може да креира бескорисне тест примере, али може и да пронађе неочекивану слабост система да се носи са неисправним улазима. Са друге стране, интелигентни тестер избегава губљење времена непотребним тест примерима, али обично претпоставља исправан рад неких функција система. Неретко се деси да овакав тестер не ради добро јер прави исте претпоставке које је правио и програмер који је писао програм, због којих и долази до одређених грешака. Такође, интелигентни

тестер може бити превише скуп због своје сложености. Балансиран приступ је најбоље решење: направити тестер који је довољно интелигентан да смањи време извршавања програма, али да не буде превише сложен и скуп, као и да не ослаби способност тестирања.

Неки аутори поистовећују тестере засноване на мутацијама са неинтелигентним тестерима, а тестере засноване на генерацијама са интелигентним тестерима. Такво поистовећивање није исправно. Тестери засновани на мутацијама најчешће заиста јесу неинтелигентни јер обично не узимају структуру улаза у обзир већ мењају битове постојећих улаза. Ипак, то није правило. Постоје и интелигентни тестери засновани на мутацијама који парсирају постојеће улазе да не би креирали оне које би програм одмах одбацио. Слично, тестери засновани на генерацијама могу бити неинтелигентни, иако је то веома редак случај. Рецимо, тестер који генерише улазе случајним подацима, без постојећих улаза, технички спада у тестере засноване на генерацијама.

У зависности од природе програма који се тестира, тестер који врши тестирање тог програма се може сврстати у неку од следећих категорија [35, 36]:

Локални тестери (енг. *local fuzzers*) У ову категорију се сврставају тестери разних локалних програма. Класичан пример програма који су погодни за расплинато тестирање су програми који омогућавају обичном кориснику да привремено (за време његовог извршавања) добије одређене привилегије. На *Unix* системима има неколико таквих подразумеваних апликација (енг. *setuid applications*), а то се може подесити и за друге. Слабости у оваквим програмима представљају велику опасност јер би грешке могле да омогуће кориснику да трајно добије привилегије и извршава произвољан код. Расплинато тестирање проналази слабости ових апликација тако што их покреће са трансформисаним аргументима. Аргументи се могу слати програму у виду аргумената командне линије (енг. *command-line fuzzers*), у виду променљивих окружења (енг. *environment variable fuzzers*), или у виду датотека одређеног формата (енг. *file format fuzzers*).

Тестери удаљених програма (енг. *remote fuzzers*) Ови тестери тестирају програме који се извршавају на удаљеним чворовима. Међу њима постоје две врсте тестера.

Прву врсту чине тестери мрежних протокола (енг. *network protocol fuzzers*). Тестирају се једноставни и сложени протоколи. Једноставни су

они који имају слабу аутентикацију или је уопште немају, користе *ASCII* карактере уместо бинарних података, итд. Сложени протоколи се састоје углавном из бинарних података међу којима се налази понека *ASCII* ниска карактера. Аутентикација је сложенија и често се за проверу послатих података користе дужина или збир података.

Другу врсту чине тестери Веб апликација (енг. *Web application fuzzers*). Они комуницирају помоћу *HTTP* протокола. Веома су важни јер данас имамо јако много апликација на Вебу с којима су се развиле и слабости јединствене за њих попут уметања *SQL* упита (енг. *SQL injection*), уметање скриптова (енг. *cross site scripting*), онемогућавања услуге (енг. *denial of service*) и многих других.

Меморијски тестери (енг. *in-memory fuzzers*) Ови тестери су корисни када су неке друге врсте неефикасне. Њихове идеје су једноставне и заснивају се на измени аргумената функција програма директно у меморији, при чему се мењају и аргументи који иначе нису доступни кориснику. Имплементације оваквих тестера су веома компликоване. Постоји више приступа оваквом тестирању. Један приступ подразумева замрзавање процеса и чувања његовог тренутног стања које се касније поново користи тј. изазове се без поновног покретања процеса. Након тога, у свакој итерацији се у процес у таквом стању уметну неисправни улазни подаци, посматра се резултат и враћа се на сачувано стање. На овај начин се добија на брзини и ефикасности јер се делови програма који су неважни за тестирање прескачу (замрзавањем стања нас не занима шта се и како се извршило пре тога), и одмах се долази до критичног дела. Ипак, ово је веома сложен процес. Могуће је у програм уметнути улазне податке који не би били могући при нормалном покретању програма. Такође, ако се дошло до грешке оваквим тестирањем, потребно је ту грешку репродуковати и нормалним покретањем програма, што може одузети доста времена.

3.4 Расплинуто тестирање формата датотека

Расплинуто тестирање формата датотека је метода тестирања програма који прихватају специфичне улазе. Улази су одређеног формата, тако да је циљ те-

стирања откривање грешака при парсирању улазних датотека. Примери су програми за руковање сликама, документима, музиком, итд. Програми су углавном на клијентској страни, али могу бити и на страни сервера (нпр. сервери електронске поште).

Особина овакве врсте тестирања која је чини једноставнијом од других врста тестирања је та што се углавном извршава целокупно на једном систему. Рецимо, у случају тестирања мрежних протокола или Веб апликација постоје барем два система: један на ком се налази програм који се тестира, и други на ком се покреће тестирање. Са друге стране, тестер формата датотека теже препознаје да је циљана апликација изазвала грешку или изузетак. При тестирању програма на серверу је најчешће очигледно када је дошло до грешке јер сервер постане недоступан. То није случај код клијентских апликација јер програм за тестирање изнова поново покреће апликацију са новим тест примерима. Без адекватног решења може да се деси да тестеру промакне важна слабост програма. Због тога је важно да тестер формата датотека надгледа програм који тестира помоћу постојећег дебагера или дебагера који је уметнут у сам програм за тестирање. То повећава сложеност тестирања формата датотека [35, 36].

Тестирање се може описати помоћу следећих пет корака [35]:

1. Припремити тест пример методом заснованом на мутацијама или методом заснованом на генерацијама
2. Покренути циљну апликацију и учитати припремљен тест пример
3. Надгледати циљну апликацију (најчешће коришћењем дебагера)
4. У случају откривања грешке, забележити све потребне информације о њој. Ако након одређеног времена ниједна грешка није пронађена, ручно угасити циљну апликацију.
5. Поновити дате кораке.

Расплинуто тестирање формата датотека може бити интелигентно и неинтелигентно, као што може користити и обе методе засноване на мутацијама и генерацијама. У наставку су описане најчешће коришћене две технике и њихове добре и лоше стране [35]:

Неинтелигентно расплинуто тестирање засновано на мутацијама представља веома једноставан приступ чији квалитет зависи од квалитета

почетних улаза. Наредни улази се добијају мењањем постојећих, па је важно имати међусобно различите почетне улазе да би што више случајева било покривено. Мењање улаза може бити у виду мењања појединачних бајтова, скупа бајтова, додавањем нових података, итд. Проблем код овог приступа је то што програми пре рада са улазним подацима често проверавају њихову исправност методама као што је провера контролне суме (енг. *checksum*). Улазни податак који је промењен не пролази проверу и бива брзо одбачен од стране програма. Због оваквих провера, тестер успева да испита јако мали део кода. Ово се може решити искључивањем провера што је тежак посао. Такође, овај приступ није ефикасан јер је број свих могућих начина на које можемо променити улазни податак превелик. Могуће је тестирати мали број могућности. Добра страна овог приступа је веома мала количина труда потребна пре покретања тестера, а она се односи на избор почетних тест примера.

Интелигентно расплинута тестирање засновано на генерацијама је приступ који генерише тест примере на основу шаблона формата датотеке који се користи у тестирању. Шаблон многих формата датотека је познат и може се лако наћи, док у случају непостојања шаблона програмер мора сам проучити дати формат датотека. Квалитет тестирања зависи од тога колико добро програмер познаје дати формат датотека, због чега се много времена улаже у његово проучавање. Проучавање одузима доста времена, али је овај приступ ефикаснији јер тежи покривању већег дела кода.

При парсирању неисправних датотека се може наићи на разне врсте грешака. Неке од најчешћих врста грешака које покушавамо да откријемо расплинутим тестирањем су [35, 36]:

Онемогућавање услуге (енг. *denial of service*) Ово је напад на систем у коме се онемогућава његов рад. Нападач преоптерећује систем тако што му шаље јако велики број захтева. Системи подложни овом нападу су апликације на серверу.

