SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 1696

# Izrada sustava Lusca za ispitivanje mrežnih aplikacija

Jure Rastić

Zagreb, studeni 2007.

*Zahvaljujem se prof.dr.sc. Vladi Glaviniću i mr.sc. Stjepanu Grošu na pruženoj prilici za upoznavanje problematike kroz praksu i stručnom vodstvu. Zahvaljujem se Lucijani na velikom strpljenju, razumijevanju i podršci tijekom izrade diplomskog rada.*

## Sažetak

*Ova diplomska radnja opisuje problematiku ispitivanja mrežnih aplikacija i tipične metode za automatizirano ispitivanje. Za potrebe definiranja ispitnog paketa koji sadrži pravila za izvođenje ispitnih slučajeva kreiran je dijalekt jezika XML, te je opisana potrebna funkcionalnost Lusca sustava za izvođenje samog ispitivanja. Prikazana je arhitektura sustava i korištenih protokola za komunikaciju između pojedinih komponenti arhitekture. Sustav je implementiran korištenjem programskog jezika Python.*

## Abstract

*This diploma thesis describes problems in testing network applications and typical methods for automated testing. For purpose of test package description which holds the rules for test case execution, dialect of XML language was created. Additionally, required functionality of Lusca system for test execution was set. Paper shows architecture of created system and network protocols used for communication between different elements in architecture. System is implemented in Python programing language.*

# Sadržaj

# 1. Uvod

Kako bi kreirali mrežnu aplikaciju koja je sigurna, stabilna i zadovoljava zadani protokol, razvojni tim mora konstantno prolazit kroz proces ispitivanja samog dizajna aplikacije kao i programske implementacije. Razvojni tim mora kreirati zadovoljavajuću ispitnu okolinu u kojoj se aplikacija izvodi kako bi se aplikaciju ispitalo. To zahtijeva postavljanje zadane mrežne topologije, instalaciju i postavljanje odgovarajućeg skupa programa potpore, te prenošenje i postavljanje razvijene aplikacije s konfiguracijskim datotekama na računala u mreži. Ispitivanje uključuje pokretanje aplikacije i nadzor rada, prikupljanje datoteka s rezultatima izvođenja i dnevnika aplikacije za daljnju analizu. Prikupljene datoteke se koriste za krajnji proces analize i verifikacije ispravnog ponašanja aplikacije u zadanim uvjetima.

Prije ulaska u samu fazu razvoja aplikacije preporuča se kreiranje dokumenta ispitne specifikacije. Taj dokument navodi uvjete pod kojima se ispitivanje provodi, te ciljeve koje aplikacija mora zadovoljiti. Na osnovi ispitne specifikacije kreira se grupa testova i pristupa se kreiranju pomoćnih skripti ili programa za automatizaciju ispitivanja. Ispitivanje aplikacije se konstantno provodi automatski ili ručno tijekom razvoja aplikacije za svaku inačicu kôda. Proces ispitivanja postaje alat pomoću kojeg se može dobiti uvid u stanje aplikacije: u kojoj se fazi razvoja nalazi, koliko je funkcionalnosti implementirano, a kroz vremenski dnevnik ispitivanja se može dobiti uvid u brzinu razvoja i ispravljanja greški. Nakon faze razvoja aplikacije ispitno okruženje s dnevnikom izvođenja može poslužiti kao preduvjet za izdavanje garancije o ispravnosti rada aplikacije.

Izrada i održavanje ispitnog okruženja time postaje bitan dio razvojnog procesa, pogotovo danas kad mrežne aplikacije predstavljaju važan dio informacijske infrastrukture, te moraju biti robusne, ispravne i efikasne. Ispitno okruženje može biti građeno iz početka ili korištenjem specijaliziranih alata u tu svrhu pri čemu se vodi računa o nadogradivosti i iskoristivosti rješenja u budućim projektima.

Ovaj diplomski rad je nastao iz potrebe da se kreira specijalizirani pomoćni alat za brzu izgradnju prilagođenog ispitnog okruženja s naglaskom na ispitivanje mrežnih aplikacija. Cilj alata je da pruži korisniku sredstva pomoću kojih bi se skratilo vrijeme razvoja automatiziranog rješenja za ispitivanje u scenarijima koji zahtijevaju složene računalne mreže s mnogo računala što izvršavaju više aplikacija koje je potrebno postaviti, pokrenuti, te paralelno nadzirati. U sustav je ugrađena podrška za rad sa sustavom za inačice kôda razvijane aplikacije, te podrška za prevođenje u izvršni kôd.

Sadržaj ove diplomske radnje podijeljen je po idućim poglavljima. U drugom poglavlju detaljnije se opisuje problematika automatiziranog ispitivanja mrežnih aplikacija i iznose zadatci postavljeni pred Lusca sustav. U trećem poglavlju nalazi se kratki pregled tehnologija korištenih za izradu sustava. Četvrto poglavlje sadrži detaljniji opis Lusca sustava – uvod u jezik za definiciju ispitivanja, arhitekturu sustava s komunikacijskim protokolima, te pregled procesa izvršavanja testa. Peto poglavlje zaključuje radnju s mislima autora o području primjene sustava i idejama za daljnje unaprjeđenje.

# 2. Automatizirano ispitivanje mrežnih aplikacija

Ispitivanje računalnih programa i aplikacija je nužan dio razvojnog procesa koji se provodi s ciljem otkrivanja nepravilnosti i programskih greški aplikacije. Ukoliko se ne provede adekvatno ispitivanje aplikacije posljedice mogu biti tragične. Uzmimo za primjer raketni lanser američke mornarice USS Vincennes koji je 1988. godine zbog nejasnog ispisa računalnog sustava zamijenio putnički zrakoplov Airbus 320 za ratni zrakoplov F-14 [2]. Pogreška je dovela do smrti 290 civila koji su putovali poviše Perzijskog Zaljeva. NASA-in projekt za ispitivanje klimatskih uvjeta na Marsu završio je neuspjehom 1999. godine radi pogreške u navigacijskom sustavu letjelice koji je slao vrijednosti u jedinicama engleskog sustava umjesto u metričkom sustavu [3]. Letjelica je zbog toga promašila ispravnu visinu orbite (letjela je na 57km umjesto na 140-150km) i tako bila uništena atmosferskim trenjem. Ukupni trošak projekta bio je 327.6 milijuna američkih dolara.

Ovo su samo neke od pogrešaka s tragičnim posljedicama koje su mogle biti spriječene adekvatnim ispitivanjem ugrađenih računalnih sustava. Većina računalnih programa nema priliku da upravlja tako bitnim sustavima te posljedice greški takvih programa često se zanemaruju. Mnogi bi rekli i na sreću, pošto se stiče općenit dojam da su računalni sustavi nepouzdani.

Razlog za nedovoljno i ne adekvatno ispitivanje aplikacija najčešće je nedostatak ljudskih ili računalnih resursa, te nepostojanje odgovarajućih alata koji bi smanjili troškove ispitivanja [4]. Tako dolazimo do pravih izazova pri ispitivanju aplikacija:

- smanjenje troškova samog procesa ispitivanja
- efikasnost ispitivanja – problem s mjerenjem kvalitete
- nepotpuna i neformalna specifikacija podložna promjenama
- smanjenje troškova upravljanja programskim pogreškama
- izgradnja i održavanje ispitnih alata
- integracijsko ispitivanje jedne aplikacije iz programskog paketa
- upravljanje s novim inačicama aplikacije
- primjena ispravaka nad kôdom itd.

Proces razvijanja i održavanja okruženja za automatizirano ispitivanje tijekom razvojnog procesa postaje zadatak kojeg se pojedinci u razvojnom timu nerado prihvaćaju. Stoga je potrebno imati rješenja za što jednostavniju i efikasniju izgradnju prilagođenog sustava za automatsko ispitivanje. Automatizacija pojedinih koraka u procesu osmišljavanja i samog izvođenja ispitivanja postaje zadatak kojeg obavlja računalo i time eliminira čovjeku zamoran posao.

U idućim poglavljima se razrađuju pristupi automatskom ispitivanju računalnih programa općenito i prelazak na ispitivanje mrežnih aplikacija. Prvo ćemo reći nešto o metodologiji ispitivanja računalnih programa, te razmotriti probleme i predložiti rješenja za ispitivanje mrežnih aplikacija. Na kraju poglavlja su definirani zahtjevi nad Lusca sustavom kako bi se ostvarila potrebna funkcionalnost.

## 2.1. Metodologije ispitivanja

Kako bi proces ispitivanja postao jednom vrstom standarda i uobičajenom praksom pri razvoju računalnih programa postoji cijela grana računalnih znanosti posvećena upravo razvoju ispitnih metoda. Ova diplomska radnja neće ulaziti u detalje formalnih metoda i postupaka u oblikovanju računalnih sustava, već ćemo samo navesti osnovne značajke bitne za diplomsku temu.

Ispitne metode se mogu podijeliti u dvije osnovne grupacije:

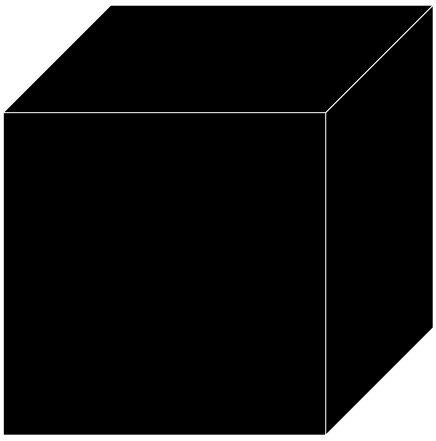
- po kriterijima ispitivanja
  - ispitivanje funkcionalnosti
  - pronalazak nepravilnosti (računalni uzrok)
  - pronalazak grešaka (ljudski uzrok)
- po izvoru informacija
  - princip bijele kutije
  - princip crne kutije

Kod prve grupacije metoda kreiranju ispitnih slučajeva se pristupa s obzirom na cilj ispitivanja. Tako ispitni uzorci mogu biti korišteni da bi se potvrdila funkcionalnost, da bi se pronašla nepravilnost ili da bi se pronašla greška u implementaciji same aplikacije.

Druga grupacija ispitnih metoda se češće razmatra. Kako bi sproveli ispitivanje zadanog objekta (aplikacije) moramo doći do informacija koje nam služe za verifikaciju ispravnosti rada aplikacije. Ovisno o izvoru iz kojeg crpimo te informacije, ispitne uzorke dijelimo po principu bijele (ili prozirne) [5] i crne kutije [6]. Možemo reći da se radi načinu promatranja aplikacije. U idućim poglavljima ćemo detaljnije opisati ove metode.

### 2.1.1. Ispitivanje metodom crne kutije

Prilikom promatranja aplikacije kao crne kutije informacije crpimo iz aplikaciji vanjskih elemenata kao što su izlazne datoteke ili ostvarene promjene u okruženju pa se takvo ispitivanje često zove i funkcionalno ili ispitivanje po specifikaciji. Skica pogleda na aplikaciju je prikazana slikom 2.1.



Slika 2.1: Aplikacija kao crna kutija

U ovom slučaju nemamo nikakvih podataka što se događa s internim strukturama aplikacije niti u kojoj se fazi izvršavanja program nalazi. Sve informacije koje možemo dobiti su izlazne informacije aplikacije u radu. Planiranje ispitivanja aplikacije kod ove metode može početi dosta rano u procesu razvoja aplikacije pošto se i razvoj i ispitivanje ovom metodom zasnivaju na specifikaciji aplikacije.

Ispitivanje se vrši na način da se aplikacija pokrene u točno određenim uvjetima koji postaju ulazni podatci ispitivanja. Nakon ispitivanja aplikacije se uzimaju izlazni podatci i analiziraju. Određena kombinacija ulaznih i izlaznih podataka se tumači kao neispravna ukoliko se ta kombinacija podataka ne podudara sa specifikacijom. Specifikaciju rada aplikacije možemo definirati sljedećom rečenicom:

*Aplikacija **A** mora se ponašati kao **P** u okruženju **O** pod scenarijem **S** i ulaznim podacima **U**, a rezultat mora biti **R**.*

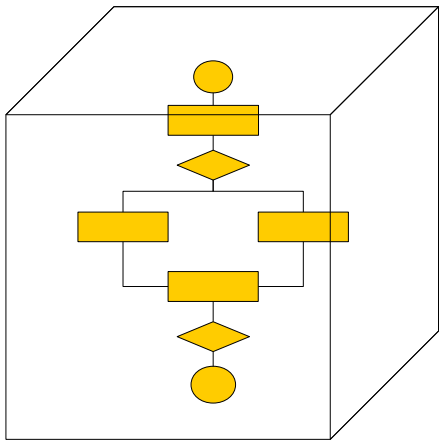
Pri čemu kratice predstavljaju:

- *A* – razvijena aplikacija
- *P* – skup događaja na računalu koji definira znakove ponašanja
- *O* – značajke računala na kojem se aplikacija izvodi
- *S* – akcije koje treba poduzeti na računalu
- *U* – konfiguracijske datoteke aplikacije i parametri
- *R* – rezultat izvođenja, izlazne datoteke aplikacije i slično

Tada su ulazni podatci *O*, *S* i *U*, a izlazni *P* i *R*. Kod ove metode ispitivanja predaje se velika važnost odabiru ulaznih podataka za ispitivanje kod definiranja ispitnih slučajeva, te njihova podjela na raspone koji mijenjaju funkcionalnost aplikacije.

## 2.1.2. Metoda bijele kutije

Kod promatranja aplikacije kao bijele kutije analiziramo podatke dobivene na osnovi poznavanja strukture programa pa se takvo ispitivanje često zove i strukturalno ispitivanje. Skica pogleda na aplikaciju je prikazana slikom 2.2.



Slika 2.2: Aplikacija kao bijela kutija

Ispitni slučajevi se generiraju na osnovi znanja o strukturi programskog kôda. Grade se ispitni uzorci kako bi se prošlo svim mogućim putovima strukture programa. Potrebno je proći svim nezavisnim putovima u pojedinom modulu aplikacije predavajući podatke koji će rezultirati istinitim i neistinitim logičkim odlukama u točkama grananja te time usmjeriti tok izvođenja kroz pojedine grane. Potrebno je ispitati svaku petlju za domenu njezinoga djelovanja, te ispitati unutarnje strukture s odgovarajućim operacijama kako bi se osigurala njihova ispravnost.

Mogli bi se zapitati zašto je bitno vršiti ispitivanje metodom bijele kutije kada su ispunjeni zahtjevi nad funkcionalnosti i programskim sučeljima te isti potvrđeni metodom ispitivanja aplikacije kao crne kutije. Pa, bitno je vršiti ispitivanja po ovom principu jer su česte logičke pogreške u pisanju programskog kôda, neke intuitivne pretpostavke na kojima se temelje određene strukture u nekim drugim okruženjima ne vrijede ili jednostavno radi toga što postoji vjerojatnost od tipografskih pogrešaka koja se neće detektirati od strane čovjeka niti prevoditelja u izvršni kôd.

Problem koji se nameće kod ove metode ispitivanja je vremenska složenost. Uzmimo jednostavnu aplikaciju s programskom strukturom od jedne petlje s 20 iteracija koja sadrži kôd s 5 različitih putova. Dobivamo $5^{20}$ ($10^{14}$) različitih scenarija izvođenja. Ako nam po svakom scenariju treba jedna milisekunda za izvršavanje, ukupno ispitivanje će trajati oko 3170 godina. Problem se rješava naprednijim metodama odabira putova koje je potrebno ispitati zavisno o promijeni vrijednosti korištenih varijabli, no proces je i dalje vremenski skup i često neprimjenjiv u praksi.

## 2.2. Automatizirano ispitivanje aplikacija

Automatizacija je nužno sredstvo kojim se ostvaruje temeljito ispitivanje razvijane aplikacije u bilo kojem trenutku razvoja. Kreira se specijalizirano rješenje koje će se brinuti za postavljanje, pokretanje i verifikaciju ispravnosti svih ispitnih slučajeva u zadanim vremenskim intervalima.

Neka od zastrašujućih činjenica vezanih uz ispitivanje aplikacija:

- ništa ne može garantirati ispravnost (barem ne u realnom vremenu)
- neodlučnost i u najjednostavnijim programima
- kombinatorna eksplozija kod ispitivanja metodom bijele kutije
- statistička istraživanja su preskupa
- sistematski pristup ispitivanju je **također** podložan greškama (ispitati pristup)

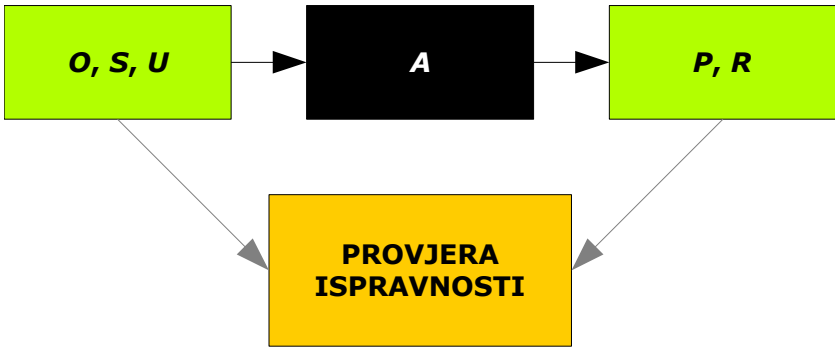
U borbu protiv tih činjenica ulazimo s:

- teorijom automata s konačnim brojem stanja
- ispitivanjem po specifikaciji
- iskustvom iz ispitivanja sličnih programa

Teorija automata s konačnim brojem stanja nam služi kao podloga pri dizajniranju sustava za formalnu verifikaciju i modeliranje programskog kôda, generiranju ulaznih podataka s odgovarajućim rezultatima i slično. Ispitivanje po specifikaciji pomaže utoliko što ograničava skup ulaznih podataka na mali podskup s kojima je moguće ispitati razvijenu

aplikaciju u realnom vremenu. Ukoliko će se aplikacija koristiti samo po specifikaciji, trebalo bi biti dovoljno ispitati zadovoljenje uvjeta u specifikaciji. Iskustvo u ispitivanju sličnih ili istih aplikacija pomaže procesu u smislu poznavanja programske podrške koju aplikacija koristi, tipičnih pogreški pri razvoju slične aplikacije koje treba istražiti ispitivanjem, te eventualnim smanjivanjem skupa ulaznih podataka na onoliki podskup koji se iskustvom pokazao dovoljan. U praksi se uglavnom radi o odstupanju iz svijeta formalne verifikacije i ulazi u ograničenje ili uvjeta u kojima će aplikacija raditi ili funkcionalnosti koja će biti ispitana. U svakom pogledu, da bi se imalo uvid u stanje ispravnosti rada aplikacije potrebno je vršiti ispitivanje što češće s definiranim skupom ulaznih podataka.

Ovaj diplomski rad će se fokusirati na metodu ispitivanja aplikacije kao crne kutije ili ispitivanju po specifikaciji. Dijagram ispitivanja aplikacije kao crne kutije je prikazan na slici 2.3.




Slika 2.3: Ispitivanje aplikacije metodom crne kutije

Pri ovoj metodi automatizacija procesa ispitivanja se svodi na automatizaciju postavljanja okruženja aplikacije (*O*) i izvođenje zadanog scenarija (*S*), dok čovjek generira ulazne podatke (*U*). Daljnja automatizacija je moguća pri verifikaciji izlaznih podataka (*P* i *R*) nakon ispitivanja. Sustav za automatsko izvođenje ispitivanja dobiva od čovjeka upute za postavljanje okruženja, samu ispitivanu aplikaciju s njenim konfiguracijskim datotekama te pravila za verifikaciju ispravnog ponašanja. Nakon toga sustav se brine za pokretanje ispitivanja aplikacije za svaku novu inačicu i prijavljuje pronađene greške.

Važno je primijetiti da opisani sustav sam po sebi nije sposoban pronalaziti greške bez da mu čovjek definira pravila koja aplikacija mora zadovoljiti. Stoga, ovaj sustav pronalazi greške koje čovjek očekuje ili potvrđuje ispravnost očekivanoga ponašanja. Njegova uloga u procesu razvoja aplikacije je bitna i utoliko što služi kao alat za pregled statusa projekta – koliko je postotak funkcionalnosti iz specifikacije zadovoljen, koliki ne. Nakon razvojne faze projekta sustav se koristi kao preduvjet za garanciju ispravnosti i kvalitete aplikacije gdje tijelo koje izdaje garanciju provjerava da li je ispitivanje ispravno provedeno.

Kako bi sustav mogao detektirati ispravno ponašanje ispitivane aplikacije potrebno ga je programirati pravilima za:

- postavljanje ispitne okoline,
- generiranje ulaznih podataka aplikacije,
- upravljanje scenarijem izvođenja,
- detekciju ispravnih rezultata.

Podjela ulaznih podataka se uobičajeno radi po funkcionalnosti aplikacije, te nadalje po scenarijima uporabe određene funkcionalnosti. Prva razina podjele ulaznih podataka se naziva podjelom po *ispitnim paketima*. Na primjeru ispitivanja aplikacije za uređivanje teksta ispitni paketi bi mogli biti:

- rad aplikacije s datotekama,
- rad aplikacije s korisničkim sučeljom,
- bojanje sintakse teksta.

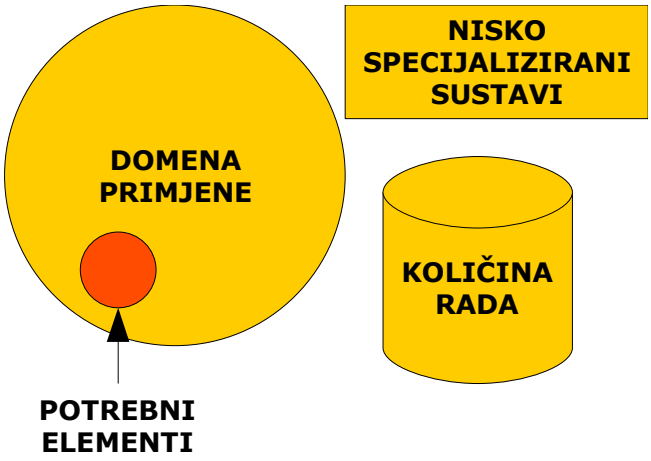
Daljnja podjela ulaznih podataka unutar ispitnog paketa se vrši po određenom scenariju korištenja jedne od funkcionalnost te se ispod ove razine podjela ne vrši. Ova razina grupacije ulaznih podataka se naziva *ispitni slučaj*. Na primjeru ispitivanja aplikacije za uređivanje teksta ispitni paketi s popisom ispitnih slučajeva bi mogli izgledati na sljedeći način:

- rad aplikacije s datotekama
  - otvaranje datoteke i učitavanje
  - spremanje promijenjene datoteke
  - spajanje dviju datoteka
- rad aplikacije s korisničkim sučeljom
  - izbornik za rad s datotekama
  - izbornik za opcije pregleda teksta
  - izbornik za korisničku pomoć
  - izbornik za kontekstnu pomoć
- bojanje sintakse teksta
  - bojanje teksta obične tekstualne datoteke
  - bojanje teksta datoteke programskog jezika C
  - bojanje teksta datoteke programskog jezika FORTRAN

Primjenom metode *podijeli pa vladaj* dobiva se segmentacija ispitnih podataka koja se koristi za lakši uvid u funkcionalnosti koje treba implementirati i ispitati.

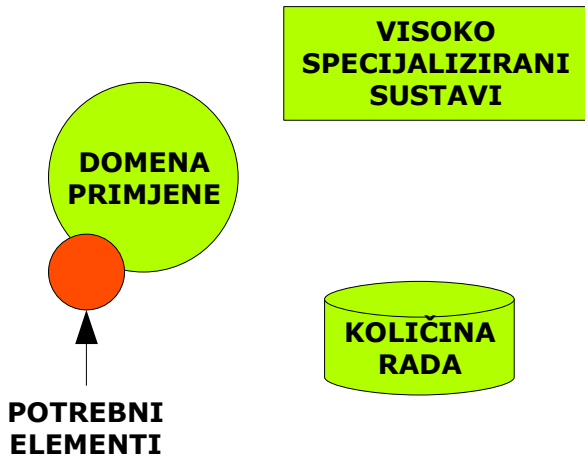
Razvojni tim kreira ulazne podatke za pojedini ispitni slučaj i pravila sustava za automatizirano ispitivanje. Radi se osnovni kostur aplikacije i pokreće prvo ispitivanje. Prilikom prvog ispitivanja ne uspijeva niti jedan ispitni slučaj. Razvojni proces se dalje odvija s ciljem zadovoljenja definiranih funkcionalnosti. Nakon svake nove inačice programskog kôda u kojoj je izvedena neka od funkcionalnosti, ispitivanje se pokreće sa svim paketima i dobiva se uvid u ispravnost implementacije. Taj proces se dalje odvija ciklički sve do trenutka kada su svi ispitni slučajevi zadovoljeni. Tada je aplikacija spremna za uporabu po istoj specifikaciji po kojoj su građeni ispitni slučajevi. Ukoliko se aplikacija koristi izvan domene definirane specifikacijom (na primjer bojanje teksta datoteke programskog jezika Python) vrlo su vjerojatne greške u radu aplikacije koje nisu ni tražene tijekom ispitivanja.