Проблеми у раду са целим бројевима Рад са целим бројевима је подложен грешкама које се односе на прекорачење меморије. Разлог за то неретко буде знак целих бројева. Јављају се грешке при алокацији меморије, раду са индексима низова, поређењу означених и неозначених целих бројева, итд.

Прекорачење меморије До прекорачења меморије може доћи из много разлога. Типичан пример подразумева копирање податка превелике величине у бафер фиксиране, мање величине. На тај начин може доћи до угрожавања безбедности меморије и извршавања произвољног кода.

Ниске карактера са описом формата Једна од познатих слабости програма је слабост при раду са нискама карактера са описом формата (енг. *format string vulnerability*). Опасност представљају функције за исписивање података на излаз које као аргументе имају овакве ниске. Примери су функције језика *C* *printf* и *fprintf*. За испис се користе параметри попут *%x*, *%s*, *%d* који говоре о типу конверзије која треба да се примени на аргументе функције. Проблем настаје када функција за испис очекује више аргумената него што јој је послато, тако да има приступ недозвољеним меморијским локацијама. Нападач успева да изазове овакво понашање тако што функцији за испис даје као аргумент ниску карактера која садржи параметре *%s*, *%x*, итд. На овај начин се на нападнутом систему могу извршити произвољне команде, прочитати недозвољени подаци, могу се изазвати пад система и друге последице.

Надметање за ресурсима Ово је ситуација када понашање програма зависи од догађаја којима не можемо предвидети време извршавања. Вишеничне апликације су склоне овом проблему (описано у поглављу [2.3.1](#)).

Важна особина програма за расплинуто тестирање је начин на који препознаје да је дошло до грешке или изузетка у програму. Повратна вредност апликације се користи као један начин препознавања и интерпретирања грешке. Програмери се ређе одлучују за ту опцију јер постоје други начини који дају више информација о изузетку. Најчешће се у ову сврху користи дебагер, али он може бити незгодан јер се већина дебагера употребљава интерактивно. То може смањити ефикасност аутоматизације расплинутог тестирања, јер захтева ручно поновно покретање програма који је изазвао грешку. Други начин је имплементирање неких особина дебагера у самом тестеру. То може одузети више времена и свакако захтева више труда и истраживања, али на овај начин се имплементира тестер који је прилагођен потребама.

Глава 4

Структура датотека у формату *PDF*

Један од формата који се користи за приказивање и размену разних врста садржаја попут текста, слика, аудио и видео записа је формат *PDF* (енг. *Portable Document Format*). Приказ датотека у формату *PDF* не зависи од софтвера, хардвера и оперативног система. Компанија *Adobe* је изумела овај формат, а данас га одржава Међународна организација за стандардизацију (енг. *International Organization for Standardization*). Датотеке у формату *PDF* имају много могућности, нпр. могу извршавати код написан у језику *Javascript*, могу бити заштићене шифром, итд [5, 4].

4.1 Објекти датотека

Свака датотека у формату *PDF* је сачињена из великог броја *објеката*. На пример, свака страница датотеке је описана *објектом странице* (енг. *page object*). Објекат странице реферише на објекте који описују садржај те странице (текст, слике, итд.).

Датотека формата *PDF* се састоји од објеката следећих типова [7, 11, 21]:

Истинитосне вредности Представљају логичке вредности и дефинишу се кључним речима *true* (тачно) и *false* (нетачно).

Бројеви Користе се два типа објеката: цели и реални бројеви. Цели бројеви се представљају једном или више цифара којима може претходити знак

броја ('+' или '-'). Реални бројеви се представљају исто као цели бројеви, с тим што у себи могу садржати децималну тачку.

Имена Представљају симболе дефинисане секвенцом *ASCII* карактера која следи одмах иза косе црте. Специјални карактери (нпр. бланко, заграда, итд.) унутар имена се представљају својим хексадецималним записом иза карактера '#'. Имена се користе за описивање садржаја објеката. Сваки објекат има садржај који је значајан за приказ датотеке, а природа самог садржаја се описује помоћу имена „/Type”. На пример, ако желимо да дефинишемо фонт којим је написана датотека, правимо објекат који садржи информације о том фонту. У таквом објекту се након имена „/Type” наводи име „/Font”, као индикатор да објекат садржи информације о фонту. Конкретан фонт дефинишемо именом који представља тај фонт (нпр. фонт „Times-Roman” бисмо дефинисали помоћу имена „/Times-Roman”).

Ниске карактера Представљају низове бајтова. Могу се представити произвољним бројем карактера између '(' и ')', заграда, или низом хексадецималних записа карактера између '<' и '>' заграда.

Низови Представљају уређене секвенце 0 или више објеката који не морају бити истог типа. Објекти се наводе између заграда '[' и ']'.

Каталози Представљају уређену листу парова кључ/вредност. Кључ се представља именом, а вредност било којим типом објекта. Парови се налазе између заграда '<<' и '>>'.

Токови Представљају секвенце бајтова неограничене дужине, због чега се велики блокови података попут слика углавном смештају у њих. Токови се означавају каталогом који описује податке који се налазе у њему, након кога се блок података наводи између кључних речи *stream* и *endstream*. Каталог тока садржи разне информације, попут информација о броју бајтова који подаци заузимају, типу објекта који ток описује, филтерима који се примењују на податке, броју објеката у подацима итд.

Објекат „Null” Представља објекат чији тип и вредност нису једнаки типу и вредности било ког другог објекта. Референца на непостојећи објекат се третира као објекат „Null”. Он се означава кључном речју *null*.

Индиректни објекти Представљају објекте који су нумерисани бројевима који представљају њихове јединствене идентификаторе. Идентификатори омогућавају другим објектима да реферишу на индиректне објекте. Индиректни објекти се означавају својим идентификатором након кога се између кључних речи *obj* и *endobj* налази сам објекат. Било који објекат у датотеци може бити индиректан објекат. Објекат странице спада у ову категорију.

4.2 Организација датотека

Због могућности приказа разноликог садржаја, структура датотека у формату *PDF* је веома комплексна. Као што је приказано на слици 4.1, структура се састоји из четири дела [7, 11, 21]:



Слика 4.1: Структура датотека у формату *PDF*

Заглавље Прва линија датотеке чини њено заглавље и садржи број верзије формата *PDF* која је коришћена у датотеци. У случају да се у датотеци налазе и бинарни подаци, потребно је у заглавље додати и другу линију која садржи најмање четири бинарна карактера (*ASCII* вредност им је већа или једнака од 128). Друга линија служи као индикатор да се у датотеци налазе бинарни подаци, што је корисно апликацијама које раде са датотекама формата *PDF*.

Тело Садржи све индиректне објекте који се налазе у датотеци. Сви објекти су смештени у стабло, чији је корен објекат који се назива каталог објекат

(енг. *document catalog*). Каталог објекат је каталог који садржи информације о датотеци и реферише на друге објекте који дефинишу податке датотеке. Подстабло (стабла свих објеката) које садржи све објекте страница се назива *стабло страница* (енг. *page tree*).

Табела унакрсних референци Табела *xref* или табела *cross reference* садржи податке о свим индиректним објектима у датотеци и омогућава брз приступ сваком од њих. Уместо претраге целе датотеке у потрази за одређеним објектом, локација објекта се прочита из табеле након чега му се може директно приступити.

Сваки објекат описан је једним редом у табели дужине 20 бајтова. Првих 10 бајтова имају различиту намену у зависности од врсте објекта. Постоје две врсте објеката: *валидни* и *ослобођени* објекти. Ове две врсте објеката, као и само ослобађање су објашњени у поглављу 4.3.

У случају валидног објекта, првих 10 бајтова садржи локацију објекта у датотеци. Локација објекта је одређена помоћу удаљености (енг. *offset*) почетка датотеке до почетка дефиниције објекта. У случају ослобођеног објекта, првих 10 бајтова садржи број следећег ослобођеног објекта (референцу на следећи ослобођени објекат у повезаној листи свих ослобођених објеката).

Након првих 10 бајтова, у случају обе врсте објекта следи *генерацијски број* (енг. *generation/revision number*), који се увећава сваки пут када се објекат ослободи. Након генерацијског броја налази се индикатор који говори о врсти објекта: 'f' за ослобођен објекат или 'n' за валидан објекат.

„Trailer” Ова секција датотеке се дефинише кључном речју *trailer*. Овде се налази каталог који садржи информације о датотеци као нпр. број редова у табели унакрсних референци, референца на каталог објекат, итд. Најважнија информација је информација о локацији табеле унакрсних референци тј. о удаљености почетка датотеке до почетка дефиниције табеле. Број бајтова који представља удаљеност се наводи након кључне речи *startxref*. Локација табеле је важна јер омогућава брз приступ табели, а она даље својом структуром омогућава брз приступ објектима датотеке. Последња линија секције *trailer* садржи индикатор за крај датотеке `%%EOF`.

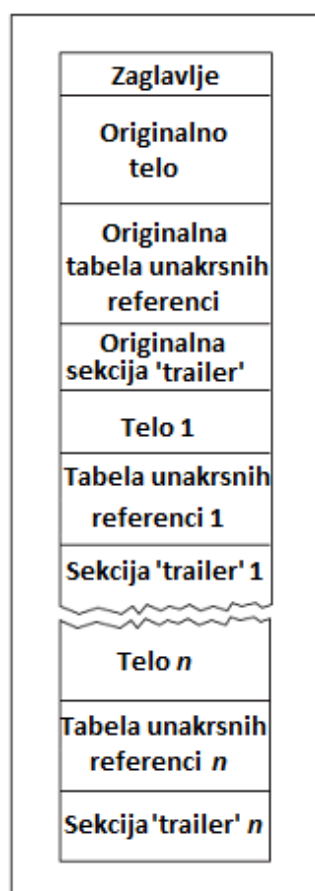
4.3 Ажурирање датотека

Датотеке у формату *PDF* се могу мењати, чиме се мења њихова структура тј. објекти у њима. Ажурирање датотеке не подразумева мењање оригиналног садржаја датотеке, већ додавање нових података на сам крај датотеке. Рецимо, заменом слике из датотеке новом сликом, објекат који представља стару слику неће бити обрисан или промењен тако да му садржај представља нову слику. Уместо тога, објекат старе слике биће означен као **ослобођен**, док ће објекат нове слике бити додат на сам крај датотеке. Тачније, ажурирањем датотеке се додаје ново тело, нова табела унакрсних референци и нова секција *trailer* на крај датотеке. Ажурирање се може обављати неограничен број пута, при чему свака итерација (свако ажурирање) додаје три нове секције на крај датотеке. Датотека која је ажурирана n пута је илустрована на слици 4.2.

У новом телу се налазе нови или измењени објекти датотеке. Нова табела унакрсних референци садржи искључиво локације нових, измењених и ослобођених објеката. Локације старих објеката се и даље читају из старе табеле унакрсних референци (табеле унакрсних референци претходне итерације). До старих објеката се лако долази, јер нова секција *trailer* поред локације нове табеле садржи и локацију претходне табеле.

Сваки објекат који се обрише или измени постаје ослобођен, док се они који нису ослобођени називају *валидни* објекти. Сви ослобођени објекти се налазе у једној повезаној листи. Они се третирају као обрисани и не могу бити у садржају датотеке, а референце на њих су једнаке објекту „Null”. Ипак, објекат који је ослобођен може поново постати валидан неким наредним ажурирањем. *Генерациски број* објекта нам говори о укупном броју ослобађања једног објекта [7, 11, 21].

Ажурирање датотеке на овај начин омогућава да се мале измене у великој датотеци брзо сачувају. Такође, некада је важно сачувати оригиналан садржај датотеке [7].

Слика 4.2: Структура ажуриране датотеке у формату *PDF*

Глава 5

Имплементација програма за расплинуто тестирање читача датотека у формату *PDF*

Као демонстрација могућности Скале у паралелизацији задатака развијен је програм за расплинуто тестирање читача датотека у формату *PDF*. Паралелизован је процес генерисања датотека у формату *PDF*, као и њихово отварање у одређеном читачу. Програм користи библиотеку *Akka* за имплементацију модела размене порука помоћу модела Актер. Што се тиче самог тестирања, имплементиран је интелигентан тестер заснован на мутацијама. Мутације се примењују на садржај датотека из корпуса датотека, који је описан у наставку. Коришћена је верзија 2.12.5 језика Скала, а програм је развијен у окружењу *IntelliJ IDEA Community Edition 2017.3.5*. Изворни код се налази у репозиторијуму на сајту *github*¹.

Корпус датотека је директоријум који садржи различите датотеке у формату *PDF*. Корисник може додавати и брисати датотеке из корпуса. Иницијално је у корпус смештено двадесет шест датотека. Све датотеке садрже типове објеката описане у поглављу 4.1, а оно што их разликује је садржај тих објеката и величина датотеке. Обухваћени су типови садржаја као што су текст, слике, аудио и видео записи, линкови, интерактивни дугмићи и тродимензионални објекти.

¹<https://github.com/ancim/Master/tree/master/fuzzer>

5.1 Улазни параметри

Покретањем програма са аргументом командне линије *--help* програм исписује упутство за корисника. Упутство садржи информације о аутору, сврси, начину рада и начину употребе програма. Пре почетка тестирања од корисника се тражи да унесе:

1. **Читач датотека у формату *PDF*** који жели да тестира. Могуће је изабрати један од понуђених читача или унети путању до извршне датотеке произвољног читача.
2. **Број итерација** који представља број датотека који ће отворити сваки од објеката који су задужени за тестирање. Ти објекти су објекти Актер који су детаљно објашњени у наредном поглављу.
3. **Број Извршилаца** који представљају објекте Актер који конкурентно (паралелно, уколико је могуће) мењају садржај датотека и отварају их у одабраном читачу (детаљније објашњење налази се у поглављу 5.3). Минималан број Извршилаца који корисник може да унесе је 1, а максималан зависи од рачунара на коме се извршава програм. Програм рачуна максималан број на основу два критеријума: броја језгара процесора и количине слободне меморије.

Уколико је број језгара процесора x , максималан број Извршилаца се најпре поставља на $3 \cdot x$. Иако истовремено не може радити више од x Извршилаца, одабрано је да их буде више. Разлог томе је тај што сваки од њих након одређених акција чека да прође одређени број секунди (што је објашњено у поглављу 5.3). Како не би долазило до великог губљења времена док сви Извршиоци истовремено чекају, идеја је да их има више како би се извршавали конкурентно. Такође је важно да их не буде превише. Не треба преоптеретити рачунар непотребним бројем објеката који се не могу паралелно извршавати јер нема довољно језгара. Превелики број објеката Актер може смањити перформансе због дељења много мањег броја процесора. Број 3 је одабран произвољно.

Сваки Извршилац учитава целу датотеку у меморију. Због тога максималан број Извршилаца може бити и мањи од $3 \cdot x$: број се смањује уколико слободне меморије нема довољно да сваки од Извршилаца учита датотеку.

Програм је подешен тако да дозвољава да датотеке заузимају половину слободне меморије.

4. **Број милисекунди** који представља време које се додељује читачу да отвори одређену датотеку. Неким читачима као што су *Adobe*, *Foxit* и *Sumatra* довољно је 1000ms да отворе датотеку, док су неки други читачи као што је читач *Slim* много спорији и потребно им је барем 3000ms. Уколико се читачу не додели довољан број секунди за отварање датотеке, процес задужен за отварање ће бити угашен пре него што је датотека отворена. У том случају програм неће успети да детектује грешку у читачу (уколико она постоји). Због тога је најбоље пре самог тестирања читача проценити отприлике који број милисекунди му је довољан за отварање датотека.