Do sad smo razmatrali općeniti sustav za automatizirano ispitivanje kojemu je za ispravan rad potrebna poprilična količina informacija od čovjeka. U praksi postoje specijalizirani sustavi za određene tipove aplikacija koji imaju ugrađenu podršku za podizanje razine automatizacije jednog ili više koraka procesa ispitivanja. Ugrađena podrška se temelji na iskustvenom znanju stečenom kroz ispitivanje aplikacija iz neke specifične domene. Pomoću dodatnih svojstava tih sustava olakšani su recimo zadatci generiranja poznatih ulaznih podataka ili određivanje očekivanih rezultata izvođenja. Unaprjeđenje i olakšanje procesa definiranja ispitnih slučajeva se može postići ugrađivanjem mehanizama za jednostavnije definiranje pravila izvođenja scenarija tipičnih za tu domenu aplikacije i verifikacije ispravnosti rada. Ovisno o razini prilagodbe sustava određenoj problematici iz domene, sustav postaje specijaliziran i iskoristiv u manjoj domeni, pa tako dijelimo sustave na one s visokom i niskom razinom specijalizacije. Sustav s niskom razinom specijalizacije prikazan grafički usporedno s domenom primjene i količinom potrebnog rada je prikazan na slici 2.4.




Slika 2.4: Sustav s niskom razinom specijalizacije

Na prethodnoj slici se može primijetiti veća količina potrebnog rada da bi se ostvarilo automatizirano rješenje za ispitivanje aplikacije naspram količine rada u sustavu s visokim stupnjem specijalizacije prikazanom na slici 2.5.




Slika 2.5: Sustav s visokom razinom specijalizacije

Na ovoj slici možemo primijetiti da je potrebna manja količina potrebnog rada, no da neki elementi potrebni za potpunu automatizaciju ispitnog procesa izlaze iz domene nad kojom je sustav primjenjiv. U tom slučaju potrebno je uložiti nešto dodatnog rada u proširenje domene primjene kroz uporabu dodatnih alata kako bi se postigla potrebna razina automatizacije rješenja. Kod sustava koji imaju nižu razinu specijalizacije preporuča se dio uloženog rada iskoristiti na pronalazak i uporabu dodatnih alata koji će specijalizirati uporabu sustava te time dodatno smanjiti potrebnu količinu rada. Pokušava se odrediti zadovoljavajuća razina specijalizacije koja neće ograničiti primjenu sustava za ispitivanje na premali broj aplikacija, a s druge strane da se postigne što veća razina pojednostavljenja ispitnog procesa.

Nakon ove teorijske rasprave o sustavima za automatsko ispitivanje aplikacija pozabavit ćemo se nešto detaljnije problemom postavljenim pred ovaj diplomski rad. Potrebno je izgraditi sustav za automatizirano ispitivanje mrežnih aplikacija i pronaći način za olakšavanje procesa definiranja ispitnih slučajeva u takvim uvjetima.

## 2.3. Problematika ispitivanja mrežnih aplikacija

Mrežne aplikacije imaju jednu bitnu razliku naspram aplikacija koje se izvode samostalno na jednom računalu – jedna instanca mrežne aplikacija je u interakciji s drugim instancama iste ili neke druge aplikacije čiji se proces odvija na nekom računalu u računalnoj mreži. Komunikacija između više instanci aplikacije se odvija putem mrežnih protokola stoga njezino stanje ovisi o većem kontekstu od njezine primarne okoline – računala na kojem se izvodi. Kako bi se stekao bolji uvid u ponašanje aplikacije i postiglo ispitivanje ispravnosti rada potrebno je promatrati rad aplikacije s razine računalne mreže u kojoj se nalazi. Kako ta razina implicitno mijenja okolinu izvođenja aplikacije potrebno je proširiti i definiciju pripreme okoline u kojoj se aplikacija izvodi. Rezultati izvođenja postaju raspoređeni po računalnoj mreži. Također se javlja problem verifikacije ispravnosti rada aplikacije i upravljanja scenarijem. U tom slučaju prethodno iznesena stavka o specifikaciji rada aplikacije ostaje ista, no mijenja se značaj pojedinih stavki:

*Aplikacija **A** mora se ponašati kao **P** u okruženju **O** pod scenarijem **S** i ulaznim podacima **U**, a rezultat mora biti **R**.*

Pri čemu kratice sada predstavljaju:

- *A* – razvijena aplikacija
- *P* – skup događaja unutar računalne mreže koji definira znakove ponašanja
- *O* – okruženje u kojem se aplikacija izvodi je računalna mreža
- *S* – slijed akcija koje treba poduzeti unutar računalne mreže
- *U* – konfiguracijske datoteke i parametri svih aplikacija u mreži, postavke mreže
- *R* – rezultat izvođenja, izlazne datoteke, mrežni promet i slično

Kako bi izgradili specijalizirani sustav za ispitivanje mrežnih aplikacija potrebno je uhvatiti se u koštac s problemima koji nastaju kao posljedica okoline koja je distribuirana po računalnoj mreži:

- postavljanje mrežnih parametara računalne opreme
- distribucija konfiguracijskih datoteka i ostalih resursa na računala u mreži

- pokretanje i nadzor udaljenih aplikacija
- upravljanje tokom izvođenja ispitnog slučaja
- verifikacija ispravnog rada aplikacije
- sakupljanje rezultata izvođenja ispitnog slučaja.

Za distribuciju i prikupljanje datoteka moguće je iskoristiti postojeće mrežne protokole te gotova rješenja i relativno jednostavno izgraditi vlastiti sustav za upravljanje datotekama na računalnoj mreži. Također postoje rješenja za postavljanje računalne mreže iz neke unaprijed definirane konfiguracije. Najveći problem ostaje upravljanje tokom testa u distribuiranoj okolini i integrirano rješenje koje će ponuditi upravljanje resursima i konfiguracijom mreže na jednom mjestu.

Treba naglasiti možda u ovom trenutku da su problemi postavljanja i ispitivanja samih računalnih mreža vrlo veliko područje u koje ovaj diplomski rad ne ulazi već samo razmatra neke praktične probleme sa stajališta izgradnje sustava za automatizirano ispitivanje mrežnih aplikacija.

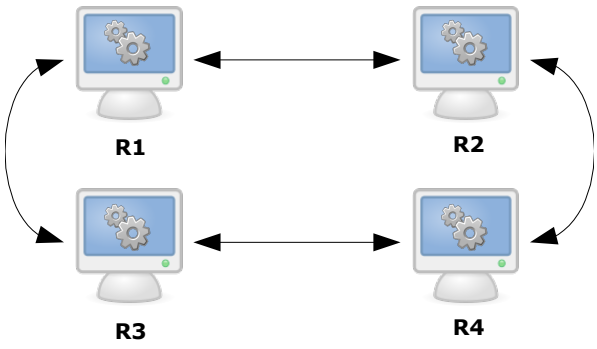
Potrebno je dakle izgraditi specijalizirani sustav za ispitivanje aplikacija koji pruža sredstva što olakšavaju izradu ispitnih slučajeva rješavajući predstavljene probleme. Na taj način sustav bi trebao smanjiti potrebnu količinu rada i vremena za automatizirano ispitivanje mrežnih aplikacija. Određen je primarni problem upravljanja tokom izvođenja testa u računalnoj mreži, a detaljnija razrada je prikazana u idućem potpoglavlju.

### 2.3.1. Tok testa u distribuiranoj okolini

Priroda aplikacije koja se izvršava u distribuiranoj okolini na računalima u mreži predstavlja poseban zadatak sustavu za ispitivanje aplikacija. Ukoliko sustav ispituje aplikaciju na jednom računalu situacija je jednostavna jer su svi resursi kojima sustav mora upravljati na tom računalu. Kod mrežne aplikacije događaji se zbivaju u računalnoj mreži i često je potrebno poduzimati akcije na udaljenim računalima da bi se postigao željeni ispitni tok. Prilikom definiranja zamišljenog ispitnog slučaja, rečenica koja opisuje potrebne radnje bi mogla izgledati:

*Pokreni aplikaciju na računalu R1. Ako je aplikacija uspješno pokrenuta, pokreni aplikaciju na računalu R2 i provjeri dnevnik izvođenja... Ispitni slučaj je zadovoljen ukoliko se u dnevniku izvođenja aplikacije na računalu R4 pojavio znakovni niz "podatci primljeni".*

Okolina i opisana situacija prikazana je pojednostavnjeno na slici 2.6.




Slika 2.6: Distribuirani tok upravljanja

Informacije o stanju aplikacije se moraju prenositi s jednog računala na drugo kako bi se na tom računalu poduzela odgovarajuća akcija. Aplikacija se izvodi na svakom od računala i ovisno o fazi ispitnog slučaja potrebno je pokretati druge aplikacije, provjeravati izlazne podatke ili pretraživati dnevnik izvođenja određene instance aplikacije na nekom od računala.

Za potrebe ove radnje smišljen je jezik kojim se definiraju pravila koja će definirati proces izvođenja ispitnog slučaja. Pri tome se koristi metoda programiranja upravljanom događajima [8]. Programiranje upravljano događajima je računalna programska paradigma u kojoj je tok izvođenja programa definiran korisničkim akcijama ili porukama od drugih programa. Ova paradigma se koristi primjerice u:

- operacijskim sustavima – sklopovski i programski prekidi
- programiranju grafičkih sučelja – procedure *onClick*, *onRelease*, *onMouseOver*...
- programiranju s utičnicama – procedura za primanje podataka (*onData*...)

Programiranje upravljano događajima je paradigma najbliža prirodnom načinu definiranja toka testa u ovakvoj okolini. Potrebno je iskazati koju akciju treba izvršiti ukoliko se određeni događaj pojavi. Kako je okolina distribuirana na računalnoj mreži dodat ćemo još podatak na kojem računalu da se akcija izvrši i dobivamo jezik kojim možemo opisati proces ispitivanja. Ukoliko je akciju potrebno izvršiti na istom računalu gdje se događaj zbio, kažemo da događaj *pali lokalnu akciju*. Ukoliko je akciju potrebno izvršiti na nekom drugom računalu u sustavu tada kreiramo signal i time taj događaj *pali udaljenu akciju*. Primjer u pseudo-jeziku bi mogao izgledati na sljedeći način:

```
01 computer1 job one(on start_test)
02   run(my_app)
03     on(start_success)
04       create signal(signal_one)
05
06 computer2 job two(on signal_one)
07   run(my_app)
08     on(event1)
09       run(check_logs)
10         on(logs_ok)
11           create signal(test_passed)
12         on(logs_bad)
13           create signal(test_failed)
```

Isječak kôda opisuje dva zadatka (*one* i *two*) za dva računala u ispitnoj okolini (*computer1* i *computer2*). U prvoj liniji kôda vidimo zaglavlje zadatka *one* za računalo *computer1*. Zadatak se izvršava kada se primi ugrađeni signal *start_test* te definira akciju koju treba poduzeti, a to je pokretanje aplikacije *my_app*. Prati se ponašanje aplikacije i na događaj *start_success* kreira se signal *signal_one* prikazano u trećoj i četvrtoj liniji kôda. Taj signal pokreće zadatak *two* na računalu *computer2* sa zaglavljom akcije u šestoj liniji. Zadatak definira pokretanje programa *my_app* koji generira događaj *event1*. Akcija za taj događaj je pokretanje programa *check_logs* u devetoj liniji. Ukoliko taj program generira *logs_ok* kreira se ugrađeni signal *test_passed* koji predstavlja znak ispravnog izvođenja ispitnog slučaja.

Ako program generira *logs_bad* događaj, kreira se ugrađeni signal *test_failed* koji predstavlja znak sustavu da ispitni slučaj nije zadovoljen.

Na ovom primjeru možemo reći da događaj *start_success* generiran na računalu *computer1* pali udaljenu akciju pokretanja programa na računalu *computer2*. Za događaj *event1* kažemo da pali lokalnu akciju pokretanja programa *check_logs*. Ispitni slučaj je zadovoljen (ili je prošao) ukoliko je sustav generirao *test_passed* signal, a ispitni slučaj nije zadovoljen (ili je pao) ukoliko se pojavi signal *test_failed*.

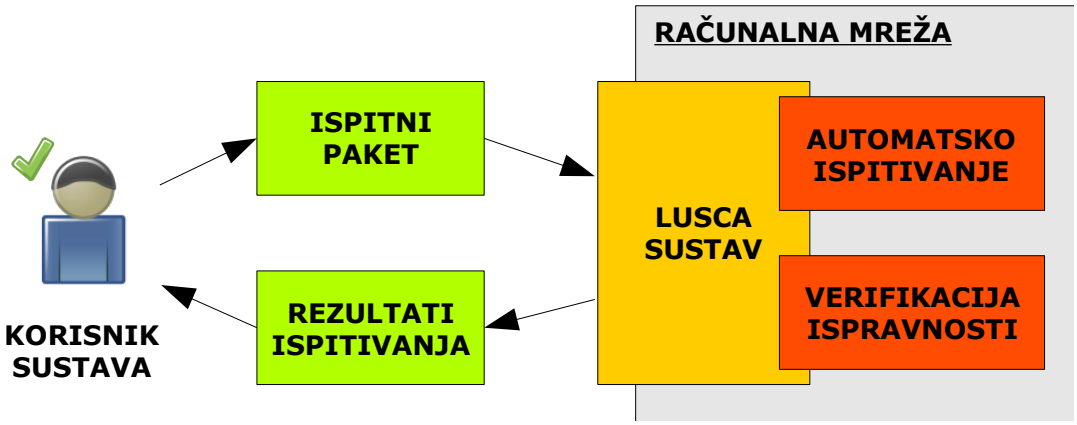
## 2.4. Zahtjevi nad Lusca sustavom

Tijekom prethodne teorijske razrade kreirali smo podlogu za jasno definiranje ciljeva Lusca sustava. Odabrana je programska paradigma za izradu jezika kojim će se postaviti pravila za upravljanje tokom ispitivanja. U ovom poglavlju ćemo zaključiti teorijsku razradu listom potrebnih funkcionalnosti i opisom idejnoga rješenja.

Sustav mora pružiti sredstva za automatizirano ispitivanje mrežnih aplikacija. Time se moraju olakšati sljedeći zadatci:

- postavljanje računalne mreže
- upravljanje ulaznim datotekama za ispitivanje
- prevođenje izvornog kôda razvijane aplikacije u izvršni
- upravljanje inačicama izvornog kôda
- izvršavanje definiranih pravila za ispitivanje
- jednom definirani ispitni paketi se mogu bilo kad izvršiti
- upravljanje datotekama nastalih tijekom ispitivanja.

Uloga sustava u procesu automatskog ispitivanja aplikacije je da se brine o svim resursima uključenim u ispitivanje. Korisnik kreira datoteku u definiranom formatu koja sadrži pravila ispitnog paketa, definiciju aplikacije i datoteke potrebne za izvođenje ispitivanja. Datoteku predaje sustavu na izvršavanje. Sustav izvođenjem pravila vrši ispitivanje i verifikaciju ispravnosti rada aplikacije. Nakon toga sustav skuplja nastale datoteke i predaje rezultate ispitivanja nazad korisniku. Skica ovog procesu prikazana je slikom 2.7.




Slika 2.7: Dijagram uporabe Lusca sustava

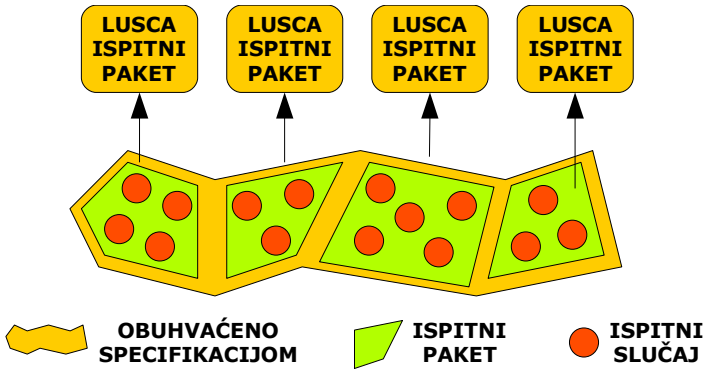
Kako bi sustav mogao upravljati računalima u mreži očigledno je da i on sam mora biti raspoređen po mreži. Za komunikaciju između elemenata sustava koristit će se mrežni protokol za udaljene pozive, a za prijenos datoteka odgovarajući mrežni protokol. Iduća potpoglavlja opisuju sadržaj samog ispitnog paketa i pristup izradi sustava za izvršavanje ispitnog paketa.

### 2.4.1. Lusca ispitni paket

Kako bi sustav mogao ispitivati razvijenu aplikaciju potrebno mu je predati informacije o:

- aplikaciji
  - kako do nje doći
  - kako je prevesti u izvršni kôd
  - koju inačicu kôda ispitati
- pomoćnim programima
  - lokacija skripti ili programa za izvođenje ispitivanja
  - ulazni parametri programa
- datotekama korištenim u sustavu
  - izvorna lokacija datoteka
  - odredišna lokacija datoteka
- računalnoj mreži koja se koristi
  - popis računala
  - mrežna oprema na računalima
  - parametri mrežne opreme

Sve informacije o navedenim resursima se nalaze u jednoj datoteci koja ih opisuje. Uz informacije o resursima, sustavu se predaje i lista ispitnih slučaja s pravilima za izvođenje i verifikaciju ispitivanja unutar iste datoteke. Pravila koriste prethodno definirane resurse kako bi kreirali zadane scenarije. Sustavu se može predati više ispitnih paketa koji zajedno sačinjavaju sve scenarije uporabe aplikacije koji su definirani ispitnom specifikacijom. Podjela specifikacije na ispitne pakete i ispitne slučajeve je prikazana na slici 2.8.




Slika 2.8: Podjela specifikacije na ispitne pakete

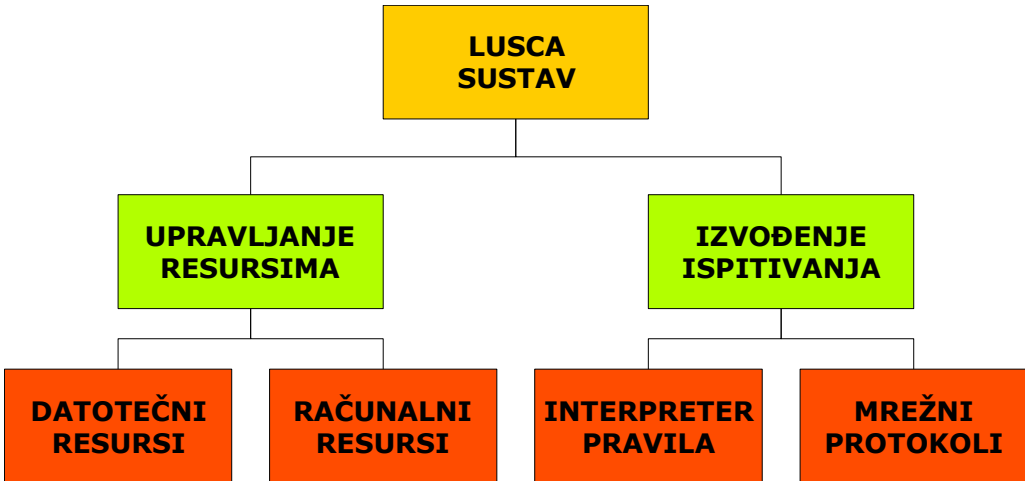
Datoteku koja sadrži informacije o resursima i pravila za izvođenje ispitivanja nazivamo **opisom ispitnog paketa**. Definiramo nadalje **Lusca ispitni paket** kao zapakiranu datoteku koja sadrži opis ispitnog paketa i sve navedene datotečne resurse potrebne za izvođenje ispitivanja.

Jednom pripremljen paket se predaje sustavu koji dalje sam izvršava ispitivanje na korisnikov znak, te je time postignut automatizam ispitivanja. Podjela i grupacija ispitnih slučajeva se može direktno preslikati iz dokumenta specifikacije sustava. Potrebno je još izraditi pomoćne programe koji kreiraju računalne procese i generiraju događaje.

### 2.4.2. Sustav za izvođenje ispitivanja i upravljanje resursima

Definirali smo sadržaj datoteke ispitnog paketa, a sada ćemo reći nešto više o samom sustavu za ispitivanje. Nakon predaje ispitnog paketa sustav otpakira datoteku i učitava opis ispitnog paketa. Sustav pohranjuje ispitni paket i ulazi u fazu dohvata i prevođenja aplikacije u izvršni kôd nakon čega pripremljenu aplikaciju sprema za buduću uporabu. Na komandu korisnika sustav distribuira datoteke po mreži i vrši ispitivanje interpretacijom zadanih mu pravila. Na kraju ispitivanja sustav sakuplja nastale datoteke na jedno mjesto i pakira ih s dnevnikom izvođenja u datoteku rezultata ispitivanja spremnu za korisnika.

Kako bi ostvario svoju funkciju sustav mora upravljati datotečnim i računalnim resursima, te izvoditi samo ispitivanje. Osnovna podjela funkcionalnosti sustava prikazana je dijagramom na slici 2.9.




Slika 2.9: Podjela Lusca sustava po funkcijama

Upravljanje resursima možemo podijeliti na upravljanje datotečnim resursima kao što su distribucija i prikupljanje datoteka, pohrana ispitnog paketa i rezultata izvođenja, te upravljanje računalnim resursima kao što su postavljanje mrežnih parametara računala, pokretanje/nadziranje procesa i slično. Kako bi se postiglo samo izvođenje ispitivanja potrebno je kreirati interpreter pravila koji izvodi što je u nekom trenutku ispitivanja potrebno učiniti i prepoznaje zadovoljenje ispitnog slučaja. Za ostvarenje samog ispitivanja u zadanoj računalnoj mreži potrebno je definirati i implementirati prilagođene mrežne protokole koji će omogućiti prenošenje informacija o kreiranim signalima i akcijama koje treba poduzeti na nekom od računala u mreži.

# 3. Tehnologije korištene u izradi sustava Lusca

Lusca sustav je razvijan da bude aplikacija otvorenog kôda za Linux operacijski sustav, te kao takav i sam je građen uporabom tehnologija otvorenog kôda. Sustav je implementiran pomoću programskog jezika Python [9], te se temelji dobrim dijelom na programskoj biblioteci Twisted [11]. Jezik za opis ispitnih resursa i pravila ispitnih slučajeva je izveden u XML jeziku s odgovarajućom XML shemom. Za komunikaciju između elemenata sustava korištena je SOAP tehnologija [10] za pozive udaljenih procedura, a podatci koje sustav predaje korisničkoj aplikaciji su u JSON formatu [12]. Sustav podržava Subversion protokol [13] za upravljanje inačicama izvornog kôda. Prevođenje izvornog kôda u izvršni je se provodi uporabom GNU Autotools skupa alata [14]. U narednim potpoglavljima je dan kratki uvid u pojedinu tehnologiju na koju se Lusca sustav oslanja.

## 3.1. Programski jezik Python

Python je objektno orijentirani dinamički programski jezik koji se može uporabiti u raznim područjima razvoja računalnih programa. Pruža snažnu podršku za integraciju s ostalim programskim jezicima i alatima te dolazi s popriličnom količinom standardnih biblioteka. Možda najveća značajka programskog jezika Python za njegove korisnike je vrlo povoljna krivulja učenja te se korisni programi mogu pisati već nakon kratkog perioda učenja. Mnogi Python programeri izvješćuju o značajnom porastu produktivnosti i smatraju da ih sam jezik potiče na razvoj kvalitetnijih programa s lako održivim programskim kôdom.

Python je programski jezik neovisan o platformi i pokreće aplikacije na Windows, Linux/Unix, Mac OS X, Amiga računailma, te Palm dlanovnicima i Nokia mobilnim uređajima. Python je također prilagođen za izvođenje u virtualnim mašinama Java i .Net programskih jezika. Distribuiran je pod OSI-odobrenom licencom otvorenog kôda koja ga čini slobodnim za uporabu, čak i za komercijalne proizvode.

Neki od najvećih projekata koji koriste Python su Zope aplikacijski server, Mnet-ova distribuirana pohrana datoteka, Youtube portal i originalni BitTorrent klijent. Neke od velikih organizacija koje koriste Python su Google, NASA i Industrial Light and Magic. Zbog prirode njegovog izvoditelja uspješno je ugrađen u velik broj programskih proizvoda kao skriptni jezik u tim proizvodima. Zbog njegove jednostavnosti često se koristi kao u programima za 3D animaciju kao što su Maya, Softimage XSI i Blender.