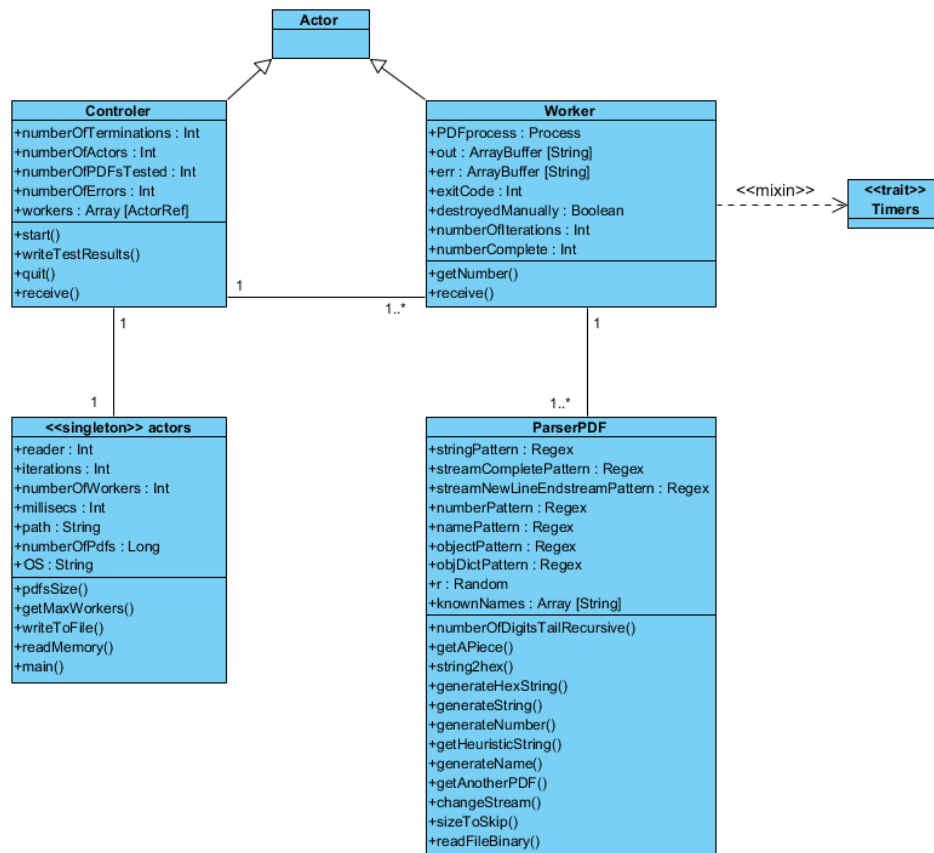
Пре тестирања одређеног читача потребно је подесити га тако да сваку датотеку отвара у новом прозору. Уколико гашење једног прозора не утиче на остале прозоре, то значи да отварање сваке датотеке представља независан процес. Ово је неопходно за правилан рад програма.

5.2 Организација програма

Програм је организован у четири класе које су приказане дијаграмом на слици 5.1. Класе *Controler* и *Worker* представљају објекте Актер задужене за паралелизацију тестирања. Способност *Timers* је спојено у класу *Worker*, а класа *ParserPDF* је задужена за мутацију садржаја датотека. Главни програм налази се у објекту *actors* који представља једини објекат истоимене класе. Због тога се он означава као објекат *singleton*. У наредним поглављима је детаљније приказан начин рада објеката ових класа.

5.3 Објекти Актер

Сви објекти Актер у програму припадају једном систему *ActorSystem*. На врху хијерархије корисничких компоненти налази се један објекат Актер класе **Controler** који је задужен за покретање и праћење рада свих осталих објеката Актер. Њега називамо *Контролер*. Систем *ActorSystem* и објекат Контролер



Слика 5.1: Дијаграм класа

се креирају у функцији *main* која се налази у објекту *actors*. Њихово креирање и слање поруке објекту Контролер је приказано у следећем примеру:

```

val system = ActorSystem("fuzzerSystem");
val a = system.actorOf(Props[Controller],
    name="fuzzer-controller-actor")
a ! "Pocni!";

```

Објекат Контролер креће са радом када прими поруку „Pocni!”. Тада позива метод *start()*, који је приказан у следећем сегменту кода:

```

def start() = {
    println("Pokrecem Worker-e");
    val workers: Array[ActorRef] = new Array[ActorRef](numberOfActors);
    for(i <- 0 until numberOfActors) {
        /* pravimo Actor-a i saljemo mu poruku za pokretanje */
    }
}

```

```
workers(i) = context.actorOf(Props(new Worker(i,  
    numberOfActors)), name = "myWorker-" + i);  
context.watch(workers(i));  
workers(i) ! "Pokreni";  
}  
}
```

Објекат Контролер генерише друге објекте Актер који су објекти класе **Worker**. Њих називамо *Извршиоци*. Контролер чува референце на Извршиоце (типа *ActorRef*) у низу, и сваком од њих шаље поруку за покретање (порука „Pokreni”). Метод *watch* омогућава Контролеру да „посматра” креиране Извршиоце, што значи да ће бити обавештен сваки пут када неки од њих заврши са радом.

Извршиоци су објекти који су задужени за примену расплинутог тестирања. Сваки од њих има свој редни број на основу кога учитава једну од датотека из корпуса датотека. Извршиоци садрже објекат класе **PDFparser**, који примењује мутације над подацима те датотеке, како је описано у поглављу 5.4. На тај начин Извршилац добија тест пример тј. нову датотеку коју треба да отвори у читачу који се тестира.

Отварање измењене датотеке се догађа у спољашњем процесу чије се извршавање може надзирати. То се постиже помоћу библиотеке *scala.sys.process*. Ова библиотека омогућава извршавање, посматрање спољашњег процеса и руковање његовим улазом и излазом. Основу ове библиотеке чине класе *Process* и *ProcessBuilder* језика Јава. Скала има класе истих имена и оне су коришћене у имплементацији.

Претпоставка је да сваки читач треба да препозна лоше конструисане и злонамерне датотеке, да пријави грешку и нормално настави са радом. У случају да се читач сам угаси након отварања датотеке, то је индикатор да је дошло до грешке коју није умео да обради. Због тога се спољашњем процесу који покреће читач даје одређено време (улазни параметар *број милисекунди* наведен у поглављу 5.1) да отвори датотеку, након чега се анализира његово стање. Уколико је процес и даље активан, то значи да је датотека и даље отворена у читачу или да је читач успешно обрадио грешку насталу при отварању. Тада се овај процес прекида од стране Извршиоца који га је креирао. Са друге стране, уколико процес није активан, то значи да се сам угасио и да је пронађена слабост читача. У том случају се документују стандардни излаз,

стандардни излаз за грешке и повратна вредност процеса. То су информације о извршавању процеса. Ове информације се чувају у текстуалној датотеци у директоријуму „results”. Назив текстуалне датотеке садржи префикс „CRASH_” након кога следи редни број Извршиоца. Такође, поред префикса „CRASH_” могу следити и префикси „ERR_” и „NULL_”. Префикс „ERR_” се појављује уколико стандардни излаз за грешке није празан, а „NULL_” уколико је нека од информација о процесу *NULL*. Чува се и датотека која је узроковала грешку, у директоријуму „fuzzedPDFError”.

Команда која отвара датотеку која се налази у директоријуму „fuzzedPDFcorpus” у читачу *SlimPDFReader* и покретање спољашњег процеса приказани су у следећем примеру:

```
val command = ".\\PDF_citaci\\slim\\SlimPDFReader.exe
               .\\fuzzedPDFcorpus\\" + numberComplete + "_fuzzed.pdf";
/* за nekoliko sekundi saljemo poruku "stop" sami sebi kako bismo
   analizirali stanje procesa */
timers.startSingleTimer("key", "stop", millisecs milliseconds);

/* pokretanje spoljasnjeg procesa */
val qb: ProcessBuilder = Process(command)
PDFprocess = qb.run (ProcessLogger((s) => { out ::= s},
                                   (s) => { err ::= s}));
```

Команда која отвара датотеку је дата као ниска карактера облика: „.\\odabranCitac.exe .\\nazivDatoteke.pdf”. Чекање од одређеног броја милисекунди пре анализирања статуса процеса се остварује помоћу способности *akka.actor.Timers* који садржи објекат *timers* апстрактне класе *TimerScheduler*. Метод *startSingleTimer* овог објекта омогућава Извршиоцу да сам себи пошаље поруку у одређено време тј. након дефинисаног броја секунди. Као што се види у примеру, објекат ће након *millisecs* милисекунди себи послати поруку „stop”. Када Извршилац прими поруку „stop”, то значи да је истекао период рада спољашњег процеса. Тада Извршилац треба да анализира статус процеса и да га заустави уколико је још увек активан. Класе *Process* и *ProcessBuilder* се користе за креирање процеса на основу дате команде. Метод *run* извршава дати процес, а њему се прослеђује објекат типа *ProcessLogger* који рукује стандардним излазом и стандардним излазом за грешке. Стандардни излаз се уписује у листу ниски карактера *out*, а стандардни излаз за грешке у листу ниски карактера

err.