### 3.1.1. Sintaksa i semantika

Sintaksa jezika Python je dizajnirana da bi bio vrlo čitljiv programski jezik. Pri tome teži preglednosti kôda koristeći često riječi iz engleskog govornog jezika gdje ostali jezici koriste specijalne znakove. Koristi uvlačenje odsječaka programskog kôda razmacima kako bi se označila pripadnost blokova određenoj rutini umjesto vitičastih zagrada ili ključnih riječi kao što je slučaj s većinom programskih jezika. Povećanje broja razmaka nakon određenog izraza označava pripadnost bloka tom izrazu, dok smanjenje broja razmaka označava kraj bloka.

Neki od ključnih izraza jezika su:

- *if* – uvjetno izvršavanje pripadajućeg bloka kôda, zajedno s *else* i *elif* (skraćeno od *else-if*) ključnim riječima

- while – izvršavanje pripadajućeg bloka kôda dok uvjet ne postane logička laž
- *for* – iteriranje pripadajućeg bloka kôda s predanim iterirajućim objektom, dohvat elemenata se vrši pri svakom prolasku kroz naredbu i sprema se u lokalnu varijablu
- *class* – ključna riječ za definiciju razreda u pripadajućem bloku kôda
- *def* – ključna riječ za definiranje procedure u pripadajućem bloku kôda

Python koristi zakašnjelo određivanje tipa podataka (takozvani eng. *duck-typing* ili eng. *latent-typing*). Provjera ograničenja operacija nad tipovima podataka se ne provodi tijekom prevođenja izvornog kôda u među kôd, već se generira pogreška kao posljedica izvođenja operacije nad neodgovarajućim tipovima. Iako Python ne sprovodi statičko provjeravanje tipa podataka ipak spada u skupinu programskih jezika s čvrsto određenim tipovima podataka jer zabranjuje operacije s različitim tipovima podataka (na primjer operacija zbrajanja broja sa znakovnim tipom) umjesto da ih sam pokuša *shvatiti* i na neki način sprovesti.

Python raspolaže s ugrađenim tipovima podataka prikazanim u tablici 3.1.

| Naziv tipa | Kratki opis | Primjer |
|---|---|---|
| **str, unicode** | Postojani niz znakova. | `'Lusca', u'Lusca'` |
| **list** | Promjenjiva lista za pohranu objekata. Može sadržavati miješane tipove podataka. | `[1,'Lusca',[ 2,3 ]]` |
| **tuple** | Postojana lista za pohranu objekata. Može sadržavati miješane tipove podataka. | `(1,'Lusca',[ 2,3 ])` |
| **set, frozenset** | Neporedana lista sačinjena od jedinstvenih elemenata. | `set([1,'Lusca',5])` |
| **dict** | Asocijativna lista s elementima tipa ključ-vrijednost. | `{'kljuc' : 'vrijednost'}` |
| **int** | Cijelobrojni tip podatka. | `42` |
| **float** | Decimalni tip podatka. | `3.14` |
| **complex** | Tip podataka za kompleksne brojeve. | `2 + 3j` |
| **bool** | Logički tip podataka. | `True` ili `False` |

*Tablica 3.1: Ugrađeni tipovi podataka*

Python dozvoljava definiranje vlastitih tipova podataka. To se postiže kreiranjem korisničkih razreda što je često korištena metoda u objektno orijentiranom programiranju. Nove instance razreda se kreiraju pozivom imena razreda (na primjer *MojRazred()*) te se razredi smatraju instancom istog tipa razreda čime je omogućeno meta-programiranje i refleksija.

Metode razreda su funkcije pozvane s instancom razreda, pa je time sintaksa poziva metode u obliku *instanca.imeMetode(argument)* (uobičajeni način poziva metode) u biti pojednostavljeni oblik poziva tipa *ImeRazreda.imeMetode(instanca, argument)*. Zbog toga definicija metode u Python-u ima eksplicitno naveden parametar *self* koji predstavlja instancu razreda nad kojim je metoda pozvana pri izvođenju, za razliku od nekih drugih objektno orijentiranih programskih jezika (na primjer C++ i Java).

Primjer uporabe razreda je prikazan u idućem odsječku kôda:

```
# razred Jabuka
class Jabuka:
        # konstruktor
        def __init__(self, boja):
                self.mojaBoja = boja
        # metoda za pretvorbu instance u str tip
        def __str__(self):
                return 'Jabuka ' + self.mojaBoja

# razred Kosara
class Kosara:
        # konstruktor
        def __init__(self):
                self.mojSadrzaj = []
        # metoda za dodavanje instanci u kosaru
        def ubaci(self, o):
                self.mojSadrzaj.append(o)
        # ispis sadrzaja kosare
        def ispisiSadrzaj(self):
                print 'sadrzaj kosarice:'
                for element in self.mojSadrzaj:
                        print '\t', element

# glavni program
kosara = Kosara()

jednaJabuka = Jabuka('crvena')
kosara.ubaci(jednaJabuka)

kosara.ispisiSadrzaj()
# kraj programa
```

Program prikazuje jednostavnu uporabu dvaju kreiranih razreda – *Jabuka* i *Kosara*. Svojstvo instanci iz razreda *Jabuka* je boja jabuke spremljena u lokalnoj varijabli *mojaBoja*. Svojstvo instanci razreda *Kosara* je da mogu spremati druge objekte u sebe koristeći proširivu listu *mojSadrzaj*. Primjećujemo dvije posebne metode u razredu Jabuka: *__init__* i *__str__*. Prva je ugrađeni tip metode za definiranje konstruktora razreda, dok druga služi za pretvaranje instance razreda u *str* tip podataka (znakovni tip podataka). Metoda *ispisiSadrzaj* razreda *Kosara* ispisuje sadržaj košarice na način da iterira jedan po jedan element iz liste *mojSadrzaj*. Trenutni element liste se direktno ispisuje na konzolu. Kako razred *Jabuka* posjeduje metodu *__str__* ispis ovog programa je sljedeći:

```
$ python jabuke.py
sadrzaj kosarice:
        Jabuka crvena
$
```

Prikazani program nije koristio niti jednu biblioteku za svoj rad. Korištena su samo svojstva programskog jezika i ugrađeni tipovi podataka. U idućem poglavlju ćemo razmotriti neke dodatne biblioteke s kojima se Python distribuira.

### 3.1.2. Standardne Python biblioteke

Python ima velik broj standardnih biblioteka što se često smatra jednom od njegovih najvećih prednosti, pružajući sredstva primjerena za rješavanje mnogih različitih zadataka. To je rezultat stava voditelja Python programskog jezika po pitanju funkcionalnosti – *baterije uključene*. Moduli standardnih biblioteka se mogu proširivati novim modulima napisanim u programskom jeziku C ili samom Python-u. Zbog velike raznolikosti modula standardne biblioteke spojenom s mogućnosti Python-a da koristi niže jezike (C ili C++) za dodavanje novih funkcionalnosti, Python može biti vrlo moćna spona između raznih programskih jezika i alata.

Standardna biblioteka je posebno dobro dorađena za programiranje Internet aplikacija zbog velikog broja podržanih standarda i protokola (kao što su primjerice MIME i HTTP). Moduli za kreiranje korisnički orijentiranog grafičkog sučelja, spajanje na relacijske baze, aritmetiku s proizvoljnom preciznosti i manipulacija regularnim izrazima su također uključeni. U standardnim bibliotekama Python tako primjerice ima biblioteku za automatsko ispitivanje segmenata programskog kôda u Python-u (PyUnit biblioteka) pružajući time sredstva za iscrpno ispitivanje Python aplikacija.

Za uporabu neke od programskih biblioteka koristi se ključna riječ *import* sa značenjem da su biblioteke i njihovi moduli dostupni za korištenje u programu. Primjer uporabe je prikazan u idućem odsječku kôda.

```
import datetime
print 'Danas je:', datetime.date.today()

import md5
print 'md5(Lusca):', md5.new('Lusca').hexdigest()

import urllib
gf = urllib.urlopen('http://www.google.com')
google_html = gf.read()
gf.close()
print 'www.google.com:', google_html

import re
nadjeno = re.findall('.<img.', google_html)
print 'Google ima slika:', len(nadjeno)
```

Prikazana je uporaba *datetime* modula za dohvat vremena, *md5* modula za izračun MD5 sažetka teksta, *urllib* biblioteke za dohvat sadržaja putem HTTP protokola i *re* modul za regularne izraze.

Neki dijelovi standardne biblioteke su pokriveni specifikacijama, no većina modula je definirana samo dokumentiranim kôdom i ispitnim slučajevima. S druge strane, pošto je većina modula standardne biblioteke pisana u jeziku Python, tako je i standardna biblioteka prenosiva na razne platforme i operacijske sustave.

Osim standardnih biblioteka postoji mnoštvo biblioteka za razne svrhe. Za potrebe ovog diplomskog rada trebalo je pronaći biblioteku koja bi se na što jednostavniji način borila s problemima upravljanja istovremenim izvršavanjem skupa zadataka i mrežnim protokolima.

## 3.2. SOAP protokol

SOAP (eng. *Simple Object Access Protocol*) je protokol za razmjenu XML poruka preko računalne mreže, uobičajno putem HTTP/HTTPS protokola. SOAP kreira podlogu za kreiranje Internet servisa pružajući osnovni okvir za apstrakciju prenošenja podataka s ciljem kreiranja prilagođenih informacijskih usluga.

Postoje razne primjene za razmjenu poruka putem SOAP-a, ali najčešće se koristi kao protokol za poziv udaljenih procedura u kojem jedna strana (klijent) šalje poruku zahtjeva drugoj strani (poslužitelj). Poslužitelj obrađuje zahtjev i vraća poruku odgovora. SOAP je direktni nasljednik XML-RPC-a (eng. *XML Remote Procedure Call*) mada koristi format omotnica/zaglavlje/tijelo za postizanje neutralnosti iz nekih drugih protokola, vjerojatno WDDX-a (eng. *Web Distributed Data eXchange*).

Transporter SOAP poruka može biti SMTP i HTTP, ali HTTP je češće rabljen jer dobro funkcionira na postojećoj infrastrukturi, posebno uzimajući u obzir mrežne obrambene stijene. U kombinaciji s HTTPS protokolom SOAP komunikacija se može voditi sigurno pri čemu je moguća autentifikacije obje strane. To je ujedno i velika prednost SOAP-a prema ostalim distribuiranim protokolima kao što su GIOP/IIOP ili DCOM koji često bivaju filtrirani u mrežnom prometu. XML je odabran kao standardni format poruka zbog njegove raširenosti i podržanosti od strane velikih korporacija i dostupnosti alata otvorenog kôda za obradu XML-a. Također, postojanje velikog broja programskih biblioteka za jednostavnu izradu aplikacija u više-manje bilo kojem programskom jeziku, uvelike pomaže prelasku na SOAP-orijentiranu implementaciju aplikacije.

SOAP-u se često zamjera složena sintaksa XML-a za kreiranje poruka, što je ujedno i njegova prednost ponekad. Prednost je utoliko što omogućuje prijenos skoro svakog oblika strukturalno složenog podatka, a nedostatak je što takve poruke često znaju imati veliku dodatnu količinu podataka kako bi se zadovoljio format poruke, te je s time otežana i obrada poruka. Danas postoje specijalizirana sklopovska rješenja koja se bave ubrzanjem obrade SOAP poruka pa je taj problem u većini slučajeva riješen.

Neka od bitnih pravila sintakse SOAP poruka su:

- SOAP poruka MORA biti u XML formatu
- poruka MORA koristiti SOAP *Envelope* prostor imena
- poruka MORA koristiti SOAP *Encoding* prostor imena
- poruka NE SMIJE sadržavati DTD referencu
- poruka NE SMIJE sadržavati instrukcije za obradu XML podataka.

U idućem odsječku XML-a je prikazan kostur oblika SOAP poruke.

```
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

<soap:Header>
  ...
  ...
</soap:Header>

<soap:Body>
  ...
  ...
  <soap:Fault>
    ...
    ...
  </soap:Fault>
</soap:Body>

</soap:Envelope>
```

SOAP poruka sa sastoji od tri osnovna elementa:

- *Envelope* element – omotnica cijele poruke
- *Header* element – zaglavlje poruke
- *Body* element – tijelo poruke

Omotnica poruke je glavni element XML dokumenta i definira dokument kao SOAP poruku. Sadrži atribute za shemu dokumenta i definira prostor imena korištenih u dokumentu te stil kôdiranja poruke. Opcionalni dio je zaglavlje dokumenta u *Header* elementu sadrži podatke bitne za primjenu poruke (na primjer podatke o autentičnosti i slično). Ukoliko je element prisutan mora biti prvi podelement *Envelope* elementa. U zaglavlju je moguće definirati kome je poruka namijenjena – krajnjoj točki ili nekom poslužitelju na putu poruke. Tijelo poruke sadrži same podatke SOAP poruke namijenjene krajnjim sudionicima razmjene. Podelementi *Body* elementa mogu biti definirani dodatnim prostorom imena. SOAP prostor imena definira samo *Fault* podelement koji se koristi za definiranje pogreške.

U idućem odsječku je prikazana jedna poruka SOAP zahtijeva.

```
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

<soap:Body>
   <m:GetPrice xmlns:m="http://www.s.com/prices">
      <m:Item>Apples</m:Item>
   </m:GetPrice>
</soap:Body>

</soap:Envelope>
```

Ovaj poruka opisuje zahtjev za cijenom jabuka. Element *GetPrice* i *Item* nisu dio uobičajenog SOAP prostora imena, već korisnički definiranog. Poruka odgovora bi moglo izgledati na sljedeći način:

```
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

<soap:Body>
   <m:GetPriceResponse xmlns:m="http://www.s.com/prices">
      <m:Price>1.90</m:Price>
   </m:GetPriceResponse>
</soap:Body>

</soap:Envelope>
```

Poruka odgovora sadrži podatke o traženoj cijeni.

Kao što je već rečeno, jedan od razloga za raširenost SOAP protokola u praksi je i postojanje velikog broja implementacija alata i programskih biblioteka za izradu SOAP aplikacija. Jedna od takvih biblioteka za Python je Twisted programska biblioteka.

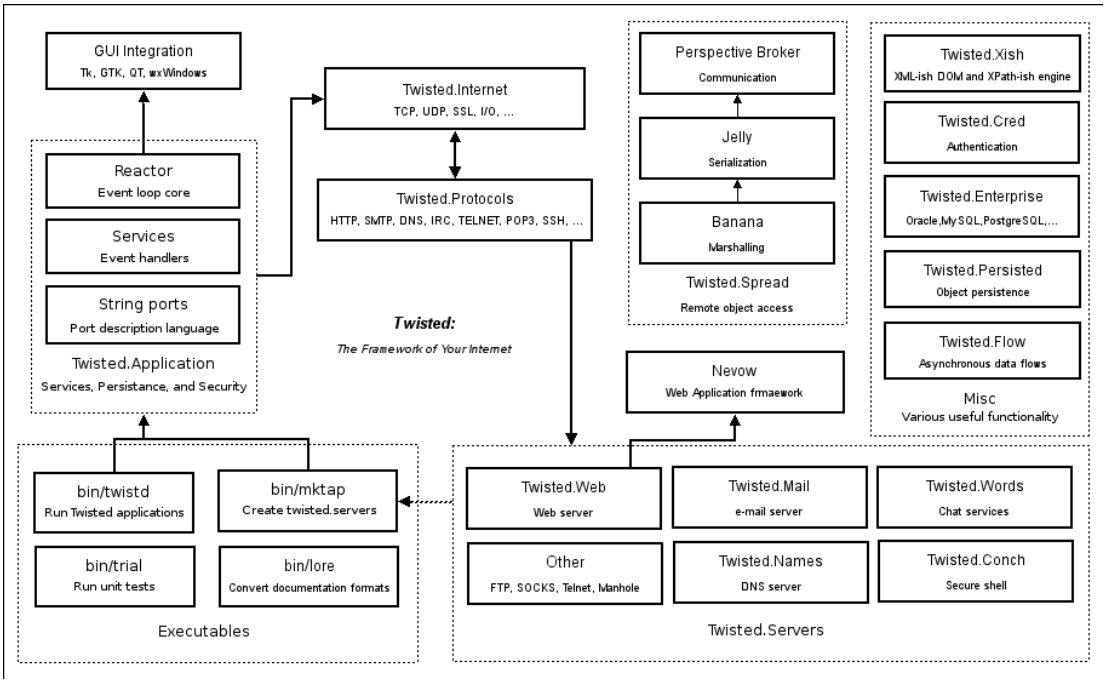
## 3.3. Python programska biblioteka Twisted

Twisted je Python programska biblioteka za izradu mrežnih aplikacija upravljanim događajima i izvorni kôd je distribuiran pod MIT licencom. Njegovi autori kažu da bi Twisted prije svega trebao biti zabavan. Počeo je kao igra, danas se koristi komercijalno u računalnim igrama i rad s njim za krajnjeg korisnika bi trebao pružiti interaktivno i zabavno iskustvo. Twisted je platforma za razvoj Internet aplikacija. Dok je Python, sam za sebe, vrlo moćan jezik, postoje mnoga područja koja nisu dobro pokrivena kao što je to slučaj kod nekih starijih jezika čija se funkcionalnost bila nadopunjavala kroz vrijeme velikim brojem programskih biblioteka. Twisted pokušava popuniti neka područja za Python.

Moduli koje pruža Twisted možemo podijeliti po funkcionalnosti u dvije osnovne skupine:

- moduli za asinkrono programiranje
- moduli za programiranje Internet aplikacija

Moduli za programiranje Internet aplikacija se temelje na modulima za asinkrono programiranje. Twisted pruža sredstva za asinkrono programiranje koja omogućavaju neblokirajuće pozive odgađanjem izvršavanja bloka naredbi do trenutka kada su spremni očekivani podatci. Na tim temeljima su izgrađeni moduli za programiranje Internet aplikacija s podrškom za mnoge protokole. Prikaz funkcionalne podjele Twisted biblioteke je prikazan na slici 3.1.




Slika 3.1: Dijelovi Twisted biblioteke

U idućim potpoglavljima je detaljnije razmatrana uporaba Twisted biblioteke u područjima potrebnim za razvoj Lusca sustava.

### 3.3.1. Programiranje odgođenih akcija s Twisted bibliotekom

Osnovna značajka programiranja mrežne aplikacije je čekanje na podatke. Uzmemo li na primjer proceduru za slanje elektroničke pošte koja šalje neke prikupljene podatke. Procedura se mora spojiti na poslužitelj, čekati na odgovor, provjeriti da li udaljeni poslužitelj može obraditi poruku, čekati na odgovor, poslati poruku, čekati potvrdu i odspojiti se. Svaki od ovih koraka može trajati poprilično dugo. Najjednostavnija aplikacija bi ovaj scenarij mogla imati implementiran na način da ne radi ništa drugo dok se podatci šalju ili dok se čeka na primitak, ali u tom slučaju aplikacija ne može recimo slati više elektroničkih poruka istovremeno niti raditi išta drugo u biti. Ovaj model se izbjegava osim u baš najjednostavnijim mrežnim aplikacijama.

Postoje mnogi modeli za pisanje efikasnijih mrežnih aplikacija. Bitni među njima su:

1. obrađivati svaku vezu u odvojenom procesu operacijskog sustava i prepustiti brigu operacijskom sustavu o istovremenosti izvođenja
2. obrađivati vezu u odvojenim dretvama i mehanizmima za vođenje dretvi propuštati rad one dretve koja je spremna

3. koristiti ne-blokirajuće sistemske pozive za obradu svih veza u jednoj dretvi.

Twisted koristi treći model kako bi ostvario potrebnu funkcionalnost. Kada treba baratati s više veza istovremeno u istoj dretvi, odgovornost je aplikacije ili programske biblioteke da planira dodjelu prava izvođenja. Ta funkcionalnost se uobičajeno postiže pozivima registriranih procedura kada je veza spremna za čitanje ili pisanje – često zvano asinkronim programiranjem ili programiranje povratnim pozivima (eng. *callback-based programming*).

U ovom modelu prijašnja situacija s procedurom za slanje elektroničke pošte bi izgledala na idući način:

1. poziva se procedura za spajanje na udaljeni poslužitelj
2. procedura završava odmah, no bilježi se da treba pokrenuti definiranu proceduru kad se veza uspostavi
3. kad se veza uspostavi, mehanizam za upravljanje akcijama pokreće proceduru za slanje elektroničke poruke

Ovakav scenarij ima prednost pred prethodnim zbog toga što se ostatak programa može odvijati nesmetano dok nije uspostavljena veza s udaljenim poslužiteljem.

Tipičan način dojave o spremnosti nekih podataka u asinkronom modelu je metoda povratnog poziva. Aplikacija zove funkciju koja zahtijeva neke podatke, te u istom pozivu predaje kao argument povratnu funkciju koja će se pozvati kada podatci budu spremni. Povratna funkcija obrađuje podatke. Dakle, poziva se funkcija koja zatraži podatke, a mehanizam za povratni poziv se brine da funkcija bude pozvana kada su podatci spremni.

Twisted koristi poseban objekt za kontrolu povratnih poziva – *Deferred*. Klijent aplikacija dodaje seriju funkcija odgođenom objektu da budu pozvani u ispravnom redoslijedu kada su rezultati asinkronog zahtjeva spremni. Serija takvih funkcija se naziva *povratni lanac* (eng. c*allback chain*). Dodatno se definira i serija funkcija koja će se pozvati ukoliko dođe do pogreške prilikom dohvata rezultata (eng. *errback chain*). Biblioteka za asinkrono upravljanje nakon kraja funkcije za dohvat podataka poziva prvu funkciju *callback* lanca ukoliko su podatci spremni ili prvu funkciju *errback* lanca ukoliko je došlo do greške, a *Deferred* objekt se brine za predaju ispravnih podataka svakoj od poziva *errback* ili *callback* funkcija u pripadajućem lancu. U idućem primjeru prikazana je uporaba Twisted biblioteke za upravljanje asinkronim operacijama.

```
from twisted.web.client import getPage
from twisted.internet import reactor

def obradaGreske(greska):
    print "Dogodila se greska: <%s>" % str(greska)
    reactor.stop()

def ispisSadrzaja(podatci):
    print podatci
    reactor.stop()

# zahtjev za stranicom koja ne postoji
deferred = getPage('http://twistedmatrix.com/does-not-exist')
# kreiranje povratnog lanca
deferred.addCallback(ispisSadrzaja)
# dodavanje funkcije za obradu pogreske
deferred.addErrback(obradaGreske)

reactor.run()
```

Prikazani primjer se sastoji od dvije funkcije: jedna za obradu podataka – *ispisSadrzaja*, te druga za obradu greške – *obradaGreske*. U glavnom programu kreira se *Deferred* objekt pozivom funkcije za dohvat određene HTTP adrese. Odgođenom objektu se kreiraju povratni lanci dodavanjem funkcija za obradu podataka i greške pozivom odgovarajućih metoda objekta – *addCallback* i *addErrback*. U posljednjoj liniji se pokreće mehanizam za upravljanje odgođenim akcijama – *reactor.run()* pozivom.

U ovom primjeru možemo uočiti jednostavnost programiranja odgođene obrade podataka pomoću Twisted biblioteke. Korištena je jezgra Twisted biblioteke, *reactor* modul, te funkcija za dohvat sadržaja s HTTP adrese iz *web.client* modula.

### 3.3.2. Izrada SOAP servisa pomoću Twisted biblioteke

Velika moć Twisted biblioteke je u jednostavnom kreiranju prilagođenih protokola koji se zasnivaju na postojećim tehnologijama. Twisted tako omogućava jednostavnu izradu raznih TCP poslužitelja i klijenata kao i razvijanje aplikacije zasnovane na XML-RPC servisima, Perspective-Broker protokolu ili SOAP-u. Lusca koristi SOAP protokol za prenošenje poruka između elemenata sustava. Lusca SOAP servisi su kreirani pomoću Twisted *SOAPPublisher* razreda. Primjer izrade SOAP servisa je dan idućim isječkom kôda.

```
from twisted.web import server, soap
from twisted.internet import reactor

class KosaraSOAP(soap.SOAPPublisher):
    def __init__(self):
        self.mojSadrzaj=[]

    def soap_ubaci(self, ime ):
                self.mojSadrzaj.append( ime )
                return 'ubaceno u kosaru'

    def soap_sadrzaj(self):
        rezultat = 'sadrzaj kosarice: \n'
        for element in self.mojSadrzaj:
                rezultat += '\t' + element + '\n'
        return rezultat

siteRoot = KosaraSOAP()
reactor.listenTCP(7777, server.Site(siteRoot))
reactor.run()
```

Ovdje je prikazana izrada razreda *KosaraSOAP* koji će poslužiti kao SOAP servis. Kako bi se razred mogao ponašati kao SOAP servis potrebno je naslijediti Twisted-ov *SOAPPublisher* razred. SOAP metode se definiraju s prefixom *soap_* u imenu metode. Tako imamo dvije SOAP metode – *soap_ubaci* i *soap_sadrzaj*. Prva se koristi za dodavanje elemenata u košaru, druga za ispis sadržaja. SAOP klijent je prikazan idućim odsječkom kôda.

```
from twisted.web import soap
from twisted.internet import reactor

kosara = soap.Proxy('http://localhost:7777')
poziv = kosara.callRemote

def dodajUKosaru():
        poziv('ubaci', 'Jabuka crvena').addCallback(ispisPod)
        poziv('ubaci', 'Jabuka zelena').addCallback(ispisPod)
        poziv('ubaci', 'Jabuka zuta').addCallback(ispisPod)

def ispisKosare():
        poziv('sadrzaj').addCallback(ispisPod)

def ispisPod(podatak):
        print 'Primljeno:\n', podatak

# glavni program
dodajUKosaru()

reactor.callLater(1, ispisKosare)
reactor.callLater(2, reactor.stop)

reactor.run()
```

Klijent dodaje 3 jabuke u udaljenu košaricu. Razred za komunikaciju s udaljenim servisom je *Proxy* razred iz Twisted modula *soap*. Pozivom metode *callRemote* nad instancom razreda, započinje poziv udaljene SOAP metode. Metoda callRemote vreća objekt razreda *Deferred* kojemu kreiramo povratni lanac funkcijom *ispisiPod* za ispis povratne poruke. U glavnom dijelu programa pozivamo funkciju *dodajUKosaru* koja dodaje istovremeno 3 jabuke u udaljenu košaru. S jednom sekundom zakašnjenja pozivamo funkciju *ispisKosare* koja zove udaljenu metodu za dohvat sadržaja košare. U idućem odsječku je dan primjer izvođenja klijent programa.

```
$ python soap_kosara_klijent.py
Primljeno:
ubaceno u kosaru
Primljeno:
ubaceno u kosaru
Primljeno:
ubaceno u kosaru
Primljeno:
sadrzaj kosarice:
        Jabuka zuta
        Jabuka zelena
        Jabuka crvena
$
```

Dogodila su se tri ispisa povratne poruke nakon ubacivanja jabuka u košaru, te nakon jedne sekunde pauze dogodio se ispis sadržaja košare.

## 3.4. JSON format podataka

JSON (eng. *JavaScript Object Notation*) je *laki* tekstualni format podataka za opis strukturiranih podataka. Čitljiv je i jednostavan za pisanje. Jednostavan je za strojno generiranje i obradu. Temelji se na podskupu pravila sintakse JavaScript programskog jezika i potpuno je neovisan o platformama na kojima se koristi. Po svojoj strukturi je sličan familiji C jezika uključujući C, C++, C#, Javu i mnoge druge. Radi jednostavnog formata, moči pri opisivanju podataka i sličnosti s raširenim programskim jezicima postaje idealan za razmjenu podataka.

JSON je izgrađen na dvije osnovne strukture:

- lista s elementima ključ-vrijednost – u raznim jezicima ostvaruje se kao objekt, zapis, struktura ili asocijativno polje
- poredana lista elemenata – često implementiramo u programskim jezicima kao lista, vektor ili sekvenca elemenata.

Ovo su univerzalne podatkovne strukture. Gotovo svi moderni programski jezici imaju podršku za ovakve strukture u nekom obliku. JSON se zasniva na ideji da bi podatci trebali biti razmijenjivi između raznih programskih jezika koristeći upravo ove strukture.

Za definiranje ovih struktura koriste se posebni znakovi za odijeljivanje: { } [ ], dvotočka, zarez i točka-zarez. Različitom uporabom struktura kreira se skoro bilo koji strukturirani podatak. Primjer je dan idućim odsječkom kôda.

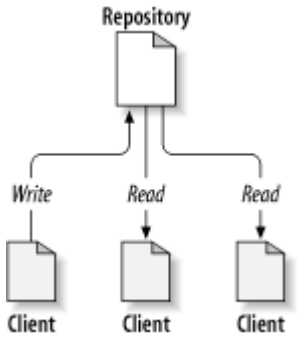
```
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}}
```

Prikazani primjer opisuje element korisničkog sučelja i definira objekt menu. Objekt sadrži jednostavne atribute *id* i *value*, te atribut *popup* čija je vrijednost složeni objekt *menuitem*. On se sastoji od poredane liste čije su vrijednosti elementi tipa lista ključ-vrijednost.

Ovakav zapis se pokazao kao dovoljan za opis vrlo složenih struktura, te je jako jednostavan za obradu. JSON se koristi u Lusca sustavu za prijenos često zatraženih podataka kako bi se izbjeglo veće opterećenje sustava i omogučilo jednostavan dohvat podataka u stanju sustava.

## 3.5. Subversion sustav

Subversion je centralizirani sustav za dijeljenje informacija. U jezgri sustava se nalazi repozitorij kao centralno mjesto pohrane podataka. Repozitorij pohranjuje informacije o stablu datoteka – uobičajeno hijerarhijski skup datoteka i direktorija. Klijenti se spajaju na repozitorij kako bi čitali i pisali po datotekama. Čim klijent zapiše svoje podatke oni su dostupni svima ostalima za čitanje. Sustav se stoga zasniva na poslužitelj-klijent arhitekturi, a princip rada je prikazan slikom 3.2.




Slika 3.2: Prikaz rada s repozitorijem datoteka

Kako sustavu pristupa više korisnika istovremeno potrebno je osigurati ispravan rad ukoliko više korisnika želi recimo pisati po istoj datoteci. Subversion sustav to omogućuje pomoću mehanizama zaključavanja datoteka dok pojedina operacija ne završi. Dodatno svojstvo sustava je da vodi dnevnik o verzijama svake od datoteka tako da je moguće dobiti bilo koju verziju određene datoteke nakon svake izmjene u bilo kojem trenutku.

Za rad sa sustavom korisnici instaliraju korisnički alat koji koji podržava komunikacijski protokol poslužitelja i brine se za lokalnu verziju datoteka na korisnikovom računalu. Rad s datotekama na poslužitelju započinje s prvim dohvatom trenutnog stanja datoteka na

repozitoriju (eng. *checkout*). Nakon izmjena lokalne kopije datoteka, korisnik izdaje komandu za pohranu promjena na poslužitelj (eng. *commit*). Kada korisnik želi nastaviti s daljnjim radom na datotekama mora preuzeti promjene nastale za vrijeme dok on nije radio (eng. *update*) s poslužitelja. Korisnik nastavlja raditi s datotekama i time je ciklus zatvoren.

Ovo je samo kratki uvod u rad sa Subversion sustavom za pohranu datoteka, stoga nisu razmatrane mnogi problemi koji nastaju tijekom rada niti opisana rješenja koja sustav pruža pošto to i nije bitno za ovu diplomsku radnju koja koristi samo mali podskup pruženih funkcionalnosti.

Potrebno je napomenuti da se sustav uglavnom koristi kao repozitorij kôda za vrijeme razvoja aplikacije pošto razvojni tim često nije na istom mjestu. Sustav rješava moguće konflikte zbog različitih verzija izvornog kôda dok članovi tima rade na istoj datoteci. Lusca koristi Subversion sustav za dohvat izvornog kôda razvijane aplikacije pomoću komandnog korisničkog alata *svn* (eng. *Subversion command line client tool*).

Uporaba komandnog programa je dana idućim formatom:

```
svn command [options] [args]
```

pri čemu *command* dio naredbe može biti *checkout*, *commit* ili *update*. Uz checkout parametar potrebno je navesti putanju poslužitelja, pa takav poziv izgleda recimo ovako:

```
$ svn checkout http://svn.posluzitelj.com/aplikacija/
...
$
```

Pokretanjem ove naredbe preuzima se direktorij *aplikacija* s poslužitelja. Program ispisuje poruke o dodavanju novih datoteka s poslužitelja. Nakon izvršenih promjena potrebno je pokrenuti naredbu u istom direktoriju za unošenje promjena na repozitorij na sljedeći način:

```
$ svn commit
Commited revision number 2.
$
```

Naredba za preuzimanje nastalih promjena izvršena u lokalnom direktoriju je sljedeća:

```
$ svn update
...
Updated to revision 3.
$
```

Program ispisuje listu promijenjenih datoteka i završava rad s ispisom broja revizije preuzetih promjena.

## 3.6. Prevođenje GNU Autotools skupom alata

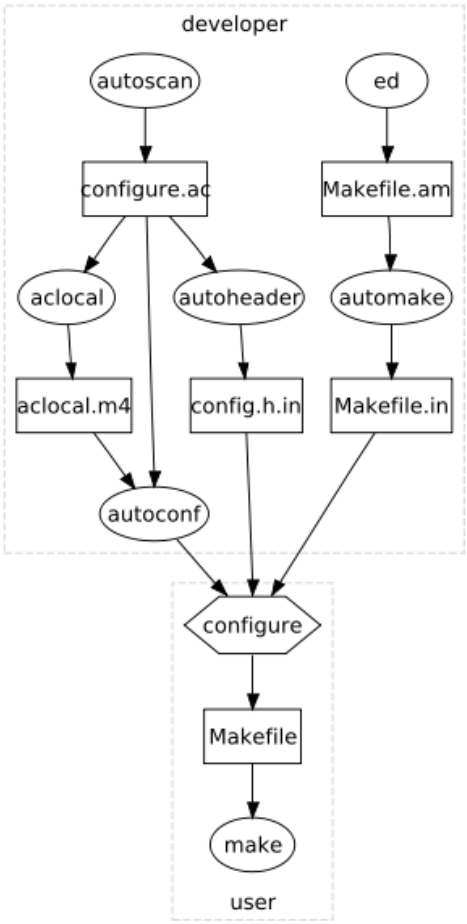
GNU (akronim eng. *Gnu is Not Unix*) sustav za izgradnju ciljnog programa, također poznat kao GNU Autotools, je skup komandno-linijskih alata proizvedenih po standardima GNU projekta. Ovi alati su dizajnirani da pomognu pri prevođenju izvornog kôda raznih programskim jezika u izvršne programe za mnoge operacijske sustave slične Unix-u. GNU sustav za prevođenje aplikacija dio je GNU lanca alata i često se koristi u mnogim projektima otvorenog i slobodnog kôda. Mada se sustav distribuira pod GPL licencom slobodnih programa, ne postoje ograničenja za uporabu sustava u komercijalnim aplikacijama.

Sustav se sastoji od sljedećih alata:

- *autoconf* – obrađuje datoteke *configure.in* ili *configure.ac* kako bi generirao *configure* skriptu. Skripta dalje koristi dodatne datoteke s uobičajenom ekstenzijom *.in* kako bi proizvela krajnju datoteku – *Makefile*. *Autoconf* pomoću raznih pomoćnih alata pokušava zaobići neke od problema na Unix sustavima kreiranjem *configure* skripte koja bi trebala detektirati prisutnost potrebnih alata za prevođenje.
- *automake* – pomaže pri kreiranju *Makefile* koji se koristi za samu izgradnju programa. Kao ulazne datoteke koristi *Makefile.am* i kreira *Makefile.in* kojeg koristi *autoconf* za izradu *Makefile* datoteke.
- *libtool* – pomaže pri kreiranju statičkih i dinamičkih biblioteka. To se postiže apstrakcijom procesa kreiranja biblioteke, sakrivajući razlike između različitih operacijskih sustava (na primjer *Gnu/Linux* i *Solaris* sustava).

Prilikom procesa generiranja ciljanoga programa koriste se i dodatni GNU alati kao što su *make*, GNU *gettext*, *pkg-config* i *gcc* (eng. *GNU Compiler Collection*). Prikaz procesa izgradnje programa dan je na slici 3.3.

Razvijatelj programa (eng. *developer*) koristi GNU alate kako bi kreirao *configure* skriptu koja se pakira zajedno s izvornim kôdom i tako distribuira. Korisnik nakon preuzimanja paketa pokreće skriptu kako bi dobio *Makefile* datoteku. GNU alat *make* koristi prizvedenu datoteku za samu izradu ciljnog izvršnog programa.




Slika 3.3: Dijagram izrade izvršnog programa GNU alatima

Lusca prevodi izvorni kôd kojeg je preuzeo sa Subversion sustava postavljanog kao repozitorij aplikacije pomoću GNU alata na način da navedene komandno-linijske programe pokreće potrebnim redoslijedom kako bi se kreirao izvršni program.

# 4. Lusca sustav

Lusca sustav je započet kao projekt s ciljem kreiranja sustava za automatsko ispitivanje aplikacija sa sredstvima koja posebno pomažu ispitivanje mrežnih aplikacija. Implementiran je u programskom jeziku Python i velikim dijelom se tamelji na programskoj biblioteci Twisted i trenutno se nalazi u ranoj fazi razvoja.

Sustav je zamišljen kao mrežna aplikacija postavljena na računalima koja sačinjavaju ispitnu okolinu. Zadatak sustava je da upravlja ispitnom okolinom kako je definirano ispitnim paketom. Ispitivanje se sprovodi na način da se izvode pravila iz opisa ispitnog slučaja s ciljem detekcije događaja koji potvrđuje ispravnost rada aplikacije ili pogrešku. Sustavu se predaje datoteka koja sadrži datotečne resurse za izvođenje ispitivanja i opis ispitnog paketa. Opis ispitnog paketa je u prethodno definiranom XML jeziku i sadrži listu resursa potrebnih za ispitivanje, te listu ispitnih slučajeva s pravilima za izvođenje. Resursi za ispitivanje su:

- pomoćne skripte i programi
- pomoćne datoteke i konfiguracije
- ispitivane aplikacije
- računalna oprema potrebna za ispitivanje.

U ispitnom slučaju se koriste opisani resursi na odgovarajući načine kako bi se definirao potreban tok izvođenja. Za definiciju toka pravila ispitnog slučaja koristi se programska paradigma upravljanja događajima koja koristi događaje za pokretanje lokalnih akcija i signale za pokretanje akcija na udaljenim računalima u ispitnoj okolini.

Lusca sustav koristi skripte kao glavni izvor informacija o procesima koji se zbivaju u ispitnoj okolini. Skripte se također koriste za samo pokretanje akcija na računalima. Prilikom pokretanja skripte moguće je definirati ulazne parametri skripte kojima se mijenja ponašanje skripte tijekom izvođenja. Vrijednosti parametara skripte mogu biti definirani unaprijed ili tijekom izvođenja samog ispitivanja koristeći varijable. Varijable poprimaju vrijednosti tijekom izvođenja ispitivanja na način da skripta generira posebni znakovni niz kojeg Lusca prepoznaje kao izlaznu vrijednost skripte. Na sličan način se definira i dojava o događaju ili nastaloj datoteci koju je potrebno pohraniti.

Sustav mora pokretati definirane skripte ili pomoćne programe istovremeno na jednom od računala te prenositi signale i datoteke između računala u ispitnoj okolini. Za tu svrhu koristi se metoda asinkronog programiranja definiranjem povratnih lanaca. Za mehanizam upravljanja asinkronim operacijama i kreiranje SOAP servisa koristi se Twisted programska biblioteka radi njene jednostavnosti, potpunosti i dokazane kvalitete.

Rezultati ispitivanja se sakupljaju i pohranjuju za korisnika zajedno s dnevnicima sustava o izvođenju. Dnevnici sustava o ispitivanju su također u XML formatu te mogu poslužiti za jednostavno kreiranje prilagođenih izvještaja ispitivanja aplikacija u dokumentacijske svrhe projekta.

U idućim potpoglavljima ćemo razmotriti XML jezik za opis ispitnog paketa, definirati arhitekturu sustava i pojasniti njegove elemente, definirati komunikacijske protokole korištene za razmjenu informacija, te objasniti rad sustava kroz faze procesa ispitivanja. Rad sa sustavom je olakšan postojanjem korisničkih programa s grafičkim sučeljem za

upravljanje i nadzor Lusca sustava, te uređivačkim programom opisa ispitnog paketa koji su izrađeni u sklopu diplomskih radova studenata Dražena Nežića i Hrvoja Slavičeka. Na kraju poglavlja su navedeni načini primjene Lusca sustava za uporabnu prilagodbu s ciljem generiranja projektne dokumentacije.

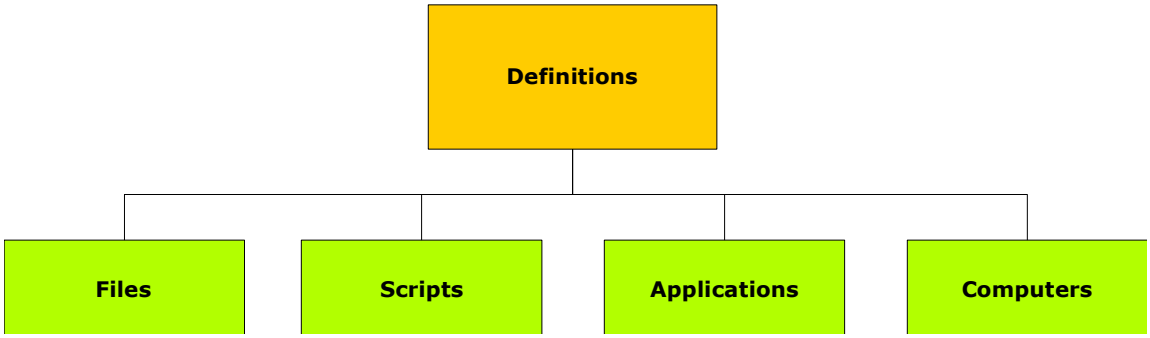
## 4.1. Opis ispitnog paketa u XML jeziku

Kako bi sustav mogao vršiti ispitivanje potrebno mu je predati sve resurse koji se koriste pri ispitivanju, te upute za sustav kako ih koristiti. Za tu svrhu kreiran je podskup XML jezika. Korišteni jezik je detaljnije opisan u seminarskom rada ovog autora [15], te su u ovoj radnji prikazani samo bitni dijelovi jezika potrebni za razumijevanje rada Lusca sustava.

Opis ispitnog paketa sadrži definiciju ispitnog paketa s informacijama o samom ispitnom paketu (autor, datum, identifikator paketa i slično), definiciju resursa i popis ispitnih slučaja s pravilima za izvođenje. U idućim potpoglavljima posljednja dva elementa su detaljnije razrađena.

### 4.1.1. Definicija ispitnih resursa

Kako bi sustav sproveo potrebno ispitivanje potrebno mu je definirati pomoćne datoteke i konfiguracije, pomoćne skripte za kreiranje procesa i dojavu događaja, informacije o ispitivanim aplikacijama, te popis računala potrebnih za ispitivanje s njihovom mrežnom opremom. Pojednostavljeni dijagram ovog elemenata dan je na slici 4.1.




Slika 4.1: Podjela osnovnih elemenata definicije resursa

Ovaj element se naziva *Definitions* i sadrži informacije za opis ispitnih resursa.

Prilikom definiranja pomoćnih datoteka koristi se *Files* podelement. U njemu se definira početna i ciljna putanja datoteke za vrijeme ispitivanja s njenim identifikatorom. Za definiciju skripte definira se njen identifikator, datoteka s početnom i ciljanom putanjom, interpreter skripte i ulazno-izlazni parametri skripte. Za definiranje aplikacije navodi se identifikator aplikacije, kratki opis, podatci o autoru i informacije o repozitoriju na kojem se aplikacija nalazi s konfiguracijskim opcijama korištenim pri prevođenju u izvršni kôd. Za računala se definira njihov identifikator, tekstualni opis računala i lista mrežnih sučelja s pripadajućim parametrima kao što su IP adresa i slično.

Svaki od definiranih resursa se može koristiti u ispitnim slučajevima navođenjem ovdje definiranih identifikatora u pripadajućim vrijednostima atributa. Definirani datotečni resursi se uobičajeno pohranjuju unutar posebnih direktorija na istoj hijerarhijskoj razini s opisom ispitnog paketa. Tako se recimo za skripte koristi direktorij *scripts*, za konfiguracijske

datoteke direktorij *confs*, a za ostale datoteke direktori *resources*. Unutar *confs* direktorija radi se dodatna podjela na direktorije potrebne za određeni ispitni slučaj, pa tako možemo imati poddirektorije *tc01*, *tc02* i *tc03* ukoliko imamo tri ispitna slučaja s navedenim identifikatorima.

Skripte mogu biti pisane u bilo kojem programskom jeziku pa je stoga potrebno definirati interpreter za pojedinu skriptu kako bi Lusca sustav mogao skriptu ispravno pokrenuti. Prilikom pokretanja skripte sustav puni listu ulaznih parametara definiranih u opisu skripte i pomoću njih pokreće skriptu. Tijekom izvođenja skripte ona može ispisivati unaprijed definirane poruke s kojima predaje informacije sustavu. Tako na primjer linija za dojavu o događaju ispisana je na ovaj način:

```
<event>:<0>:<event_id>:<opis dogadjaja>
```

pri čemu prvi element (<*event*>) ukazuje da se radi o liniji koja definira događaj, drugi element (<*0*>) označava broj za korištenje pri analizi izvođenja, treći element (<*event_id*>) je identifikator događaja, a četvrti (<*opis dogadjaja*>) služi za kratki tekstualni opis događaja.

Kako bi skripta predala sustavu vrijednost koju je potrebno pohraniti u varijablu potrebno je ispisati liniju u idućem formatu:

```
<value>:<first_value>:<second_value>:<...>
```

pri čemu prvi element (<*value*>) ukazuje da se radi o liniji koja predaje vrijednosti sustavu, a ostali elementi (<*first_value*>*:...*) su same vrijednosti. Sustav na osnovu definicije izlaznih parametara skripte i definicije za pokretanje skripte dodijeljuje varijablama predane vrijednosti. Ovakvim formatom moguće je predati velik broj izlaznih vrijednosti odjednom.

Kako bi sustav mogao prikupiti potrebne datoteke nastale za vrijeme izvođenja ispitivanja potrebno mu je ukazati na njih. Brigu o tome također vode skripte na način da generiraju izlaznu liniju u idućem formatu:

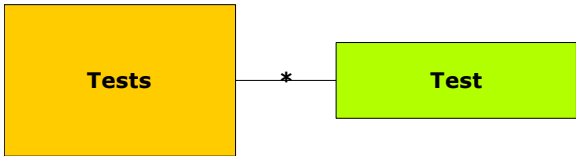
```
<file>:<file_id>:<file_path>
```

pri čemu prvi element (<*file*>) ukazuje da se radi o liniji za prikupljanje datoteke, drugi element (<*file_id*>) predstavlja identifikator datoteke, a trećim elementom (<*file_path*>) se definira relativna putanja nastale datoteke naspram putanje same skripte. Sustav bilježi informaciju o datoteci i sakuplja datoteku u odgovarajućem trenutku.

Na ovaj način skripte postaju bitan dio ispitnog paketa koje sustav koristi za dvosmjernu interakciju s ispitnom okolinom. Skripte se koriste dakle za pokretanje procesa, prikupljanje informacija o događajima na računalu i postavljanje vrijednosti varijabli te za dojavu o nastalim datotekama.