Извршиоци се извршавају паралелно и изнова креирају тест примере и нове спољашње процесе који их отварају у читачу. Број датотека које ће креирати и отворити сваки од њих зависи од *броја итерација* који је унео корисник пре почетка тестирања. Једна итерација подразумева да сваки Извршилац тестира читач отварањем једне датотеке. На пример, ако желимо да сваки Извршилац отвори x датотека, треба да унесемо x као број итерација. Укупан број датотека које ће програм отворити у читачу је онда $n \cdot x$, где је n број Извршилаца. Друга могућност је да се програм извршава све док корисник сам не прекине извршавање програма. У том случају, као број итерација треба унети број -1 .

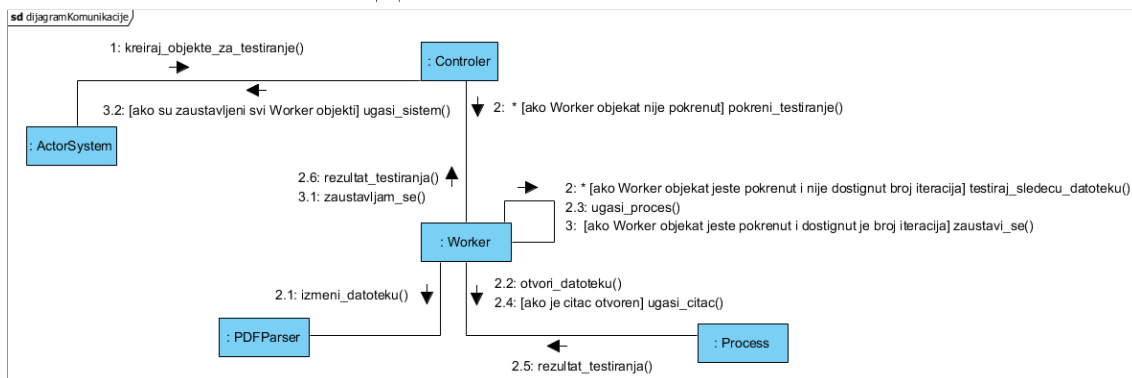
Уколико је дефинисан одређени број итерација (ако је различит од -1), сваки Извршилац завршава са радом када анализира последњи процес који је покренуо. Тада зауставља сам себе, а порука о томе се шаље његовом супервизору, објекту Контролер. Када Контролер добије поруку о заустављању од сваког детета, он зауставља цео систем *ActorSystem* чиме се завршава извршавање програма.

Интеракција објеката приказана је дијаграмом комуникације на слици 5.2. Како број Извршилаца зависи од уноса корисника и рачунара на ком се извршава програм, на дијаграму је приказан само један Извршилац и његова комуникација са другим објектима. Такође, приказ више Извршилаца на дијаграму је незгодан због паралелизма и конкурентности у програму. Не можемо предвидети када ће одређени Извршилац добити процесорско време, због чега је немогуће приказати тачан редослед корака и порука које објекти размењују. Оно што се може одредити и што је приказано на дијаграму јесу кораки кроз које пролазе сви Извршиоци. Важно је напоменути и да Извршиоци не комуницирају између себе.


5.4 Креирање датотека у формату *PDF*

Објекти класе **ParserPDF** су објекти који примењују измене над садржајем датотеке коју добију као аргумент. Ова датотека представља улаз тј. основу за конструисање нове датотеке. Применом мутација над неким подацима постојеће датотеке добија се нова датотека која ће бити коришћена као тест пример.

Због присуства бинарних података у датотекама формата *PDF*, садржај целе датотеке се најпре учитава као низ бајтова. Низ бајтова се потом конвертује у



Слика 5.2: Дијаграм комуникације

ниску карактера, јер је потребно претражити садржај датотеке у облику разумљивом људима. Ниска карактера се претражује у потрази за објектима одређених типова, ради њихове измене. Када се измени жељени садржај, потребно је ниску карактера конвертовати натраг у низ бајтова који ће бити уписан у нову датотеку. При обе конверзије је важно користити једнобајтну кодну шему *ISO-8859-1*. Ова кодна шема је погодна јер чува оригиналан садржај бинарних података. Наиме, у вишебајтним кодним шемама није свака секвенца бајтова валидна, због чега се при декодирању секвенце она може пресликати у неки заменски карактер (нпр. ? или ). То се може догодити током конверзије низа бајтова у ниску карактера. На основу ових заменских карактера је немогуће добити натраг полазну секвенцу бајтова (при конверзији ниске карактера натраг у низ бајтова). Такође, неке једнобајтне кодне шеме не кодирају свих 256 различитих карактера. Због тога је у овом случају потребно користити једнобајтно кодирање које кодира 256 различитих карактера. На тај начин се бајтови јединствено пресликавају у карактере, а касније се ти карактери јединствено пресликавају у бајтове.

Када је датотека претворена у ниску карактера, тада се део њеног садржаја мутира тј. мења. Мутације се примењују над различитим типовима објеката датотеке: бројевима, нискама карактера, именима, индиректним објектима, каталозима и токовима. Због тога је потребно препознати појављивања ових објеката у датотеци. Треба напоменути да се унутар токова игнорише појављивање било ког другог објекта јер токови садрже бинарне податке које не треба претраживати као ниске карактера. Препознавање објеката се врши помоћу *регуларних израза*. Библиотека која омогућава коришћење регуларних израза у Скали је библиотека *scala.util.matching*. Регуларни изрази су дефинисани на

следећи начин:

```
/* sabloni koje pretražujemo u datotekama */  
val stringPattern: Regex = ""\"([\\s\\S]+?)|\\<[0-9a-fA-F]*?\\>""\".r  
val streamCompletePattern: Regex = ""\"(stream)([\\s\\S]+?)(endstream)""\".r  
val streamNewLineEndstreamPattern: Regex = ""\"(stream)\\n(endstream)""\".r  
val numberPattern: Regex =  
    ""\"[+-]?((\\d+\\.?.\\d+)| (\\d*\\.?.\\d+)| (\\d+\\.?.\\d*))""\".r  
val namePattern: Regex = ""\"/[a-zA-Z0-9]+""\".r  
val objectPattern: Regex = ""\"(\\d+) (\\d+) (obj)([\\s\\S]+?)(endobj)""\".r  
val objDictPattern = ""\"(obj)([\\s\\S]*)(\\<\\>)([\\s\\S]+?)(\\>\\>)""\".r
```

Применом метода *r* на ниске карактера које представљају регуларне изразе добијамо обрасце који су типа *scala.util.matching.Regex*.

Следећи пример илуструје: читање и смештање бајтова датотеке у низ бајтова, конверзију низа бајтова у ниску карактера, и проналажење свих токова података у ниски карактера помоћу обрасца *streamCompletePattern* (који је дефинисан у претходном примеру):

```
val byteArray: Array[Byte] = Files.readAllBytes(Paths.get(fileName));  
/* pravimo string koji sadrzi podatke iz datoteke */  
val contentString = new String(byteArray,  
    Charset.forName("ISO-8859-1"))  
val matchStreams =  
    streamCompletePattern.findAllMatchIn(contentString).toList
```

Подаци се читају из датотеке чија је путања наведена у ниски карактера *fileName*. Коришћена је библиотека *java.nio.file* и њена класа *Files* која садржи различите статичке методе за манипулацију датотекама и директоријумима. Метод *readAllBytes* чита бајтове датотеке који се чувају у низу бајтова *byteArray*. Од добијеног низа бајтова се конструише ниска карактера *contentString*. Метод *findAllMatchIn* проналази сва појављивања регуларног изрази у ниски карактера, а метод *toList* сва појављивања смешта у листу.