### 4.1.2. Opis toka izvođenja testa

Element koji sadrži popis ispitnih slučajeva je *Tests* element i podijeljen je na ispitne slučajeve sadržane *Test* elementima prikazan na slici 4.2

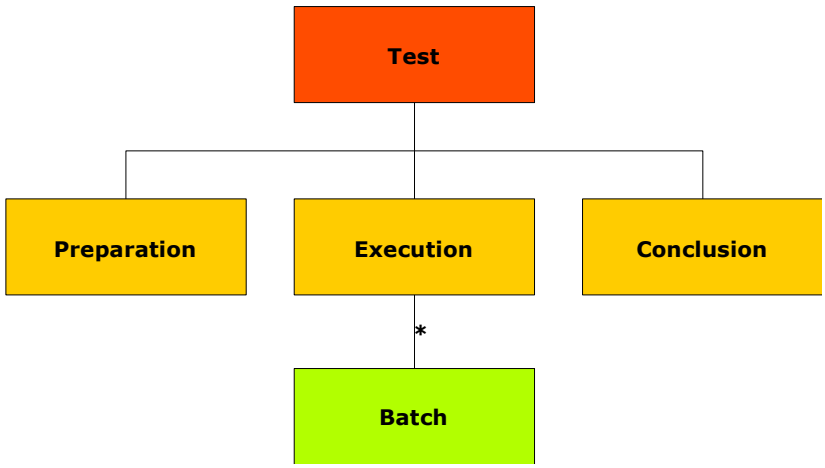



Slika 4.2: Ispitni paket podijeljen na slučajeve

Element za opis ispitnog slučaja se naziva *Test* i sadrži kratki tekstualni opis, identifikator, listu slučajeva o kojima ovisi, te elemente za definiciju triju faza ispitivanja. Moguće je definirati *DependsOn* elementom listu ispitnih slučajeva koji moraju biti ispravni da bi se ovaj slučaj mogao pokrenuti. Time je omogućeno skraćivanje vremena izvođenja ispitnog paketa na način da neke slučajeve ne treba pokrenuti ukoliko nisu zadovoljeni slučajevi o kojima slučaj ovisi. Za definiranje izvođenja ispitnog slučaja koriste se tri faze:

- priprema – pokreću se skripte potrebne za postavljanje preduvjeta u ispitnoj okolini
- ispitivanje – glavni dio ispitivanja s pravilima za izvođenje
- zaključak – pokretanje skripti za čišćenje ispitne okoline

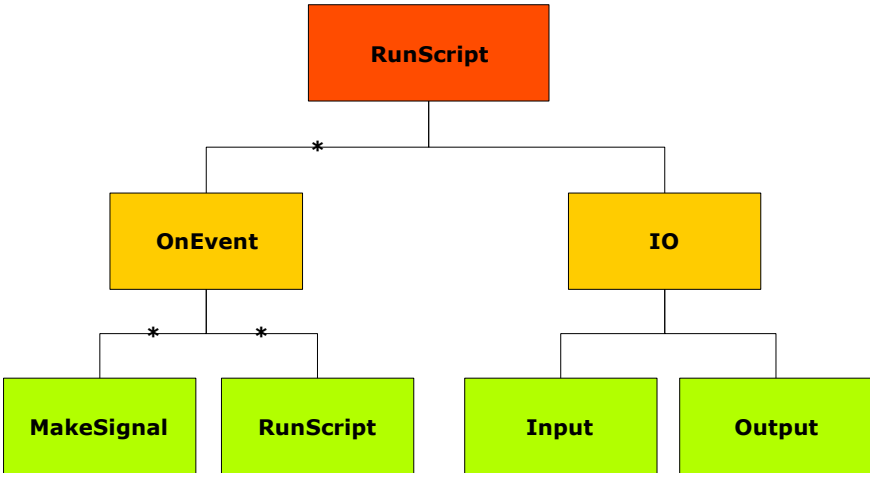
Razlika između pripremne i zaključne naspram glavne faze ispitivanja je u tome što se ispitni slučaj smatra neispravnim jedino ukoliko dođe do greške u glavnoj fazi – fazi ispitivanja, dok se u ostalim fazama greška neke skripte smatra kao nezadovoljenje preduvjeta ispitivanja. Pojednostavljeni dijagram elemenata ispitnog slučaja prikazan je na slici 4.3.




Slika 4.3: Prikaz elemenata triju faza ispitivanja

Element za glavnu fazu ispitivanja je *Execution* i sadrži više podelemenata *Batch* tipa. Potonji element definira zadatak za jedno ili više računala čije izvođenje započinje na primitak definiranog signala, a uzrokuje pokretanje skripte. Time je omogućeno jednostavno definiranje ponašanja na skupu računala što je čest slučaj pri ispitivanju mrežnih aplikacija. Na primjer tako se može jednim elementom reći cijelom skupu računala da pokrenu skriptu za generiranje HTTP zahtjeva prema ispitivanom poslužitelju pri ispitivanju aplikacije za

HTTP poslužitelj. Za *Batch* element se definira njegov identifikator i lista varijabli koje se smatraju lokalnim varijablama na domaćinima ovog zadatka. U *Command* podelementu se nalazi *RunScript* element kojim se opisuje pokretanje skripte. Pojednostavljeni dijagram *RunScript* elementa je prikazan slikom 4.4.




Slika 4.4: Prikaz elementa za pokretanje skripte

Za pokretanje skripte potrebno je definirati akcije pokrenute događajima (*OnEvent* elementom) i ulazno-izlazni parametri skripte. Za pojedini događaj se definira identifikator događaja i lista potrebnih akcija koje mogu biti:

- kreiranje signala – elementom *MakeSignal* se definira identifikator i razina signala kojeg je potrebno kreirati,
- pokretanje skripte – odgovara istom *RunScript* elementu čime su omogućeni ugnježđeni pozivi skripti.

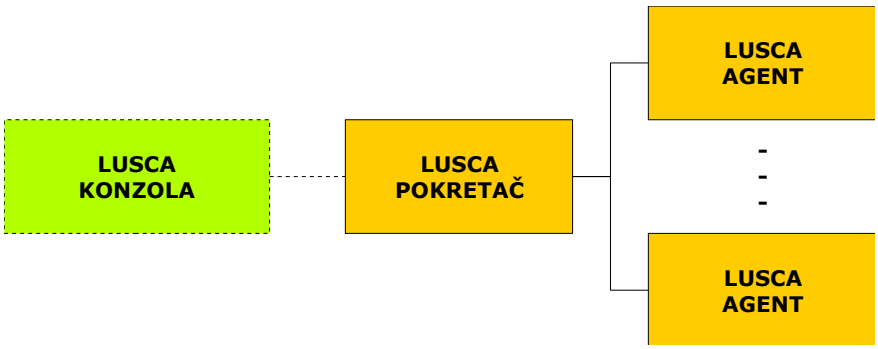
Element *Batch* je glavni građevni element za definiranje pravila ispitivanja. Jedan *Batch* se može izvršavati na više domaćina i pokretan je signalima koji se generiraju kao reakcija na događaje. Pri opisu signala koji se generiraju potrebno je navesti razinu signala koja može biti jedna od:

- *info* – uobičajena razina signala za pokretanje udaljene akcije,
- *error* – razina za dojavu o manje bitnoj grešci,
- *fatal* – razina signala koja uzrokuje pad ispitnog slučaja,
- *pass* – razina koja uzrokuje prolazak ispitnog slučaja,
- te *warn*, *debug*, *notice* – razine koje se koriste za dokumentacijske svrhe.

S ovim pravilima se istovremeno definira tok izvođenja ispitivanja i verifikacija prolaznosti. Verifikacija prolaznosti ispitnog slučaja se postiže izvođenjem pravila do trenutka u kojem se kreira signal koji ima razinu *fatal* (ispitni slučaj nije zadovoljen) ili *pass* (ispitni slučaj je zadovoljen).

## 4.2. Arhitektura sustava za izvođenje testa

Kako bi izradili sustav koji će vršiti samo ispitivanje i brinuti se za računalne resurse korištene pri ispitivanju prvo ćemo definirati arhitekturu Lusca sustava. Već je uočena potreba da sustav mora biti mrežna aplikacija kako bi se mogla nositi s problemom upravljanja okolinom raspoređenom po računalnoj mreži. Arhitektura je kreirana kao centralizirana struktura koju sačinjava pokretač sustava kao centralni element, te proizvoljni broj agenata postavljenih na računala koji sačinjavaju računalnu mrežu namijenjenu za ispitivanje. Sustavom se upravlja putem korisničke konzole, a arhitektura je prikazana na slici 4.5.




Slika 4.5: Arhitektura Lusca sustava

Pokretač Lusca sustava upravlja programskim agentima na udaljenim računalima i time postiže zadani cilj. Sami agenti tijekom ispitivanja imaju svoje lokalne zadatke koje moraju izvršiti i ne znaju za zadatke ostalih agenata u sustavu. Korisnik pristupa sustavu putem korisničke konzole, odnosno komunikacije s pokretačem Lusca sustava koji ima sve informacije o stanju ispitnih paketa, te samim agentima.

U narednim potpoglavljima su detaljnije opisane uloge pokretača sustava i agenta.

### 4.2.1. Uloga pokretača sustava

Pokretač Lusca sustava je centralni program sustava s najvećim znanjem o procesu ispitivanja. On služi za pohranu Lusca ispitnih paketa, izdavanje zadataka agentima i pohranu rezultata ispitivanja. Također se brine za dohvat aplikacije s udaljenog poslužitelja i prevođenje aplikacije u izvršni kôd.

Prilikom početka ispitivanja, pokretač odabire agente koji će preuzeti ulogu pojedinog domaćina u ispitnom slučaju nakon čega im njihove zadatke. Ukoliko ne postoji dovoljno agenata u sustavu, pokretač dojavljuje pogrešku pri pokretanju ispitivanja. Jednom postavljene uloge agentima ostaju do kraja ispitivanja te se ti agenti ne mogu koristiti za izvršavanje drugih ispitnih paketa do trenutka kada je ispitivanje gotovo.

Pokretač pokreće ispitni paket na način da izvodi jedan po jedan ispitni slučaj iz zadanog paketa. Tijekom samog ispitivanja pokretač se brine da se primljeni signali propagiraju na agente koji imaju zadatke koji se pokreću na taj signal. Pokretač pohranjuje i vrijednosti svih varijabli tijekom izvođenja ispitivanja, te pronalazi ispravne vrijednosti varijabli kada ih neki agent zatraži. Po primitku signala za prolazak ili pad ispitnog slučaja, pokretač zadaje agentima naredbe da se pripreme za idući ispitni slučaj iz ispitnog paketa. Kada su izvršeni svi ispitni slučajevi u paketu koji se mogu izvršiti pokretač izdaje naredbu svim agentima da

pripreme svoje datoteke i dnevnike za dohvat. Nakon što su svi agenti dojave da su datoteke spremne, pokretač preuzima njihove rezultate i pakira ih sa svojim dnevnikom u jedinstvenu datoteku koju korisnik može preuzeti.

Putem konzole korisnik predaje pokretaču Lusca ispitni paket i zadaje komandu za pokretanje ispitivanja. Tijekom ispitivanja korisnik putem konzole prati tijek izvođenja i na kraju ispitivanja preuzima datoteku rezultata ispitivanja.

Pokretač Lusca sustava posjeduje dodatne informacije o brzini procesora kojima raspolaže, pokrenutim procesima, opterećenju procesora, te opterećenju radne memorije do kojih korisnik može putem konzole. Iste informacije pokretač može dati o pojedinom agentu u sustavu u bilo kojem trenutku.

### 4.2.2. Uloga agenta sustava

Agent Lusca sustava je program koji se brine za kontrolu računala na kojem je postavljen i preuzimanje uloge domaćina za izvođenje zadataka kako je opisano ispitnim slučajem. U njemu je implementirano pokretanje skripti i praćenje rada skripte, obrada izlaznih linija, te generiranje signala i pokretanje zadataka po primitku signala.

Agent također može prevoditi aplikaciju u izvršni kôd, no ne može je dohvatiti s repozitorija već mu izvorni kôd predaje pokretač sa zadatkom da ju prevede i vrati u izvršnom obliku. Na taj način je omogućeno rasterećenje pokretača predajom posla prevođenja aplikacije na dostupne agente koji imaju potrebnu funkcionalnost.

Prilikom ispitivanja agent prima sve potrebne datoteke i priprema se za primanje pojedinog signala na koji mora pokrenuti određenu akciju. Nakon pokretanja skripte agent prati izlaz kojeg skripta generira i na osnovu zadanih pravila poduzima potrebne akcije. Ukoliko je akcija pokretanje nove skripte, agent je pokreće, a ako je akcija generiranje signala, agent šalje signal pokretaču sustava koji ga dalje prosljeđuje. Agent sprema sve izlaze pokrenutih skripti u zasebne datoteke i vodi dnevnik svih akcija koje je trebalo poduzeti. Pri završetku ispitivanja pakira sve kreirane datoteke i predaje ih pokretaču.

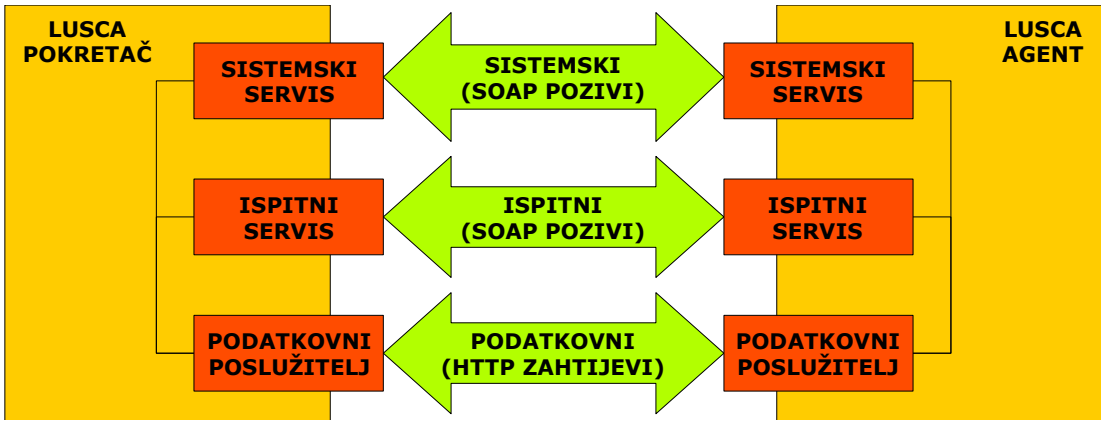
Pojedini agent sustava ne komunicira s ostalim agentima direktno već samo s pokretačem. Sam agent također ne zna detalje što se događa s ukupnim ispitnim slučajem nego izvodi predane mu zadatke od strane pokretača. Ujedno, korisnik ne pristupa agentima sustava direktno da bi iščitao informacije o resursima računala na kojemu se program agenta izvršava već se to obavlja posredovanjem pokretača sustava.

## 4.3. Komunikacija između elemenata sustava

Kako bi pokretač Lusca sustava slao zadatke agentima, prenosio signale i dobivao sistemske informacije potrebno je kreirati komunikacijske protokole koji bi na zadovoljavajući način prenijeli sve informacije u sustavu. Također, potrebno je definirati i komunikacijski protokol za korisničku interakciju putem konzole sustava. Uočena je jasna podjela komunikacije između pokretača sustava i agenata na tri osnovna dijela:

- sistemski protokol – komunikacija korištena za razmjenu sistemskih informacija
- ispitni protokol – komunikacija tijekom ispitivanja
- podatkovni protokol – komunikacija za razmjenu datoteka.

Podatkovni protokol je ostvaren putem HTTP-a dok su ostala dva protokola definirana slijedom poziva udaljenih metoda SOAP servisa agenta i pokretača sustava. Pokretač sustava izlaže metode pojedinog protokola putem SOAP servisa, kao i agent, a slijed poziva je definiran fazom izvođenja ispitivanja ili potrebe za sistemskim pozivom. Prijenos datoteka se ostvaruje na način da se sistemskim protokolom određena datoteka zatraži recimo od agenta, agent ju pripremi i poziva dalje udaljenu metodu pokretača s kojom dojavi da je datoteka spremna nakon čega pokretač preuzima datoteku i dojavi agentu da je datoteka preuzeta. Grafički prikaz podijele komunikacijskih protokola prikazan je na slici 4.6.

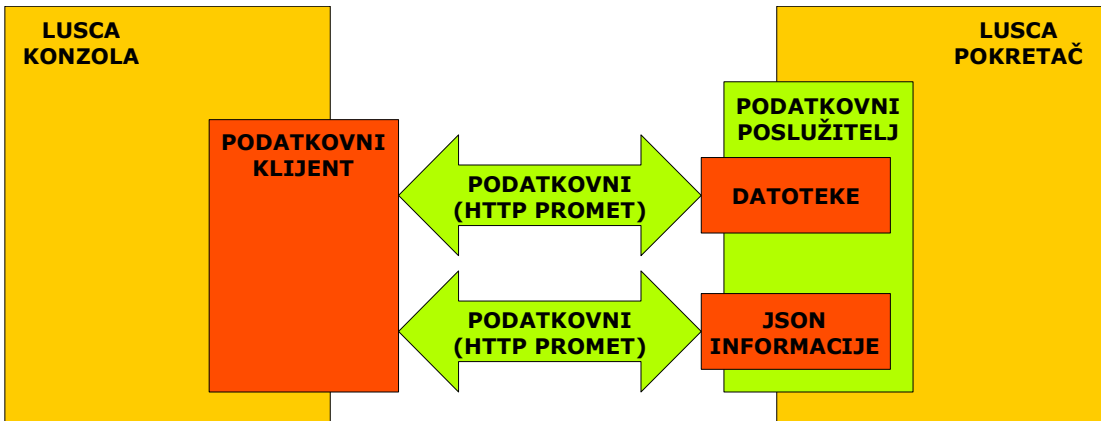



Slika 4.6: Grafički prikaz komunikacijskih protokola

Dodatni protokol je potrebno kreirati za komunikaciju sustava s korisničkom konzolom. Za komunikaciju s konzolom korišten je HTTP protokol podijeljen na dva glavna dijela:

- prijenos datoteka – HTTP-om se prenose datoteke
- prijenos informacija – HTTP-om se preuzimaju podatci u JSON formatu.

Za prijenos informacija o sustavu se koristi JSON zbog njegove jednostavnosti generiranja formata kako bi se što manje opteretilo pokretač sustava. Grafički prikaz komunikacije putem konzolnog protokola je prikazano na slici 4.7.




Slika 4.7: Grafički prikaz komunikacije s konzolom

Korištene su razredi Twisted biblioteke za izgradnju SOAP servisa i ugrađeni HTTP poslužitelj za kreiranje prilagođenog poslužitelja kao i poslužitelja datoteka.

Podjela usluga pokretača sustava i agenta je napravljena definiranjem posebnih putanji za pojedini protokol sustava. Ako se pokretač sustava ima IP adresu 10.0.0.1 i koristi aplikacijsku adresu 10000 sljedeće HTTP putanje su ispravne:

- *http://10.0.0.1:10000/SystemProtocol* – putanja SOAP servisa za sistemski protokol
- *http://10.0.0.1:10000/TestProtocol* – putanja SOAP servisa za ispitni protokol
- *http://10.0.0.1:10000/WWW* – putanja HTTP poslužitelja
- *http://10.0.0.1:10000/ConsoleProtocol* – putanja servisa za konzolni protokol

Ukoliko agent sustava ima IP adresu 10.0.0.2 i koristi aplikacijsku adresu 10000 sljedeće HTTP putanje su ispravne:

- *http://10.0.0.2:10000/SystemProtocol* – putanja SOAP servisa za sistemski protokol
- *http://10.0.0.2:10000/TestProtocol* – putanja SOAP servisa za ispitni protokol
- *http://10.0.0.2:10000/WWW* – putanja HTTP poslužitelja

Pozivom udaljenih procedura nad definiranim servisima se ostvaruje komunikacija unutar sustava. Sva komunikacija se odvija preko iste aplikacijske adrese što olakšava kreiranje mrežnih filtara ili prilagodbu na zatečeno stanje filtara.

U narednim potpoglavljima ćemo opisati pojedini protokol detaljnije i opisati neke značajnije metode servisa.

### 4.3.1. Sistemski protokol

Sistemski protokol se koristi za prijenos informacija za upravljanje sustavom. Pokretač Lusca sustava ovim protokolom dobiva informacije o stanju računala na kojem se nalazi agent (procesor, memorija, pokrenuti procesi itd.), zadaje komande za prevođenje aplikacije na agentu i sl. Agent sustava koristi ovaj protokol pri samoj inicijalizaciji kako bi se prijavio pokretaču sustava pri čemu mu predaje osnovne informacije o sebi (svoj identifikator, da li može prevoditi aplikacije i sl.) i odjavio u slučaju da više nije dostupan.

Datoteka koja definira sistemski protokol se nalazi u odgovarajućem modulu (*ta* za agenta, *te* za pokretača sustava) i naziva se *systemprotocol.py*. Proširenje sistemskog protokola se zvodi dodavanjem odgovarajućih metoda sa *soap_* prefiksom *TE_SystemProtocol* razredu za pokretača sustava, odnosno *TA_SystemProtocol* razredu za agenta.

[vidjeti da li ima smisla pričati o samom protokolu ovdje.]

### 4.3.2. Protokol za prijenos datoteka

Protokol za prijenos podataka ili datoteka kod pokretača i agenta sustava se nalazi na */WWW* HTTP putanji. Ovaj protokol se koristi unutar sustava na način da jedna strana predaje komandu putem sistemskog ili ispitnog protokola o zatraženim datotekama, druga strana priprema traženu datoteku stavljajući je u odgovarajući posluženi direktorij i čeka komandu da je može obrisati. Prva strana preuzima datoteku nakon čega izdaje komandu da je datoteka preuzeta uspješno. Unutar posluživanog direktorija se nalazi struktura poddirektorija koji se koriste u posebne svrhe. Datoteka u kojoj se nalazi definicija ovog protokola je *wwwprotocol.py* u odgovarajućim modulima i sadrži definicije razreda koji se koriste kao resursi ukupnog protokola pokretača ili agenta.

Upravljač sustava dijeli direktorije na sljedeći način:

- */WWW/apps* direktorij – sadrži zapakirani izvorni kôd pojedine aplikacije koju treba prevesti neki od agenata u sustavu
- */WWW/reports* direktorij – može poslužiti za postavljanje izvještaja o ispitivanju
- */WWW/results* direktorij – sadrži rezultate ispitivanja pojedinim ispitnim paketom
- */WWW/tests* direktorij – sadrži datoteke ispitnih paketa postavljenih na sustav.

Prilikom dohvata pojedine datoteke s pokretača potrebno je kreirati HTTP GET zahtjev s ispravnom putanjom nakon čega pokretač odgovara s zatraženom datotekom. Direktorij *results* se koristi za preuzimanje datoteke rezultata ispitivanja putem korisničke konzole. Direktoriji *apps* i *tests* se koriste od strane agenata za preuzimanje ispitnih zadataka i izvornog kôda aplikacije koju trebaju prevesti.

Agent sustava ima nešto jednostavniju podjelu vlastitog direktorija:

- */WWW/apps* direktorij – sadrži zapakirani izvršni kôd pojedine aplikacije koju je agent preveo
- */WWW/tp_id* direktorij – sadrži strukturu direktorija tijekom ispitivanja za pojedini ispitni paket gdje *tp_id* predstavlja identifikator ispitnog paketa
- */WWW/tp_id/tc_id* direktorij – sadrži izlazne datoteke ispitivanja za pojedini ispitni slučaj pri čemu je *tc_id* identifikator ispitnog slučaja.

Prikazanim putanjama na agentu pristupa pokretač sustava kada mu agent dojavi da je pojedina datoteka spremna za preuzimanje, primjerice nakon prevođenja aplikacije ili kraja izvođenja ispitnog slučaja. Direktoriji ispitnih paketa i pojedine aplikacije se brišu nakon što je pokretač preuzeo potrebne datoteke.

### 4.3.3. Ispitni protokol

Tijekom ispitivanja koristi se ispitni protokol za prenošenje informacija o početku/kraju ispitivanja i prijenos signala te vrijednosti varijabli. Putanja ispitnog protokola se dodatno raspoređuje po ispitnim paketima i ispitnim slučajevima kako bi se što jednostavnije omogućio pristup ispravnim podacima. Tako primjerice tijekom izvršavanja ispitnog slučaja *slucaj-01* iz ispitnog paketa *paket-01* kreiraju se dodatne putanje kod pokretača i agenta:

- */TestProtocol/paket-01* – razmjena poruka vezanih uz ispitni paket
- */TestProtocol/paket-01/slucaj-01* – razmjena poruka vezanih uz ispitni slučaj.

Navedene putanje se dinamički kreiraju na prilikom početka ispitivanja. Na razini ispitnog paketa se pozivaju metode za početak pripreme ispitnog paketa i brisanje rezultata ispitnog paketa na agentu, a na pokretaču sustava metode za dojavu o uspješnom dohvatu datoteke ispitnog paketa. Za putanju ispitnog slučaja se razmjenjuju poruke o generiranim signalima, prenose vrijednosti varijabli korištenih tijekom ispitivanja i poruke za početak/kraj pojedine faze ispitivanja.

Ispitni protokol je definiran kao SOAP servis s podputanjama koji su zasebni SOAP servisi s definiranim metodama za udaljene pozive u datoteci *testprotocol.py* u odgovarajućim modulima za agenta i pokretača. Implementacija je izvedena na način da svaki od razreda podservisa koji služe za razmjenu poruka ima instancu kontrolora preko kojeg se izvode

potrebne akcije. Tako primjerice na agentu imamo razred *TA_TestPackage* kao servis za ispitni slučaj i pripadajući kontroler realiziran razredom *TPControler*. Za sam ispitni slučaj koriste se razredi *TA_TestCase* kao servis i *TCControler* kao kontrolor za izvođenje akcija. Ovakva podjela razreda je napravljena kako bi se što uočljivije razdijelio programski kôd na komunikacijski dio i dio koji ostvaruje programsku funkcionalnost.