Радње које се примењују над објектима које треба изменити су брисање, додавање и измена постојећих података, при чему се над једним објектом не морају применити све радње. На пример, неке објекте можемо заменити новим објектом истог типа, док неке друге можемо добити брисањем једног дела садржаја објекта. Нови објекти могу бити исте или различите величине од објекта који мењају. Могу се добити псеудо случајно или помоћу хеуристика.

Генерисање псеудо случајних података се врши методама класе *scala.util.Random*. На пример, псеудо случајан цео број се може добити позивом метода *Random.nextInt()*. Слично, постоје и методе *Random.nextString()*, *Random.nextDouble()*, итд.

Што се тиче хеуристика, коришћен је приступ који у своје тест примере умеће вредности објеката које често доводе до грешака у раду програма. Типови објеката који су обухваћени су ниске карактера, цели и реални бројеви. Вредности ових типова су прикупљене са два репозиторијума на сајту *github*^{2,3}. Ови репозиторијуми садрже податке који имају велику вероватноћу да изазову разне врсте грешака када се користе као улазни подаци програма. Ту се могу наћи адекватни улазни подаци у зависности од типа улазних података, природе програма који се тестира и типичних врста напада на програм, оперативног система на коме се извршава програм, итд. Нова вредност објекта који се мења хеуристички ће бити случајно одабрана из свих прикупљених вредности тог типа.

Прва измена која се примењује над датотекама је брисање неких индиректних објеката. Број објеката и објекти који ће бити обрисани се одређују случајно, након чега је датотека спремна за претраживање осталих типова објеката. У наставку су објашњени могући начини њихове измене:

Бројеви При измени броја случајно се одређује начин на који ће се генерисати нови број који ће га заменити: псеудослучајно или помоћу хеуристика. Прикупљених бројева има мало и то су граничне вредности (целих и реалних бројева), вредности блиске нули, бројеви 0, -1, итд. Ови бројеви често изазивају грешке прекорачења меморије. У малом броју случајева, број се мења хеуристички изабраном ниском карактера.

Поред токова података и имена, објекти који се највише појављују у датотекама формата *PDF* су бројеви. Они су распрострањени свуда, а највише као идентификатори других објеката. Мењањем референци на друге објекте можемо постићи реферисање на непостојеће, невалидне или објекте погрешног типа.

Ниске карактера При измени ниске карактера случајно се одређује један од три начина измене. Прва два начина су једнака као и код измене бројева:

²<https://github.com/minimaxir/big-list-of-naughty-strings/tree/master/naughtystrings>

³<https://github.com/fuzzdb-project/fuzzdb/tree/master/attack>

псеудослучајно или хеуристички. Дужине псеудослучајних ниски карактера се одређују случајно тако да некад буду једнаке као ниске карактера које мењају, а некад им се дужина одреди случајно. Што се тиче хеуристика, прикупљених ниски карактера има јако много, а неки примери су приказани у табели 5.1. Најчешће ниске које се користе за тестирање су ниске карактера са описом формата, ниске које представљају претрагу по систему датотека, ниске које садрже специјалне карактере, позиве функција, *html* тагове, итд. Трећи начин мења ниску карактера ниском која представља целу датотеку формата *PDF*. Циљ је уметнути другу датотеку у датотеку чији садржај мењамо. Датотека која се умеће може бити исправна датотека или датотека коју је раније „покварио” тј. генерисао програм за тестирање.

Имена При измени имена случајно се одређује један од три начина измене. Први начин генерише име псеудослучајно тј. ниску карактера која почиње знаком *'/'* за којим следи ниска карактера произвољне дужине. Други начин уместо имена умеће хеуристички одабрану ниску карактера, док трећи начин представља замену имена једним од имена који се често користе у датотекама формата *PDF*. Често коришћена имена су дефинисана у самом програму и прикупљена су на основу корпуса датотека.

Токови података Објекти овог типа се не могу мењати као обичне ниске карактера већ се конвертују у низ бајтова. Како су токови података већином веома дугачки, није циљ променити цео ток већ променити само његове „парчиће”. „Парчићи” су његови поднизови који најчешће буду промењени псеудо случајним низовима бајтова добијеним методом *Random.nextByte()*. У малом броју случајева, подниз буде замењен низом бајтова који представља другу датотеку (уметање друге датотеке је могуће и код измене ниске карактера). Број поднизова за измену, њихова величина и локација у низу се одређују случајно. Након свих промена у низу бајтова, он се конвертује натраг у ниску карактера. При свакој конверзији користи се кодна шема *ISO-8859-1*.

Каталози Каталози који се могу променити у датотеци су каталози који следе након дефиниције индиректних објеката, тј. након кључне речи *obj*. Они се одређују случајно и мењају се на само један начин: сваки каталог ода-

| |
|--|
| System("\\ls -al /\") |
| <?xml version="1.0" encoding="ISO-8859-1"?><!DOCTYPE foo [<!ELEMENT foo ANY ><!ENTITY xxe SYSTEM "file:///etc/passwd" >]><foo>&xxe;</foo> |
| \$ENV{'HOME'} |
| ../../../../../../../../etc/passwd%00 |
| Kernel.exec("\\ls -al /\") |
| eval("\\puts 'hello world'") |
| 1'; DROP TABLE users-- 1 |
| </textarea><script>alert(123)</script> |
| <IMG SRC="javascript:alert('XSS')" |
| /%c1%9c../%c1%9c../%c1%9c../%c1%9c../%c1%9c{FILE} |
| /../../../../../../../../{FILE} |
| DCC SEND STARTKEYLOGGER 0 0 0 |
| %d %d %d%d%d%d%d%d%d |

Табела 5.1: Примери ниски карактера

бран за промену копира садржај претходно одабраног каталога (у случају првог одабраног каталога, његов садржај остаје непромењен).

5.5 Резултати тестирања

У току тестирања читача долазило је до проблема са недостатком меморије. Због тога су одређени делови кода имплементирани на два начина, због чега разликујемо прву и другу верзију програма. У наставку су описане обе верзије, њихово поређење и грешка која се јављала.

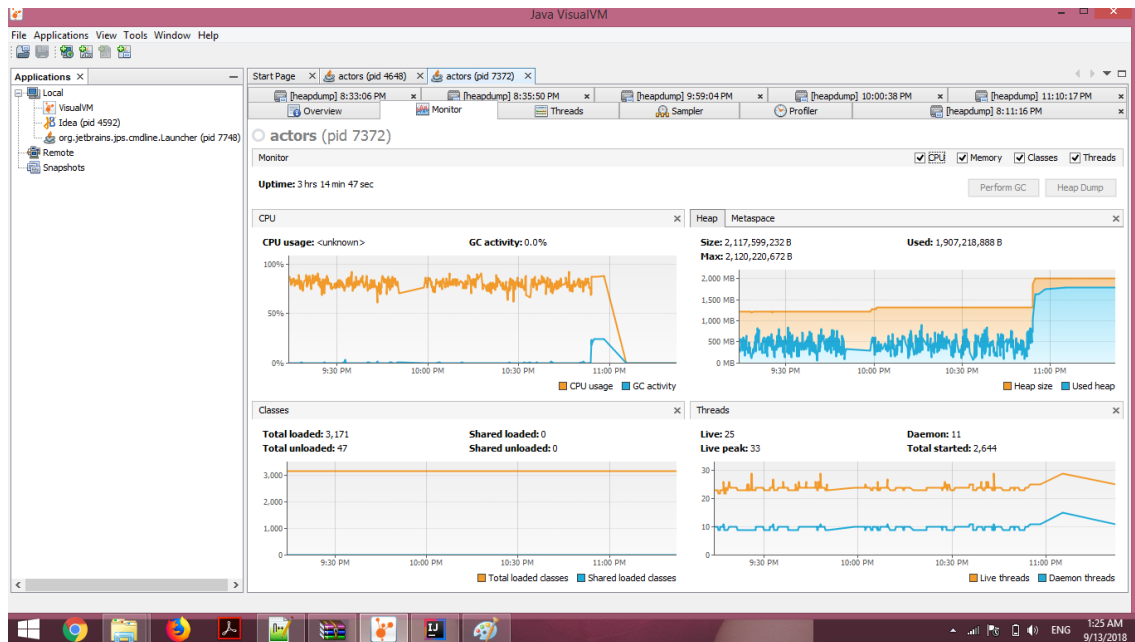
Приликом тестирања прве верзије програма дешавало се да програм избаци изузетак: *java.lang.OutOfMemoryError: Java heap space*. До овог изузетка долази када Јава виртуелна машина нема довољно слободне хип меморије. Оно што је било збуњујуће при појављивању ове грешке је што се она манифестовала веома непредвиђено: некад након три сата рада програма, некад након десет минута, а некада се није ни манифестовала. Честим исписивањем количине слободне хип меморије заиста се видало да се у неким случајевима меморија редовно ослобађа, док у другима након одређеног времена количина слободне

меморије пада све док се не деси избацавање изузетка. Овакво понашање упућивало је на цурење меморије које се не манифестује увек.