Spomenuto je da se koriste varijable tijekom ispitivanja što je pojašnjeno detaljnije u ovom paragrafu. Pokretač Lusca sustava drži vrijednosti svih korištenih varijabli bile one lokalne ili globalne varijable. Lokalne varijable se smatraju varijable definirane u *Batch* elementu i pristupanje tim varijablama se izvodi na način da se koristi iduća sintaksa *domaćin.varijabla* za dohvat vrijednosti varijabla na računalu koje ne izvršava ovaj *Batch*. Za dohvat iste varijable na izvršitelju *Batch* elementa dovoljno je navesti ime varijable za dohvat vrijednosti. Ukoliko se koristi globalna varijabla deklarirana u *Definitions* elementu, dovoljno je navesti samo ime varijable. Pokretač sustava mora spremati sve vrijednosti varijabli i razlučiti prilikom zahtijeva za vrijednost varijable ispravnu varijablu čiju će vrijednost vratiti. Rad s varijablama ispitivanja je implemetiran razredom *VarSpace* unutar datoteke ispitnog protokola. Razred posjeduje metode za postavljanje i dohvat vrijednosti varijable:

```
class VarSpace:
        ...
        def set_var(self, who, name, value):
                ...
        def get_var(self, who, name):
                ...
```

Metoda za postavljanje vrijednosti varijable prima identifikator domaćina s kojega je zahtjev stigao (*who*) kako bi se utvrdila ispravna varijabla koju postavlja, te samo ime (*name*) i vrijednost varijable (*value*). Isti parametri se predaju za dohvat varijable izuzev vrijednosti koja je u ovom slučaju povratna vrijednost metode. Pri pretraživanju prostora varijabli prvo se ispituje da li je zatražena varijabla lokalna domaćinu koji ju je zatražio. Ako nije pronađena kao lokalna, varijabla se dalje traži kao lokalna varijabla na drugim domaćinima, te ako ni tada nije pronađena varijabla, pretražuje se prostor globalnih varijabli.

Ispitivanje pojedinim slučajem započinje na način da pokretač nakon pripremne faze pokrenute pozivima udaljenih metoda *soap_preparation* na agentima odašilje svima koji sudjeluju u tom ispitivanju ugrađeni signal za početak ispitivanja (*START*) i počinje razmjena signala. Prilikom primitka signala izvedenog pozivom metode *soap_rcv_signal* servisa za ispitni slučaj pokretača, signal se prosljeđuje svima koji imaju zadatke upaljene tim signalom pomoću poziva udaljenih metoda *soap_rcv_signal* servisa za ispitni slučaj agenta. Nakon primitka signala s razinom *fail* ili *pass* pozivaju se udaljene metode agenata *soap_conclusion* za pokretanje faze zaključenja ispitivanja i prelazi se na idući ispitni slučaj. Nakon što su pokrenuti svi slučajevi pozivaju se metode agenata za kraj izvršavanja ispitnog paketa i preuzimaju se datoteke rezultata s pojedinog agenta.

### 4.3.4. Protokol za konzolu

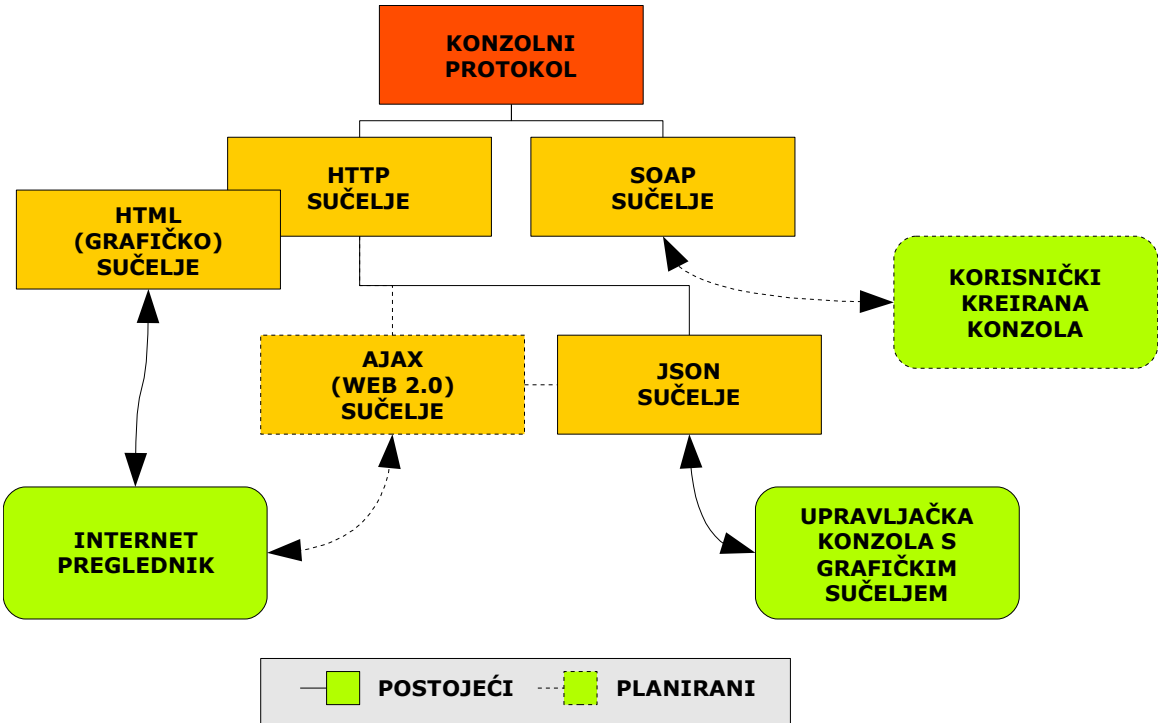
Kako bi se omogućilo interakciju s Lusca sustavom definiran je i protokol putem kojega korisnička konzola komunicira s pokretačem Lusca sustava. Putem konzolnog protokola

sustavu se predaju Lusca ispitni paketi, preuzimaju datoteke rezultata ispitivanja i dobivaju informacije o:

- aplikacijama koje sustav ispituje
- ispitnim paketima koji su predani sustavu
- prijavljenim agentima na sustav
- dodatne informacije o računalnim resursima računala u sustavu.

Konzolni protokol je zamišljen kao protokol s više podatkovnih sučelja, pa je tako primjerice moguće dobiti informacije o radu sustava putem SOAP poziva ili u JSON formatu. Pruženo je i jednostavno HTML sučelje preko kojeg se internet preglednikom može pristupiti sustavu i izvršiti osnovne operacije.

Implementacija konzolnog protokola je izvedena na način da se sve potrebne informacije dobivaju iz instance razreda *TE_ConsoleProtocol* u *consoleprotocol.py* datoteci pozivom pripadajućih metoda. Podatkovna sučelja su izvedena na način da dobivene podatke prikazuju u potrebnom formatu, te je time omogućeno jednostavno proširenje protokola konzole dodavanjem potrebnih podatkovnih sučelja. Grafički prikaz izvedenog konzolnog protokola i mogućih nadogradnji je dan na slici 4.8.



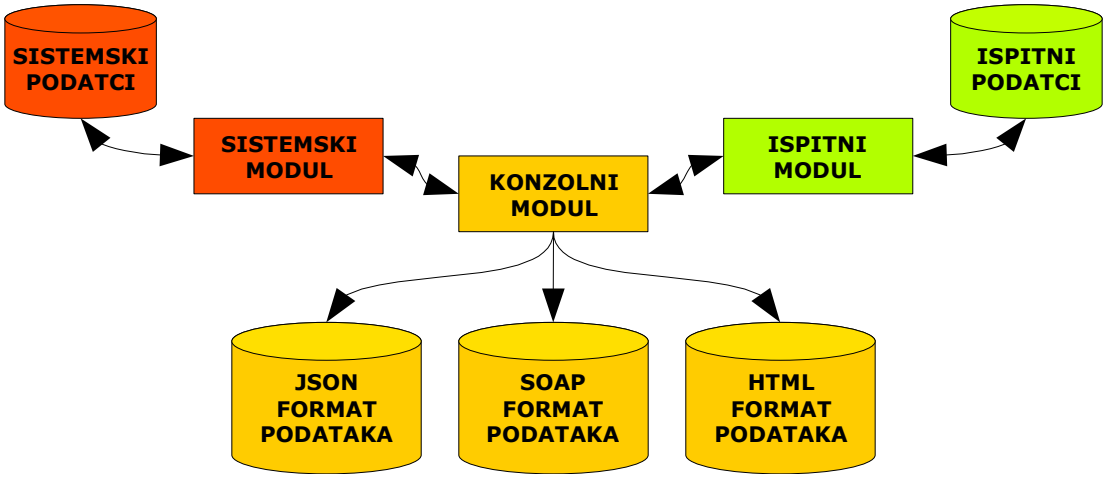

Slika 4.8: Prikaz raspodjele konzolnog protokola

Protokol konzole je definiran u datoteci *consoleprotocol.py* unutar modula pokretača sustava, te su pojedina sučelja dodatno definirana unutar pod-modula *console* u datotekama *http.py* i *soap.py* za pripadajuće podatkovno sučelje. Tako se konzolni protokol može koristiti u SOAP i HTTP inačici na idućim putanjama:

- */ConsoleProtocol/http* – sadrži jednostavno HTML sučelje s podputanjama za izvođenje pojedinih akcija

- */ConsoleProtocol/http/json* – sadrži JSON podatkovno sučelje za dohvat informacija o sustavu u JSON formatu
- */ConsoleProtocol/soap* – sadrži SOAP podatkovno sučelje za dohvat informacija o sustavu.

Sa stajališta izvora podataka grafički možemo prikazati rad modula za konzolni protokol na idući način slikom 4.9.




Slika 4.9: Funkcija konzolnog modula pri pretvorbi podataka

Za predavanje ispitnih paketa koristi se HTTP POST metoda nad odgovarajućom HTTP putanjom: */ConsoleProtocol/http?page=upload_action* iza koje se nalazi *http* modul za obradu zahtjeva i predanu datoteku pohranjuje. Ovdje ćemo još razmotriti metode za dohvat informacija o sustavu u JSON formatu, a za potpuni pregled metoda pogledati dokumentaciju izvornog kôda.

Kako bi dobili podatke u JSON formatu potrebno je kreirati HTTP GET zahtjev sa sljedećim putanjama za odgovarajuće akcije:

- */ConsoleProtocol/http/json?action=getTests* – JSON lista podataka s popisom svih ispitnih paketa s njihovim statusom
- */ConsoleProtocol/http/json?action=getAgents* – JSON lista svih prijavljenih agenata u sustavu s njihovim opisom i statusom
- */ConsoleProtocol/http/json?action=getApplications* – JSON lista svih aplikacija koje Lusca pokretač ispituje
- */ConsoleProtocol/http/json?action=getTestInfo&test_id=some_id* – JSON objekt s opisom ispitnog paketa *some_id* a sadrži listu ispitivanih aplikacija i listu ispitnih slučajeva.

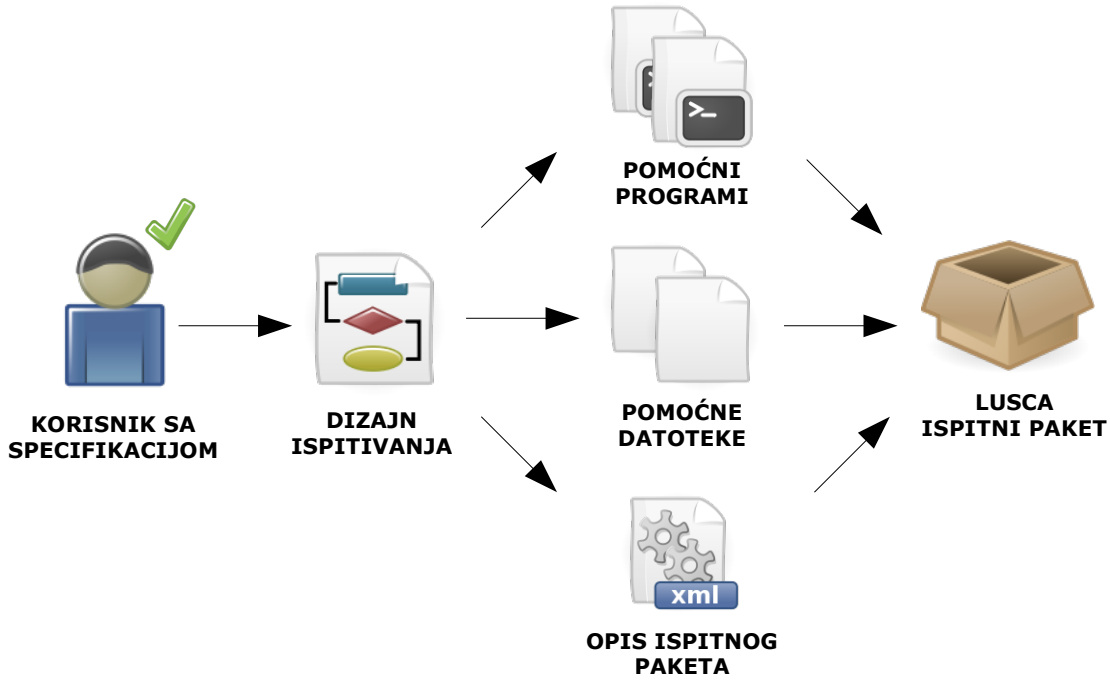
Korisnička konzola pozivom ispravnih metoda dobiva podatke koje obrađuje na odgovarajući način i prikazuje ih u grafičkom sučelju. Preporučuje se uporaba JSON podatkovnog sučelja kako bi se izbjeglo veće opterećenje sustava čestim zahtjevima za složenim podacima.

## 4.4. Prikaz procesa ispitivanja

U ovom poglavlju ćemo objasniti Lusca sustav kroz detaljniju razradu fazi rada sa sustavom počevši od kreiranja ispitnog paketa do finalne faze preuzimanja datoteke rezultata ispitivanja.

### 4.4.1. Kreiranje ispitne datoteke

Nakon definiranja dokumenta ispitne specifikacije može se pristupiti procesu izrade ispitnih paketa. Proces je prikazan dijagramom na slici 4.10.




Slika 4.10: Izrada Lusca ispitnog paketa

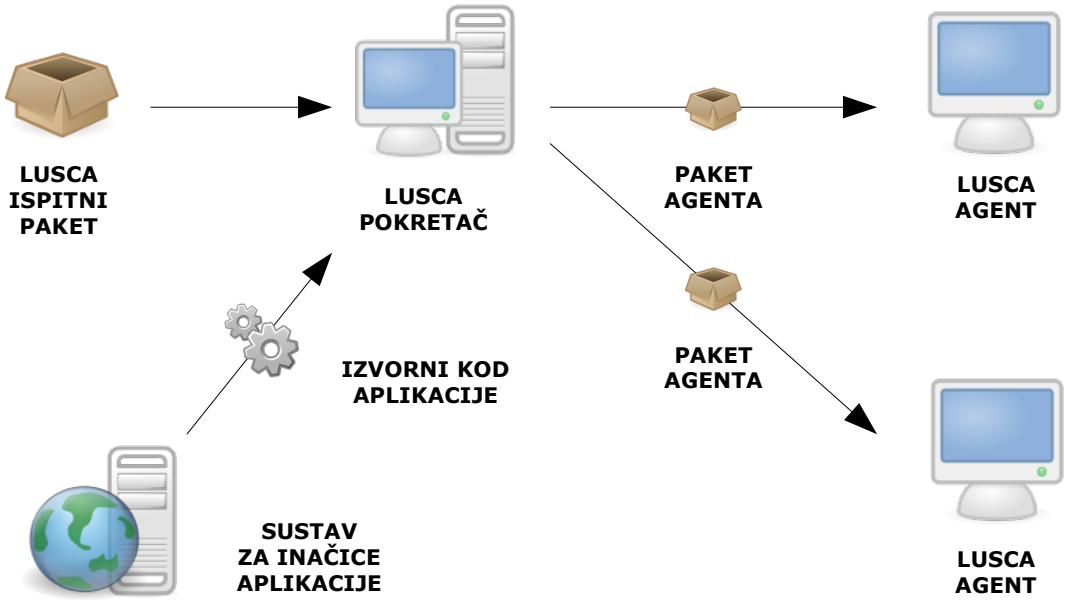
Korisnik pristupa dizajniranju procesa ispitivanja pri čemu kreira dijagrame toka testa i donosi zaključke o potrebnim resursima za ispitivanje. Na osnovi zaključaka i dizajnu procesa ispitivanja korisnik kreće u izradu pomoćnih programa i datoteka, te ih opisuje u formatu Lusca opisa ispitnog paketa kreiranjem XML datoteke. Krajnji korak u ovoj fazi je kompresija kreiranih datoteka (pomoću *tar* i *gzip* alata) u datoteku Lusca ispitnog paketa.

### 4.4.2. Predaja datoteke ispitnog paketa sustavu

Kreirane ispitne pakete korisnik predaje sustavu putem korisničke konzole (bilo putem grafičkog alata ili HTML sučelja). Sustav predane datoteke raspakira i učitava opis pojedinog ispitnog paketa. Ukoliko je opis neispravan korisniku se ispisuje poruka o grešci i ispitni paket se odbacuje. Ukoliko je opis ispravan sustav pohranjuje ispitni paket i prikazuje ga kao dostupnog za pokretanje ispitivanja putem korisničkog sučelja.

### 4.4.3. Priprema resursa za izvođenje testa

Kada korisnik zada komandu za pokretanje ispitnog paketa pokretač izvodi potrebne akcije za pripremu sustava na ispitivanje. U ovoj fazi se preuzima izvorni kôd aplikacije s definiranog repozitorija i prevodi ga se u izvršni. Grafički prikaz je dan na slici 4.11.



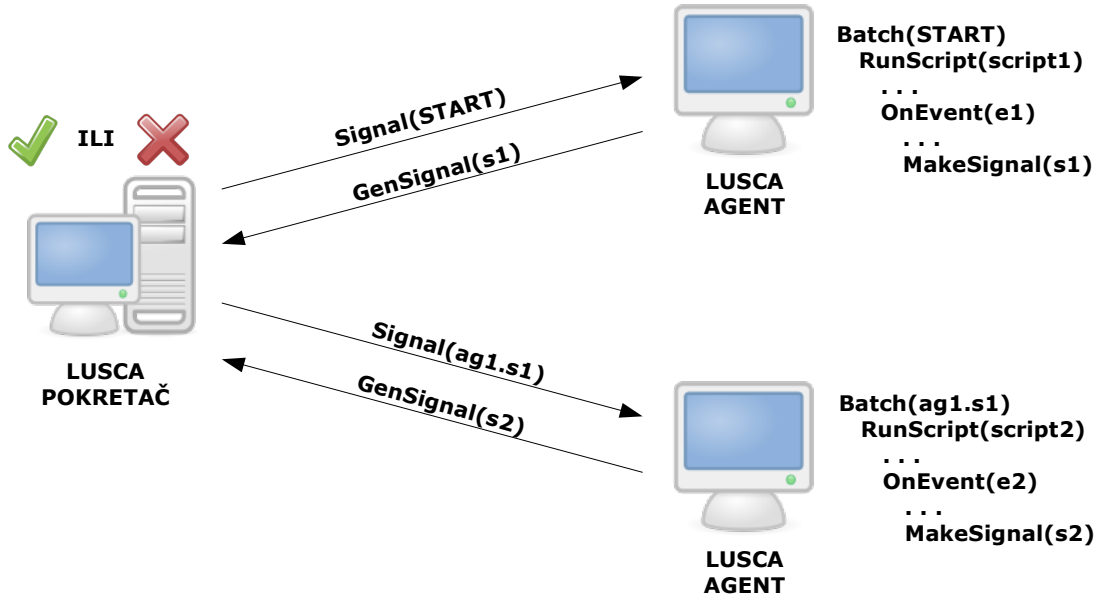

Slika 4.11: Priprema za izvođenje ispitivanja

Nakon uspješnog prevođenja aplikacije (bilo to na pokretaču sustava ili na nekom od agenata) aplikacija je spremna za distribuciju. Prvo se ispituje da li pokretač poznaje dovoljan broj agenata za izvođenje ovog ispitnog paketa, te u zadovoljavajućem slučaju odabire agente koji će se koristiti i predaje im njihove uloge u ispitnom paketu s potrebnim resursima za ispitivanje. Agenti koje je pokretač odabrao se više ne mogu odabrati za uporabu u novim ispitnim paketima do trenutka kada je ispitivanje završeno. Nakon dojave svih agenata da su preuzeli datoteke uspješno pokretač gradi slijed ispitnih slučaja koje će pokretati i započinje izvođenje jednog po jednog ispitnog slučaja.

### 4.4.4. Izvođenje ispitnog slučaja

Izvođenje ispitnog slučaja se odvija na način da se poštuju definirana pravila iz opisa ispitnog slučaja. Agenti Lusca sustava na računalu domaćinu pokreću skripte na primitak odgovarajućeg signala i prate izlaz pri radu skripte kako bi prepoznali definirane linije na koje moraju reagirati. Kada se pojavi odgovarajuća linija agent generira signal ili pokreće novu instancu skripte kako je već definirano pravilima. Pokretač sustava se brine da svi kreirani signali budu ispravno proslijeđeni agentima koji imaju zadatke pokretane tim signalom. Ispitivanje se odvija do trenutka kada pokretač primi signal razine *fatal* ili *pass* nakon čega slijedi zaključna faza ispitivanja i prikupljanje rezultata. Izvođenje ispitnog slučaja je prikazano na slici 4.12.

Ispitivanje započinje generiranjem ugrađenog signala *START* koji se prosljeđuje svim agentima koji sudjeluju u ovom ispitivanju i započinje svoj zadatak pokretan ovim signalom. Kao akcija izvršenja zadatka je pokretanje skripte s potrebnim parametrima. Ukoliko je skriptu potrebno pokrenuti s vrijednosti jedne ili više varijabli, agent zahtijeva od pokretača



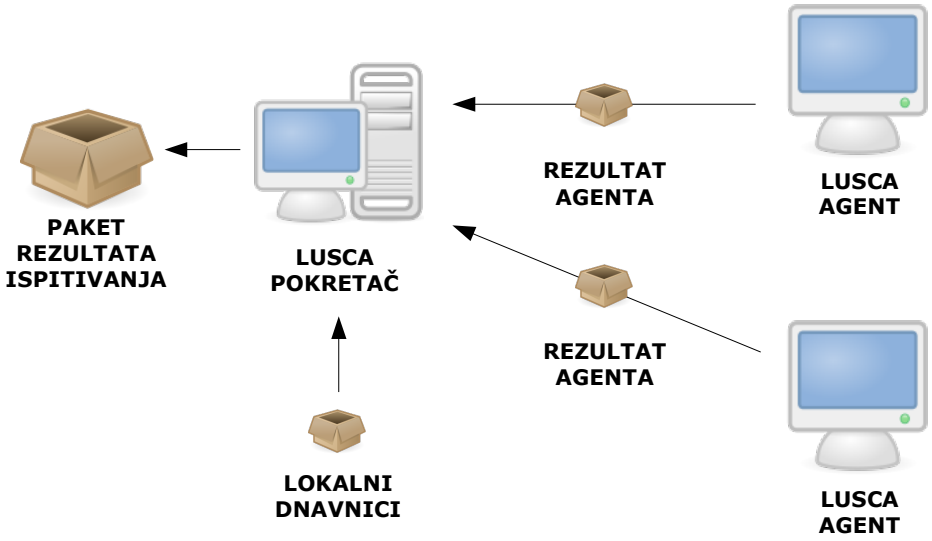

Slika 4.12: Izvođenje ispitnog slučaja

vrijednosti svih varijabli potrebnih za pokretanje. Nakon što ih dobije agent pokreće skriptu s dobivenim vrijednostima. Tijekom izvođenja skripte, agent obrađuje tekst generiran na standardnom izlazu skripte i pronalazi linije događaja, datoteke ili povratne vrijednosti. U trenutku kada je pronađena linija za događaj, agent na osnovi identifikatora događaja pronalazi u pripremljenoj tablici listu akcija koje mora poduzeti. U danom primjeru identifikator događaja je *e1*, a signal kojeg je potrebno generirati je *s1*. Agent poziva udaljenu metodu pokretača sustava za dojavu o signalu. Pokretač pomoću zaprimljenih podataka o signalu (od koga je signal primljen, identifikator i razina signala) odlučuje što treba učiniti. Ukoliko je razina signala, uzmimo za primjer, *info* pokretač pronalazi u pripremljenoj tablici listu agenata koji moraju primiti taj signal i pokretač im ga šalje. Agenti koji su primili taj signal pokreću svoje zadatke i ciklus se dalje nastavlja sve do trenutka kada se kreira signal posebne razine koji označava ispravnost (razina *pass*) ili neispravnost (razina *fatal*) ispitnog slučaja. Nakon verifikacije prolaznosti ispitnog slučaja pokreću se rutine za zaključenje ispitivanje i nastavlja se ispitivanje sa sljedećim ispitnim slučajem. U trenutku kada su gotovi svi ispitni slučajevi koji su se mogli pokrenuti kreće se u fazu prikupljanja rezultata ispitivanja.