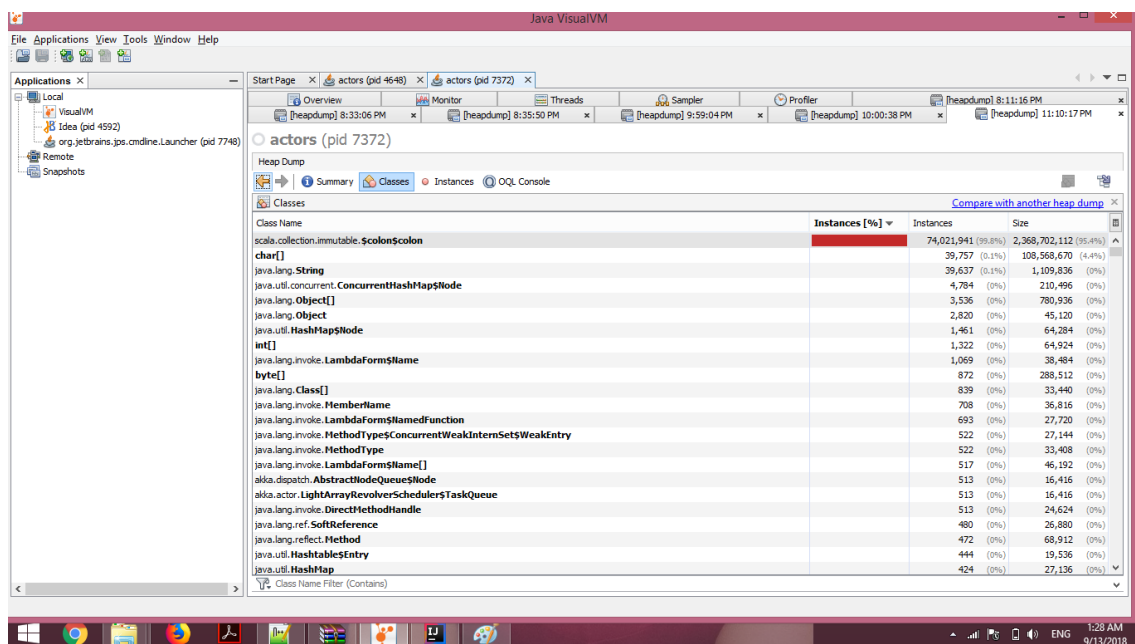
Експлицитно постављање референци објеката на *null*, и иницирање сакупљача отпадака из програма није помогло, тако да је наредни корак био покретање програма заједно са профајлером *VisualVM*. Профајлер омогућава лакше и јасније праћење рада програма и меморије, нарочито исписивањем алоцираних објеката у меморији када се деси изузетак.

На сликама 5.3 и 5.4 приказан је пример рада програма у профајлеру. За улазне параметре постављени су читач *Sumatra*, број -1 као број итерација, број 10 као број Извршилаца и број 1000 као број милисекунди. Прва слика приказује целокупан рад програма. Од посебне важности је секција *Heap* коју чини график који приказује количину слободне хип меморије у зависности од времена. Може се видети да је програм радио преко три сата, редовно ослобађајући меморију све док у једном тренутку количина меморије није нагло порасла до границе коју дозвољава Јава виртуелна машина. Када је избачен изузетак, забележени су објекти алоцирани на хипу, што је приказано на другој слици. Највећи део меморије заузимају инстанце класе *scala.collection.immutable.\$colon\$colon*, која представља непроменљиве листе у Скали. Програм је у једном тренутку заузео меморију великом количином листи чији је тип елемената био *Regex.Match* (што је такође прочитано из профајлера). Објекти овог типа представљају појављивања одређених регуларних израза. Регуларни изрази омогућавају претраживање различитих објеката у датотеци (како је објашњено у поглављу 5.4).

Начин на који је имплементирано претраживање и мењање објеката у првој верзији програма је могао бити разлог преоптерећења меморије листама. Тај начин је подразумевао да се сваки тип објекта у датотеци претражује функцијом *scala.util.matching.Regex.findAllMatchIn*, која проналази **сва** појављивања одговарајућег регуларног израза. Сва појављивања су смештана у листу, након чега се на основу броја појављивања бирао јако мали број објеката који ће бити промењени. Када је завршено са свим променама објеката, листа која садржи појављивања је постављена на *null*. Разлог недостатка меморије је могао бити тај да су листе биле превелике да би стале у меморију уколико је број појављивања одређених објеката био превелик. Такође, због конкурентности у програму, могло се десити да више нити у исто време захтева чување великих листи у меморији.



Слика 5.3: Приказ рада програма помоћу профайлера *VisualVM*



Слика 5.4: Приказ алоцираних објеката помоћу профайлера *VisualVM*

Друга верзија програма је имплементирана тако да избегне конструкцију и чување листи. Уместо претраживања свих појављивања објеката одједном, појављивања су тражена и чувана једно по једно. Коришћена је функција `scala.util.matching.Regex.findFirstMatchIn` која проналази само **прво** појављи-

вање регуларног израза тј. објекта. Када се пронађени објекат промени, случајно одређени део датотеке се прескаче, након чега се одатле поново тражи прво појављивање објекта одређеног типа. Тај поступак се понавља све до краја датотеке. На тај начин свака нит у меморији чува само по један, тренутно пронађени објекат.

Ипак, друга верзија програма није исправила грешку недостатка меморије. Грешка о недостатку хипа је наставила непредвиђено да се манифестује, појављујући се након различитог периода рада програма. Највећи део заузете хип меморије су и даље биле листе типа *Regex.Match*, иако у имплементацији не постоји део кода за који се може претпоставити да генерише овакве листе. Због тога се претпоставља да је проблем у структурама података које се користе за функционални део Скале.

Због грешке недостатка меморије тешко је упоредити прву и другу верзију програма. Тестирањем друге верзије није се видело побољшање, али се јасно поређење не може дати са сигурношћу због непредвиђеног понашања обе верзије. Друга верзија захтева више времена за обраду једне датотеке, што није значајно у случају мањих датотека, док је у случају већих датотека разлика приметна.

Иако проблем није отклоњен, обе верзије програма су неко време тестирале читаче и постигле одређене резултате. Тестирана су три читача: *Foxit*, *Sumatra* и читач *Slim*. Одабрани су мање познати читачи због претпоставке да су мање отпорни на грешке од познатијих читача као што је читач *Adobe*. Ниједна верзија програма није успела да пронађе грешку у читачима *Foxit* и *Sumatra*. Оба читача су успешно препознала сваку неисправну датотеку пријављујући грешку при отварању. Са друге стране, читач *Slim* се показао као веома слаб. Обе верзије програма су често изазивале грешке у његовом раду. Један начин на који је читач *Slim* показао слабост је гашење одмах при отварању неисправне датотеке. Обе верзије програма су детектовале овакво понашање и сачувале датотеке које су изазвале гашење читача. Други начин на који је читач *Slim* показао неправилно понашање је престанак реаговања након отварања датотеке. У том случају оперативни систем издаје обавештење да је читач престао да реагује. Програм то не успева да детектује јер је у том случају спољашњи процес који је задужен за отварање датотеке и даље активан.