### 4.4.5. Prikupljanje i pohrana rezultata ispitnog paketa

Kada je samo ispitivanje gotovo, potrebno je prikupiti nastale datoteke na računalo pokretača sustava zajedno s dnevnicima pojedinog agenta o izvršavanju. Prikupljenim datotekama pokretač dodaje i svoj dnevnik izvođenja ispitivanja, te dnevnike o dohvatu aplikacije s udaljenog repozitorija i prevođenju aplikacije u izvršni kôd. Nabrojene datoteke pokretač pakira i vrši kompresiju datoteka (koristeći *tar* i *gzip* alate). Dijagram prikupljanja datoteka je prikazan na slici 4.13.

Korisnik može preuzeti kreiranu datoteku za detaljniju analizu zbivanja tijekom ispitivanja po svakom ispitnom slučaju. Datoteka sadrži hijerarhijsku strukturu direktorija s posebnim direktorijima za svaki ispitni slučaj i direktorijom za pojedinu fazu ispitivanja (*prep*




Slika 4.13: Prikupljanje rezultata ispitivanja

direktorij za pripremu ispitivanja, *exec* direktorij za glavnu fazu i *conc* direktorij za zaključnu fazu).

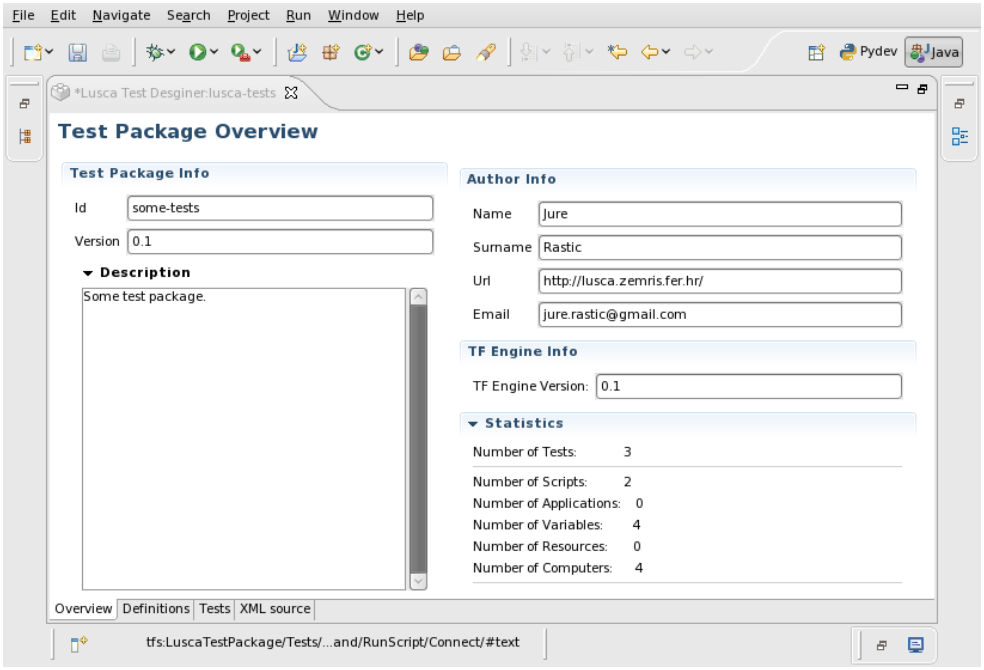
## 4.5. Korisnički alati

Uporaba sustava Lusca je olakšana korisničkim alatima s grafičkim sučeljom koji su izrađeni u sklopu dvaju diplomskih radnji. Student Hrvoje Slaviček je kreirao uređivač Lusca opisa ispitnog paketa [16], a student Dražen Nežić je kreirao korisničku konzolu za upravljanje sustavom [17]. Oba rada su realizirana kao dodatci za Eclipse razvojno okruženje. U narednim potpoglavljima dan je kratki uvod u korištenje i mogućnosti kreiranih aplikacija.

### 4.5.1. Grafički uređivač datoteke opisa testa

Za potrebe kreiranja i uređivanja opisa Lusca ispitnog paketa napravljen je dodatak s grafičkim sučeljom za Eclipse razvojno okruženje. Odabran je oblik Eclipse dodatka za izradu ovog korisničkog alata zbog popularnosti Eclipse okruženja i ideje da korisnik može kreirati ispitne pakete u istom okruženju u kojem razvija samu aplikaciju. Na taj način programer, ispitivač aplikacije ili obični korisnik mogu iz istog okruženja raditi na programskom kôdu aplikacije i ispitnom paketu.

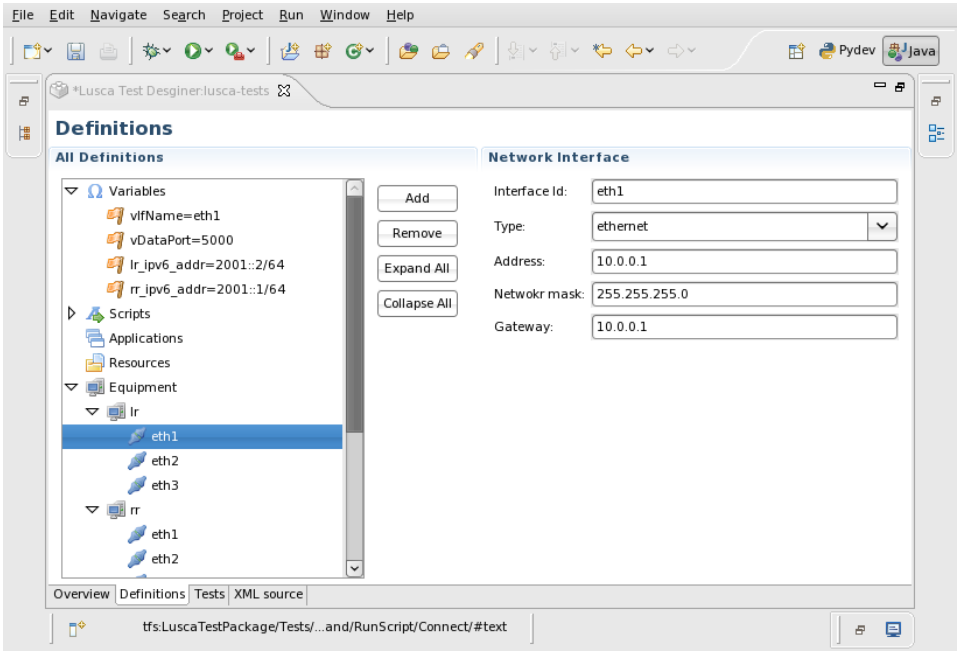
Potreba za grafičkim alatom za izradu opisa ispitnog paketa je proizašla iz jednostavne činjenice što je definirani dijalekt XML jezika složen za ručno pisanje koristeći tekstualni uređivač ukoliko opis postane prevelik. Radi toga je kreiran uređivač s grafičkim sučeljom koji sakriva od korisnika relativno složenu sintaksu kreiranog jezika i omogućuje rad s opisom ispitnog paketa na pregledan način. Sučelje se sastoji od par različitih pogleda koji prate i dijelove opisa ispitnog paketa. Postoji sučelje za unos informacija o samom ispitnom paketu, sučelje za uređivanje opisa resursa i sučelje za uređivanje samih ispitnih slučajeva. Alat omogućuje i pregled dobivenoga XML-a koji se izgrađuje interakcijom korisnika s aplikacijom. Kreiranje opisa ispitnog paketa započinje pokretanjem Eclipse razvojnog okruženja i dodavanjem novog projekta putem čarobnjaka za Lusca ispitni paket. Početni pogled alata je pregled za unos informacija o ispitno paketu i prikazan je na slici 4.14.




Slika 4.14: Pogled na informacije o ispitnom paketu

Prikazani su podatci u poljima za uređivanje informacija o ispitnom paketu (identifikator, verzija i opis), te podatci o autoru ispitnog paketa. Na ovom pogledu može se pregledati i statistički podatci o broju ispitnih slučaja, korištenih skripti i slično.

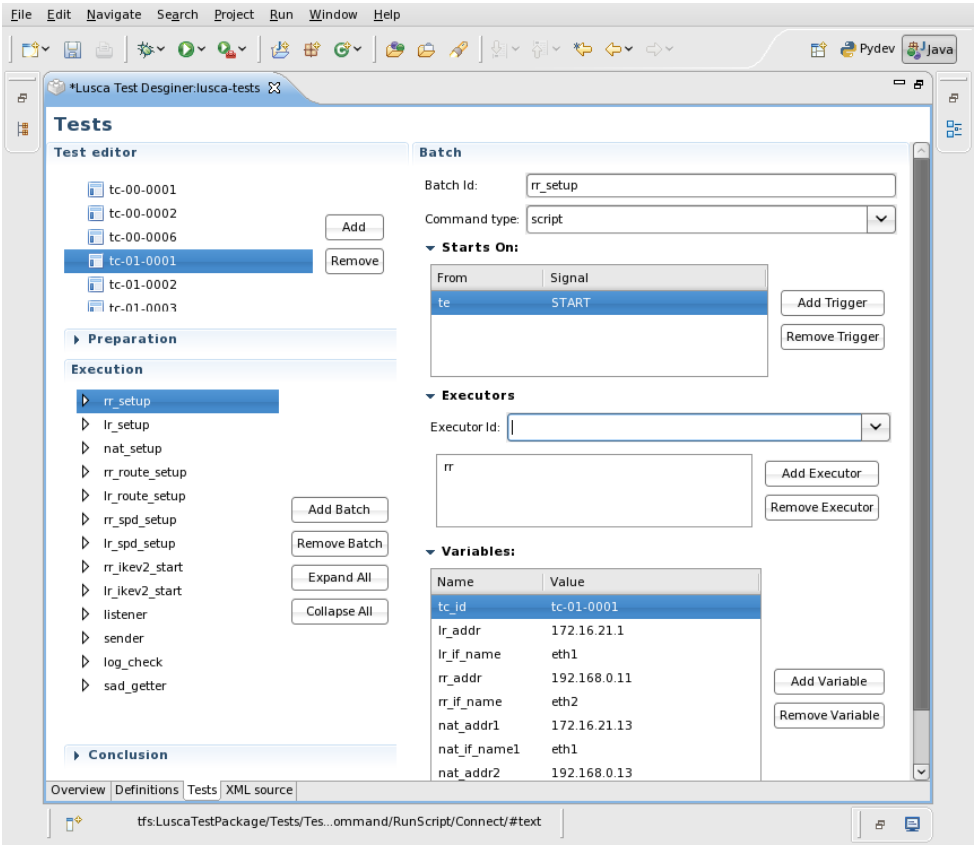
Odabirom pogleda za definicije resursa dobivamo pogled na sučelje za uređivanje resursa prikazanom na slici 4.15.




Slika 4.15: Pogled na definicije resursa ispitnog paketa

S lijeve strane pogleda su prikazani ikonama pojedini elementi koji se koriste u ispitnom paketu, dok su desne strane polja za uređivanje atributa pojedinog resursa. Na slici je odabrana ikona mrežnog sučelja *eth1* računala *lr* čije je parametre moguće promijeniti tekstualnim poljima s desne strane.

Za definiranje pravila izvođenja pojedinog ispitnog slučaja koristi se prilagođeni pogled prikazan na slici 4.16.



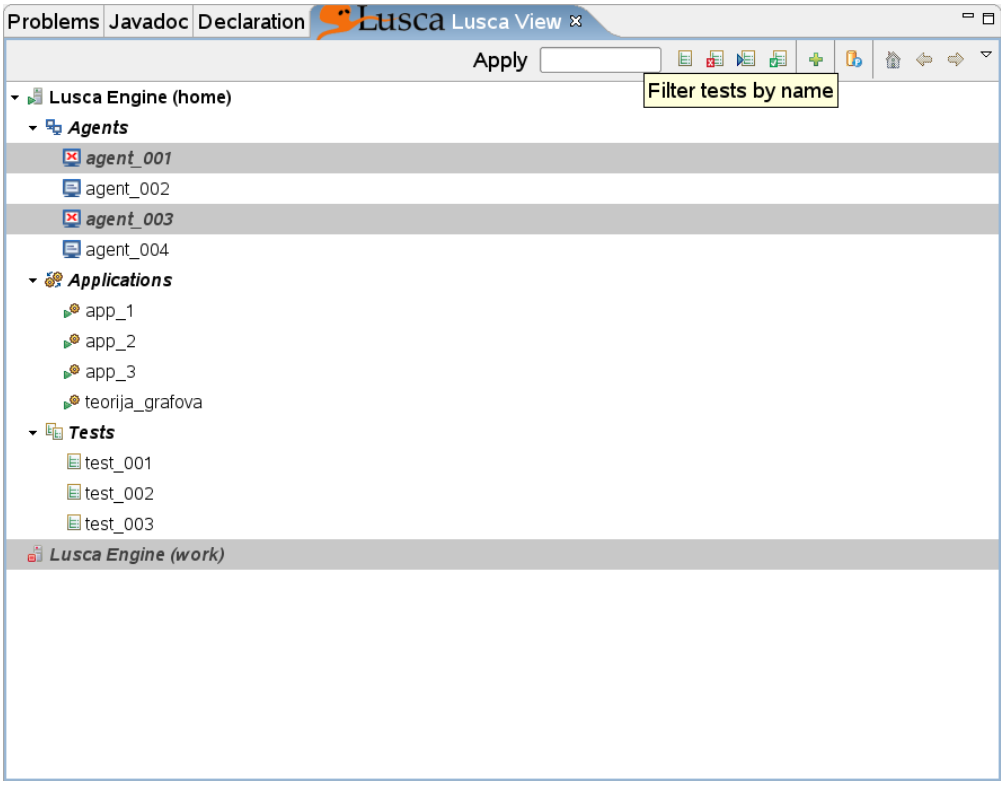

Slika 4.16: Pogled za uređivanje ispitnih slučajeva

S desne strane su posloženi hijerarhijskim redom odozgo prema dolje: lista ispitnih slučajeva, akcije za pripremnu fazu, glavnu i zaključnu fazu ispitnog slučaja pri čemu je otvorena lista zadataka glavne faze ispitivanja. Odabran je zadatak s identifikatorom *rr_setup* čiji se parametri uređuju na desnoj strani pogleda. Desna strana pogleda sadrži polja (odozgo prema dolje) za uređivanje identifikatora zadatka, uređivanje liste signala na koje zadatak počinje, uređivanje liste izvršitelja zadataka, te popisa lokalnih varijabli za ovaj zadatak.

## 4.5.2. Upravljačka konzola

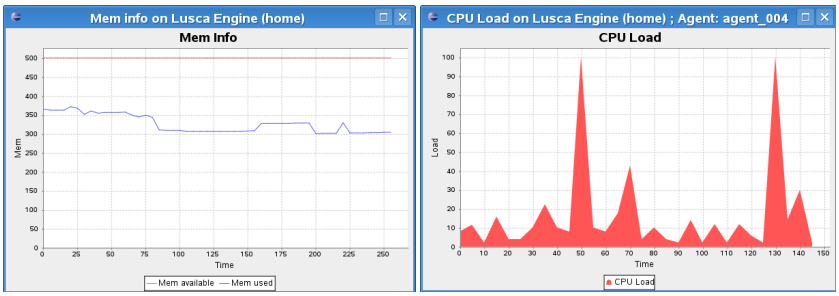
Za interakciju sa samim sustavom napravljena je upravljačka konzola u obliku Eclipse dodatka kako bi se omogućio rad sa sustavom iz istog razvojnog okruženja za izradu aplikacije i izradu ispitnog paketa. Sučelje komunicira putem JSON podatkovnog sučelja s pokretačem Lusca sustava i dobivene podatke prikazuje korisniku u grafičkom sučelju. Osnovni prikaz podataka o Lusca sustava je dan na slici 4.17.

Sučelje prikazuje udaljeni Lusca pokretač i podatke koje je prikupio o osnovnim stavkama: popis agenata, popis ispitivanih aplikacija i popis postavljenih ispitnih slučajeva. U kontekstnom izborniku pojedinog elementa dobiva se popis akcija za odabrani element. Tako je moguće za pojedinog agenta prikazati opterećenje procesora, listu pokrenutih procesa i stanje memorije. Za ispitni paket moguće je odabrani paket pokrenuti, zaustaviti, preuzeti rezultate ispitivanja ili prikazati rezultate za pojedini ispitni slučaj.



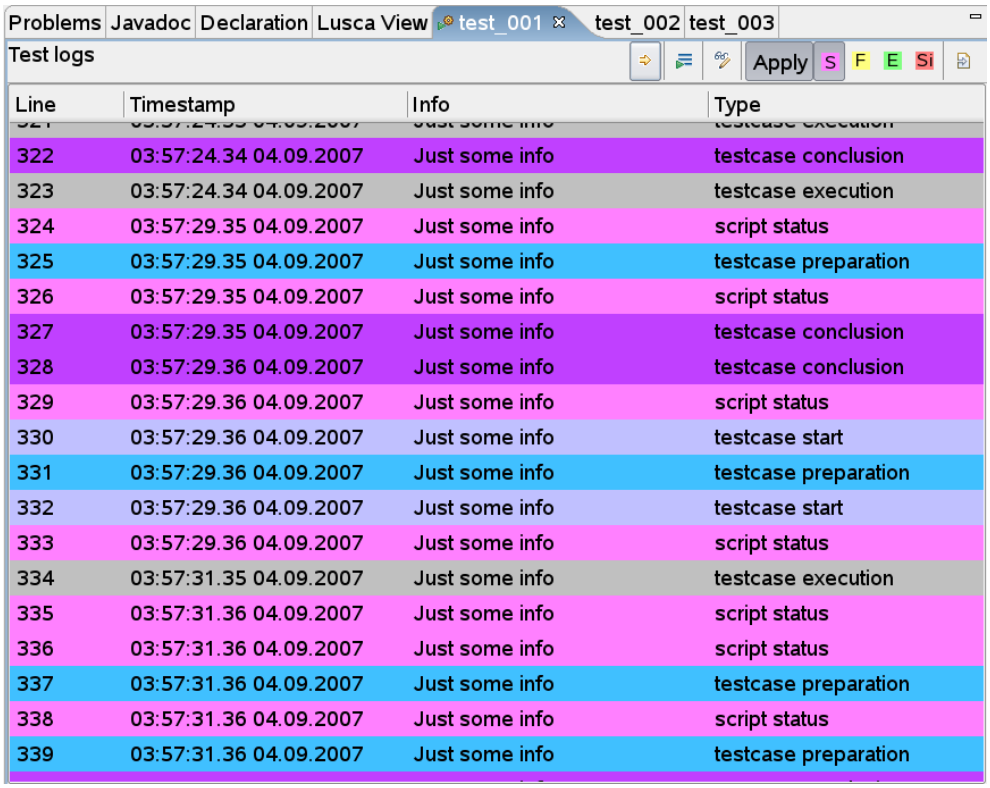

Slika 4.17: Prikaz Eclipse upravljačke konzole Lusca sustava

Prilikom odabira opcije za prikaz računalnih resursa pojedinog elementa sustava (pokretač ili agent sustava) otvaraju se mali prozori s grafovima stanja procesora ili memorije tijekom vremena kako je prikazano na slici 4.18.




Slika 4.18: Prikaz stanja memorije i procesora računala u Lusca sustavu

Tijekom izvođenja ispitnog slučaja moguće je dobiti uvid u događaje i signale koji se prenose tijekom ispitivanja odabirom pogleda za praćenje ispitivanja. Pogled je prikazan na slici 4.19.




| Line | Timestamp | Info | Type |
|---|---|---|---|
| 322 | 03:57:24.34 04.09.2007 | Just some info | testcase conclusion |
| 323 | 03:57:24.34 04.09.2007 | Just some info | testcase execution |
| 324 | 03:57:29.35 04.09.2007 | Just some info | script status |
| 325 | 03:57:29.35 04.09.2007 | Just some info | testcase preparation |
| 326 | 03:57:29.35 04.09.2007 | Just some info | script status |
| 327 | 03:57:29.35 04.09.2007 | Just some info | testcase conclusion |
| 328 | 03:57:29.36 04.09.2007 | Just some info | testcase conclusion |
| 329 | 03:57:29.36 04.09.2007 | Just some info | script status |
| 330 | 03:57:29.36 04.09.2007 | Just some info | testcase start |
| 331 | 03:57:29.36 04.09.2007 | Just some info | testcase preparation |
| 332 | 03:57:29.36 04.09.2007 | Just some info | testcase start |
| 333 | 03:57:29.36 04.09.2007 | Just some info | script status |
| 334 | 03:57:31.35 04.09.2007 | Just some info | testcase execution |
| 335 | 03:57:31.36 04.09.2007 | Just some info | script status |
| 336 | 03:57:31.36 04.09.2007 | Just some info | script status |
| 337 | 03:57:31.36 04.09.2007 | Just some info | testcase preparation |
| 338 | 03:57:31.36 04.09.2007 | Just some info | script status |
| 339 | 03:57:31.36 04.09.2007 | Just some info | testcase preparation |

Slika 4.19: Pregled informacija o ispitivanju

Prikazani pogled omogućuje uvid u proces ispitivanja tijekom samog izvođenja ispitnog paketa. Za svaku informaciju dan je redni broj, vrijeme, tekstualni opis i tip događaja. Prikaz informacija je moguće filtrirati po tipu informacije kako bi se ostvario što jednostavniji uvid u bitne informacije o ispitivanju.

Potrebno je napomenuti da upravljačka konzola ima svojstvo integriranja u sam razvojni proces na način da koristi mogućnosti Eclipse razvojnog okruženja za upravljanje s udaljenim repozitorijem izvornog kôda. Naime, omogućeno je pokretati određeni ispitni paket s trenutnom inačicom izvornog kôda koja je još na korisnikovom računalu. Tako se postiže ispitivanje aplikacije s inačicom kôda koja još nije postavljena u službenom repozitoriju što omogućava ispitivanje aplikacije prije objave radne verzije.

## 4.6. Kratki pregled mogućnosti za uporabnu prilagodbu

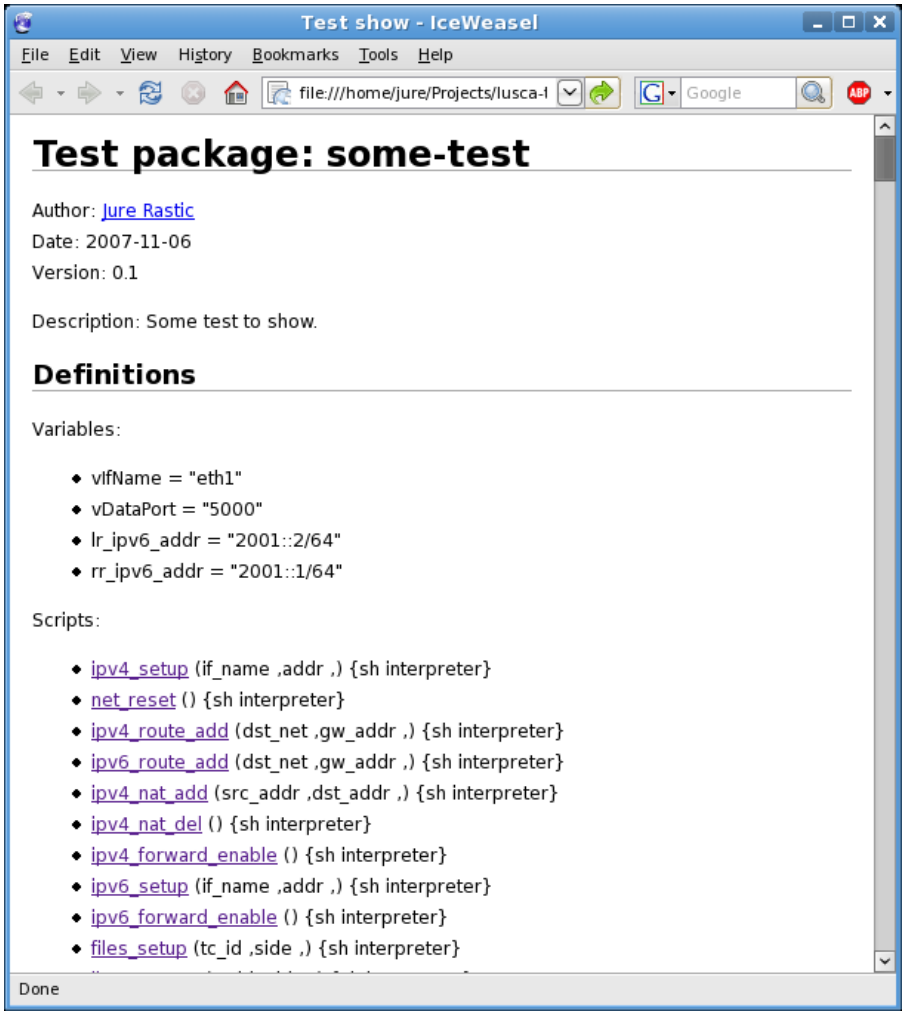
Prilikom vođenja projekta razvoja aplikacije potrebno je voditi projektnu dokumentaciju o svim dijelovima procesa, pa tako i o ispitivanju aplikacije. Dokumentacija o ispitivanju se vodi za cilj i metodu ispitivanja, te kroz vođenje dnevnika izvješća ispitivanja aplikacije kako bi se mogao dobiti brzi uvid u status projekta.