Такође је важно напоменути да обе верзије програма пријављују грешке код читача *Slim* и онда када он ради како треба. Узрок томе је карактери-

стика читача *Slim* која је присутна и код читача *Adobe*. Наиме, оба читача гасе покренуту инстанцу читача када препознају неисправну датотеку, и издају обавештење о неисправној датотеци. Због тога што сами угасе инстанцу читача, програм то детектује као да је процес угашен због проналаска грешке. Зато је при тестирању читача *Slim*, за сваку пронађену неисправну датотеку потребно ручно проверити да ли заиста изазива грешку читача. Читаче са оваквим начином рада није погодно тестирати имплементираним програмом, али је у овом раду читач *Slim* ипак тестиран јер је пронађено доста правих грешака.

Глава 6

Закључак

Због великих потреба за паралелним извршавањем веома је важно упознати нове технологије, језике и библиотеке које се развијају како би олакшале развој захтевних програма. Зато су представљена функционална својства Скале и начин на који се у њој може имплементирати модел Актер помоћу библиотеке *Akka*. Идеја која стоји иза овог модела и поменута библиотека су се показале као веома интуитивне и лагане за рад. Имплементиран је програм који конкурентно и паралелно тестира читаче датотека у формату *PDF* помоћу технике расплинутог тестирања. Концепт размене порука је омогућио брзу и лаку имплементацију паралелизације.

Приликом расплинутог тестирања се за генерисање улазних параметара некад користе специфичне вредности, али генерално не постоје правила по којима се они генеришу. Због тога ова техника тестирања омогућава развој креативности и даје слободу програмеру. У овом раду су покретани читачи датотека са датотекама чији је садржај претходно измењен у програму.

Један од изазова који су карактеристични за конкурентно и паралелно извршавање је детектовање и исправљање грешака. Грешке које се појављују у конкурентним и паралелним програмима се често манифестују непредвиђено што се показало и у овом раду. Приликом извршавања програма за тестирање читача датотека је у различитим тренуцима долазило до недостатка хип меморије. Било је веома тешко открити узрок овакве грешке, тако да је било потребно извршавати програм уз помоћ профајлера. Профајлер је показао резултате који су показали да се узрок овакве грешке највероватније налази у имплементацији неких од метода Скалине библиотеке за рад са регуларним изразима.

Такође се показало и да није лако изазвати грешке у раду читача датотека. На интернету се брзо може доћи до разних познатих напада на програме тако да је већина данашњих читача доста јака и отпорна на нападе. Потребно је много истраживања, времена и труда за откривање грешака у њима. У раду су за тестирање одабрана три мање позната читача од којих се један показао као веома слаб јер је доста често престајао са радом.

Датотеке у формату *PDF* су веома комплексне и њихово дубље проучавање представља највећи простор за побољшање имплементираног програма. Бољим парсирањем датотека и мењањем њихових објеката на више начина постоје веће шансе за проналазак грешака у читачима. Програм се може побољшати и тако да омогући: детектовање престанка реаговања читача, покретање читача уз дибагер, мењање датотека без учитавања целог њиховог садржаја одједном, итд.

Библиографија

- [1] Best Fuzzing Tools 2017. <http://opaida.com/2017/08/12/best-fuzzing-tools-2017/>.
- [2] Description of race conditions and deadlocks. <https://support.microsoft.com/en-us/help/317723/description-of-race-conditions-and-deadlocks>.
- [3] Martin Odersky: Biography and current work. <https://people.epfl.ch/martin.odersky/bio?lang=en&cvlang=en>.
- [4] PDF Reference And Adobe Extensions To The PDF Specification. https://www.adobe.com/devnet/pdf/pdf_reference.html.
- [5] What is PDF? <https://acrobat.adobe.com/be/en/acrobat/about-adobe-pdf.html>.
- [6] Akka Documentation: Version 2.5.13. <https://doc.akka.io/docs/akka/current/index.html>, 2011-2018.
- [7] Adobe Systems Incorporated. Document management - Portable Document Format - Part 1: PDF 1.7. https://wwwimages2.adobe.com/content/dam/acom/en/devnet/pdf/PDF32000_2008.pdf, 2008.
- [8] Marcel Böhme, Van-Thuan, and Pham Abhik Roychoudhury. Coverage-based Greybox Fuzzing as Markov Chain. *ACM CCS 2016*, 2016.
- [9] Paul N. Butcher. *Seven concurrency models in seven weeks: when threads unravel*. The Pragmatic Bookshelf, Dallas, Tex., 2014.
- [10] Felice Cardone and J. Roger Hindley. History of lambda-calculus and combinatory logic. 2006.

- [11] GNUpdf contributors. Introduction to PDF. https://web.archive.org/web/20141010035745/http://gnupdf.org/Introduction_to_PDF, May 2010.
- [12] Lee Copeland. *A Practitioner's Guide to Software Test Design*. Artech House, Norwood, MA, U.S.A., 2004.
- [13] 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.
- [14] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based Directed Whitebox Fuzzing. *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on. 2009*, 2009.
- [15] Marius Herring. Concurrency made easy with Scala and Akka. <http://www.deadcoderrising.com/concurrency-made-easy-with-scala-and-akka/>, March 2015.
- [16] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. 1973.
- [17] Matt Hillman. 15 minute guide to fuzzing. <https://www.mwrinfosecurity.com/our-thinking/15-minute-guide-to-fuzzing/>, August 2013.
- [18] Milena Vujošević Jančić. Programske paradigme: Funkcionalna paradigma. http://www.programskijezici.matf.bg.ac.rs/ppR/2018/predavanja/fp/funkcionalno_programiranje.pdf.
- [19] Milena Vujošević Jančić. Verifikacija softvera: Dinamička analiza softvera. http://www.programskijezici.matf.bg.ac.rs/vs/predavanja/02_testiranje/02_testiranje.pdf.
- [20] Milena Vujošević Jančić. *Automatsko generisanje i proveravanje uslova ispravnosti programa*. PhD thesis, Matematički fakultet, Univerzitet u Beogradu, 2013.
- [21] Dejan Lukan. PDF File Format: Basic Structure. <https://resources.infosecinstitute.com/pdf-file-format-basic-structure/>, May 2018.
- [22] Saša Malkov. Materijali sa predavanja: Funkcionalno programiranje - Pojam funkcionalnih programskih jezika. <http://poincare.matf.bg.ac>.

- rs/~smalkov/files/fp.r344.2017/public/predavanja/FP.cas.2017.01.Uvod.p2.pdf.
- [23] Saša Malkov. Materijali sa predavanja: Informacioni sistemi - Bezbednost. <http://poincare.matf.bg.ac.rs/~smalkov/download.html?dpth=1&cap=Informacioni+sistemi&bp=is.r271.2018%2Fpublic&rp=%2Fpredavanja>.
- [24] Saša Malkov. Materijali sa predavanja: Razvoj softvera - Konkurentnost. <http://poincare.matf.bg.ac.rs/~smalkov/download.html?dpth=1&cap=Razvoj+softvera&bp=rs.r290.2018%2Fpublic&rp=%2FPredavanja>.
- [25] Bertrand Meyer. Seven Principles of Software Testing. *Computer*, 41:99–101, 8 2008.
- [26] John C. Mitchell and Krzysztof Apt. *Concepts in Programming Languages*, chapter 4.2, pages 57–67. Cambridge University Press, 2003.
- [27] 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.
- [28] Martin Odersky. Scala’s Prehistory. <https://www.scala-lang.org/old/node/239.html>, 2008.
- [29] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima Press, Walnut Creek, California, 2 edition, 2011.
- [30] Nilanjan Raychaudhuri. *Scala in action*. Manning Publications, 1 edition, 4 2013.
- [31] Marc HJ Romanycia and Francis Jeffry Pelletier. What is a heuristic? *Computational Intelligence*, 1(1):47–58, 1985.
- [32] Michael L. Scott. *Programming language pragmatics*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 3 edition, 2009.
- [33] 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.

- [34] Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal, C/C++ Users Journal*, December 2004.
- [35] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Pearson Education, One Lake Street, Upper Saddle River, NJ 07458, 2007.
- [36] Ari Takanen, Jared DeMott, and Charlie Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, 685 Canton Street Norwood, MA 02062, 2008.