U ovom dijelu su razmatrana dva jednostavna pristupa generiranju projektne dokumentacije o procesu ispitivanju pomoću datoteke koja se predaje Lusca sustavu na ispitivanje i datoteke koju Lusca sustav generira na kraju ispitivanja. Opis Lusca ispitnog paketa može poslužiti kao osnova za generiranje dokumentacije o samom ispitivanju aplikacije, dok datoteka rezultata ispitivanja može poslužiti za izradu prilagođenih izvješća.

### 4.6.1. Kreiranje dokumentacije o ispitnom paketu

Za potrebe vođenja projektne dokumentacije o ispitnim paketima moguće je kreirati prilagođenu aplikaciju za prikaz ispitnog paketa u zahtijevanom formatu koristeći sam opis Lusca ispitnog paketa. Kako je opis definiran u XML jeziku, korištenjem programskih biblioteka za rad s XML-om moguće je crpiti informacije iz opisa kako bi se podatci prikazali na potrebni način. Ovdje ćemo razmotriti primjenu aplikacije XSL tehnologije za obradu XML podataka kako bi kreirali čovjeku pregledan HTML dokument. Na taj način se omogućuje brz uvid u definirane resurse i ispitne slučajeve uporabom internet preglednika koji je sposoban izvršavati XSL transformacije.

Primjer dobivenog HTML dokumenta direktno iz XML opisa ispitnog paketa je prikazan na slici 4.20.

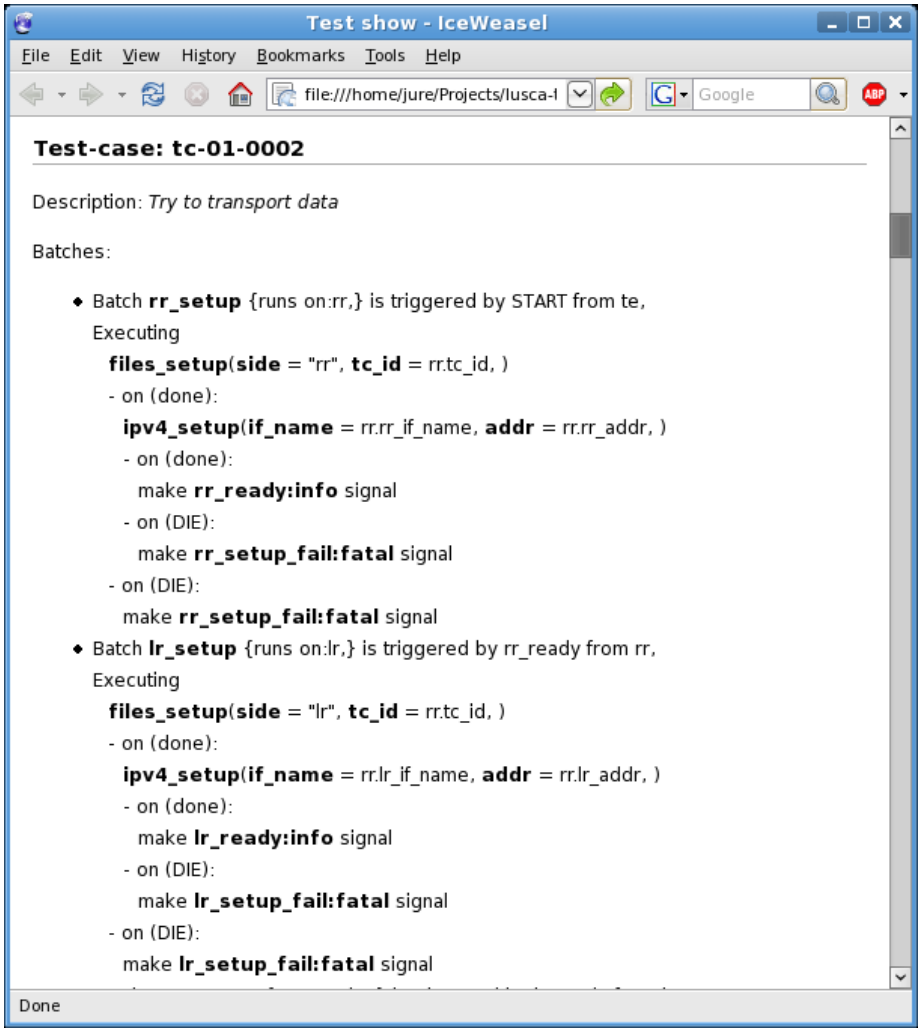



Slika 4.20: Prikaz opisa resursa u prilagođenoj transformaciji

Datoteci opisa ispitnog paketa je dodana XML instrukcija za definiranje XSL transformacije na idući način:

```
<?xml version = '1.0' encoding = 'UTF-8'?>
<?xml-stylesheet type="text/xsl" href="toHtml.xsl"?>
...
```

Druga linija isječka definira datoteku *toHtml.xsl* kao datoteku XSL transformacije koju je potrebno primijeniti nad XML datotekom. Datoteka se nalazi u istom direktoriju kao i sama datoteka opisa, te definira instrukcije za obradu podataka i njihovu pretvorbu HTML oblik. Transformaciju obavlja internet preglednik i iscrtava generirani HTML u korisničkom sučelju. Dobiveni pregled prikazuje listu globalnih varijabli, korištenih skripti i listu ispitnih slučajeva. Početak tijela dokumenta za ispitni slučaj je prikazan na slici 4.21.




Slika 4.21: Prikaz ispitnog slučaja kod prilagođene transformacije

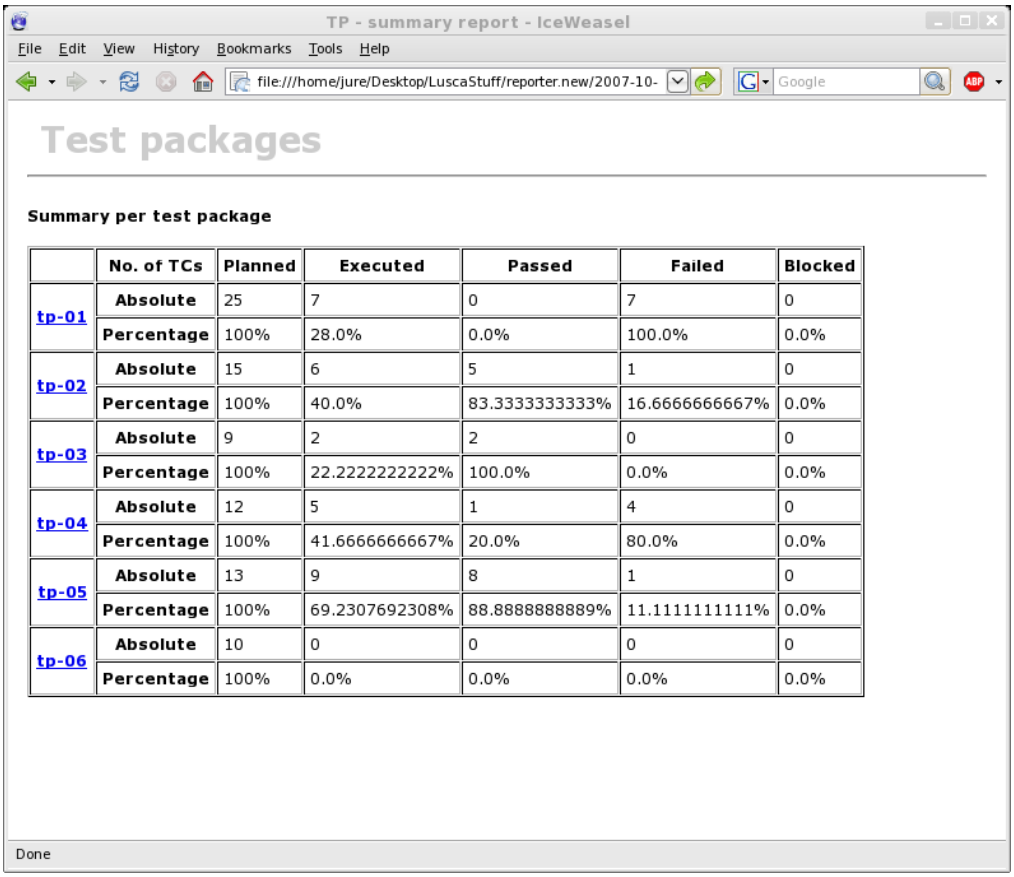
Primjer prikazuje podatke o ispitnom slučaju na čovjeku razumljiviji način od samog sadržaja datoteke opisa, pa se može dobiti jednostavniji uvid u tijek izvođenja ispitivanja.

Prikazani primjer je prilično jednostavan, no ovom metodom se mogu dobiti i složeniji dokumenti u zahtijevanom obliku uz nešto više napora oko kreiranja prilagođene transformacije. Također moguće je dobiti dokumente u bilo kojem zahtijevanom formatu

koji se koristi za dokumentiranje projekta kao što je primjerice OASIS *Open Document Format* (korišten u *OpenOffice.org* programskom paketu aplikacija za urede) ili recimo LaTeX formatu.

### 4.6.2. Izrada prilagođenih izvještaja

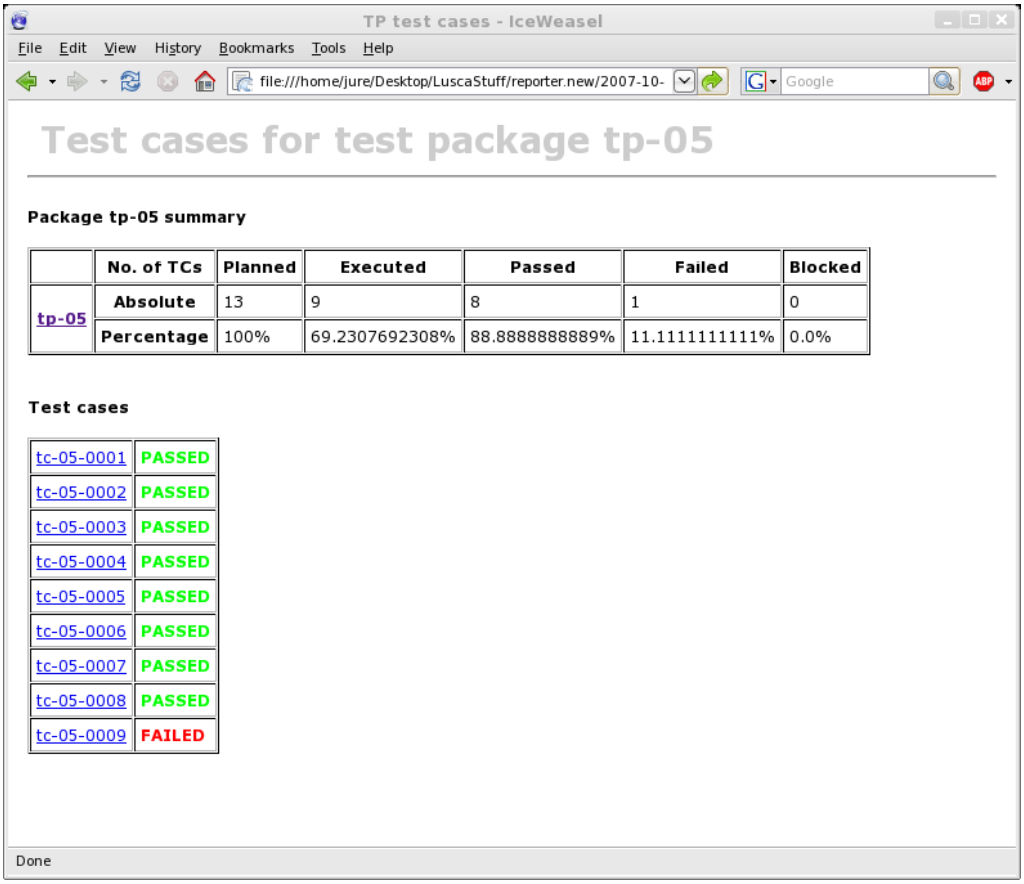
Kada je ispitni paket izveden može se iskoristiti informacije iz dnevnika pokretača i agenta sustava te nastalih datoteka kako bi se preglednije prikazalo rezultate ispitivanja. U tom slučaju potrebno je izraditi aplikaciju koja će preuzetu datoteku rezultata ispitivanja obraditi na način da se kreiraju potrebne datoteke s prilagođenim prikazom dobivenih podataka. Kreirane datoteke zajedno sačinjavaju izvješće o izvođenju ispitnog paketa i primjer prilagođenog izvještaja je dan na slici 4.22.




| | No. of TCs | Planned | Executed | Passed | Failed | Blocked |
|---|---|---|---|---|---|---|
| tp-01 | Absolute | 25 | 7 | 0 | 7 | 0 |
| | Percentage | 100% | 28.0% | 0.0% | 100.0% | 0.0% |
| tp-02 | Absolute | 15 | 6 | 5 | 1 | 0 |
| | Percentage | 100% | 40.0% | 83.3333333333% | 16.6666666667% | 0.0% |
| tp-03 | Absolute | 9 | 2 | 2 | 0 | 0 |
| | Percentage | 100% | 22.2222222222% | 100.0% | 0.0% | 0.0% |
| tp-04 | Absolute | 12 | 5 | 1 | 4 | 0 |
| | Percentage | 100% | 41.6666666667% | 20.0% | 80.0% | 0.0% |
| tp-05 | Absolute | 13 | 9 | 8 | 1 | 0 |
| | Percentage | 100% | 69.2307692308% | 88.8888888889% | 11.1111111111% | 0.0% |
| tp-06 | Absolute | 10 | 0 | 0 | 0 | 0 |
| | Percentage | 100% | 0.0% | 0.0% | 0.0% | 0.0% |

Slika 4.22: Sažetak izvještaja po ispitnim paketima

Ovaj izvještaj je kreiran korištenjem izrađenog programa za obradu dnevnika izvođenja nastalih na pokretaču Lusca sustava. Koriste se podatci o prolaznosti pojedinog ispitnog slučaja koji su grupirani po prefiksu njihovih identifikatora. Odabirom poveznice na grupu ispitnih slučaja dobiva se uvid u podatke o prolaznosti pojedinog slučaja prikazanom na slici 4.23.




Slika 4.23: Sažetak izvještaja po ispitnim slučajevima

Prikazana je lista ispitnih slučajeva s oznakama o njihovoj prolaznosti i sažetku o prolaznosti grupacije ispitnih slučajeva.

Nakon svakog izvođenja ispitivanja pokreće se izrađeni program kojemu se kao ulazni parametar predaje datoteka preuzeta s pokretača sustava što sadrži sve nastale datoteke i dnevnike ispitivanja. Dobiveno izvješće se pohranjuje s datumom kreiranja i služi kao jedan zapis u dnevniku izvođenja ispitivanja.

# 5. Zaključak

Zadatak radnje je izrada sustava koji bi trebao pružiti sredstva za automatizaciju procesa ispitivanja mrežnih aplikacija. Istražene su metodologije ispitivanja aplikacija te je odabrana metoda ispitivanja aplikacije kao crne kutije zbog njene praktičnosti i širokog područja primjene. Pri izradi sustava posao je podijeljen na dva osnovna dijela: upravljanje datotečnim resursima i izvođenje pravila ispitivanja. Oba zadatka trebalo je ispuniti u distribuiranoj okolini kao što je računalna mreža.

Kreiran je sustav za ispitivanje čiji se rad zasniva na izvođenju pravila zadanih opisom ispitnog paketa. Upravljanje datotečnim resursima raspoređenim po računalnoj mreži je izvedeno na način da se prije ispitivanja datoteke prenose na ciljna računala u ispitnoj okolini, tijekom ispitivanja se bilježe informacije o nastalim datotekama, a nakon ispitivanja datoteke se sakupljaju na centralni element sustava. Za upravljanje tokom izvođenja ispitnog slučaja koristi se programska paradigma upravljanja događajima. Arhitektura sustava je centralizirana struktura s programom koji upravlja ispitivanjem (upravljač sustava) i programom za izvođenje zadataka (agent sustava). Komunikacija unutar sustava je ostvarena korištenjem SOAP poziva, a prenošenje datoteka se obavlja putem HTTP protokola. Implementacija je izvedena u programskom jeziku Python pri čemu je korištena programska biblioteka Twisted za asinkrono programiranje i izradu mrežnih servisa.

Lusca sustav se može svrstati u grupaciju nisko specijaliziranih sustava za ispitivanje aplikacija, pošto funkcionalnosti koje pruža ne ograničavaju korisnika na ispitivanje točno određenih aplikacija niti je specijaliziran za neku posebnu vrstu aplikacije. Velik je trud uložen u olakšanje definiranja ispitnog procesa što bi implicitno trebalo pojednostavniti izradu ispitnih slučajeva do zadovoljavajuće razine. Kao posljedica niske razine specijalizacije, Lusca sustav ostavlja dobar dio posla pri kreiranju ispitnih paketa na korisniku, jer ne pruža dodatne alate za automatizaciju procesa izrade ispitnog paketa. Nakon izrade ispitnog paketa sustav pruža potpunu automatizaciju ispitnog procesa na čemu je i bio naglasak pri izradi. Jednom kreirani ispitni paketi se mogu bilo kad pokrenuti *jednim klikom miša* nakon čega nema potrebe za korisničkom intervencijom da bi se ispitivanje sprovelo do kraja. Sa strane upravljanja datotekama potrebnih za ispitivanje i nastalih datoteka, rješenje je izvedeno po principu – jedna ulazna datoteka, jedna izlazna datoteka koju korisnik preuzima s pokretača sustava; sve između te dvije datoteke je na sustavu da obradi.

Sustav je tijekom razvoja primijenjen za ispitivanje *ikev2* mrežne aplikacije – implementacije IKEv2 protokola (eng. *Internet Key Exchange version 2*) [18] za Linux operacijski sustav. Kroz proces ispitivanja *ikev2* aplikacije iskristalizirale su se određene potrebe kako bi se olakšao rad sa sustavom. Potrebno je bilo izraditi dodatni program za izradu izvještaja kako bi razvojni tim imao uvid u proces ispitivanja i informacije o prolaznosti ispitnih paketa. Izrađeni program je pokretan nakon svakog ispitivanja s novom inačicom izvornog kôda, a proizveo bi skup HTML datoteka s pregledom prolaznosti pojedinog ispitnog slučaja i ostalim detaljima. Kreirani izvještaj se pohranjuje na poslužitelj dostupan članovima razvojnog tima te je na taj način ostvarena komunikacija razvojnog tima s ispitnim timom.

Zaključenjem ovog diplomskog rada završava i prva faza projekta Lusca u kojoj su zamišljene ideje sprovedene u praksu i stečene spoznaje o dobrim i lošim stranama

kreiranog sustava. Među dobrim stranama sustava je kvalitetan jezik za definiciju ispitnih paketa gdje je uočena potreba za manjim doradama na području opisa resursa korištenih u ispitivanju. Dobra strana sustava su i podjela komunikacijskih protokola na jasno odvojene dijelove korištene za određenu svrhu s mogućnosti za jednostavnu nadogradnju. Izvedena je i dobra podloga za upravljanje datotečnim resursima, no primijećena je i slabost u procesu prenošenja datoteka koja se manifestira prilikom nadogradnje podatkovnih protokola. Primjerice za svako prenošenje datoteke je potrebno dodati metode za slijed akcija *pripremi-spreman-preuzimanje-preuzeo* s obe strane u komunikaciji kao i metode za oporavak od greške. Ovaj i slični problemi bi se mogli riješiti boljim dizajnom modula za prijenos podataka ili kreiranjem posebnog protokola koji koristi spojnu uslugu što se izbjegavalo u ovoj fazi projekta.

Za dodatno proširenje sustava u idućim fazama razvoja je ostavljeno poprilično mjesta. Pri razmatranju nekih ideja za poboljšanje sustava mnoge ideje nisu do kraja razmatrane jer ne postoji ugrađena funkcionalnost za autentifikaciju i upravljanje korisnicima sustava. Trenutno sustav ne razlikuje pristigle komande s korisničke konzole po njihovom izvoru, pa je time sustav trenutno ograničen. Razmotrimo li scenarij uporabe sustava u kojemu dva korisnika žele pokrenuti isti ispitni paket, no s različitim inačicama izvornog kôda, iako postoji potreban broj računala za dvije istovremene instance izvođenja ispitnog paketa, drugi korisnik mora pričekati završetak prvog ispitivanja kako bi on došao na red. Taj korisnik može doduše pokrenuti neki drugi ispitni paket ukoliko ima dostupnih računala. Dodavanjem kombinacije modula za autentifikaciju i autorizaciju mogla bi se ostvariti funkcionalnost sustava da se ponaša kao pravi servis za ispitivanje u kojem bi administrator određenim korisnicima mogao dodijeliti prava za ispitivanje na računalnim resursima. Korisnici bi pristupili sustavu, postavili ispitne pakete, te unaprijed zakazali izvođenje ispitivanja u vremenskim intervalima dodatnim modulom sustava. Tada bi dobro došao i modul za dojavu o završetku ispitivanja putem elektroničke pošte ili SMS usluge (eng. *Short Message Service*).

Rad na projektu je bio interesantan zbog primjene mnoštva novih pristupa u dizajniranju i izradi programskih alata kao što je paradigma upravljanja događajima i metoda asinkronog programiranja, čija je implementacija bila relativno jednostavna zbog mogućnosti programskog jezika Python i programske biblioteke Twisted. Nastavak rada na projektu je planiran.

Jure Rastić

_____

# 6. Literatura


1. Software bug - Wikipedia, the free encyclopedia.
   URL: http://en.wikipedia.org/wiki/Computer_bugs, (07/11/07).
2. USS Vincennes (CG-49) – Wikipedia, the free encyclopedia.
   URL: http://en.wikipedia.org/wiki/USS_Vincennes_(CG-49), (07/11/07).
3. Mars Climate Orbiter - Wikipedia, the free encyclopedia.
   URL: http://en.wikipedia.org/wiki/Mars_Climate_Orbiter, (07/11/07).
4. Introduction to Software Testing.
   URL: http://www.onestoptesting.com/introduction/, (07/11/07).
5. Nilesh Parekh, Software Testing - White Box Testing Strategy.
   URL: http://www.buzzle.com/editorials/4-10-2005-68350.asp, (07/11/07).
6. Nilesh Parekh, Software Testing - Black Box Testing Strategy.
   URL: http://www.buzzle.com/editorials/4-10-2005-68349.asp, (07/11/07).
7. Kerry Zallar, Practical Experience in Automated Testing.
   URL: http://www.methodsandtools.com/archive/archive.php?id=33, (07/11/07).
8. Event-driven programming - Wikipedia, the free encyclopedia.
   URL: http://en.wikipedia.org/wiki/Event_driven_programming, (07/11/07).
9. Python Programming Language.
   URL: http://www.python.org/, (07/11/07).
10. SOAP Specifications.
    URL: http://www.w3.org/TR/soap/, (07/11/07).
11. Twisted – Trac.
    URL: http://twistedmatrix.com/, (07/11/07).
12. JSON.
    URL: http://json.org/, (07/11/07).
13. subversion.tigris.org - Version control system.
    URL: http://subversion.tigris.org/, (07/11/07).
14. GNU build system - Wikipedia, the free encyclopedia.
    URL: http://en.wikipedia.org/wiki/GNU_build_system, (07/11/07).
15. Jure Rastić, Ispitno okruženje - Definicija XML-a i parser / seminarski rad. FER, Sveučilište u Zagrebu, 2006.
16. Hrvoje Slaviček, Sustav za uređivanje testova temeljen na razvojnom okruženju Eclipse / diplomski rad br. 1673. FER, Sveučilište u Zagrebu, 20.09.2007.
17. Dražen Nežić, Sučelje za uporabu Lusca sustava iz Eclipse razvojnog okruženja / diplomski rad br. 1672. FER, Sveučilište u Zagrebu, 20.09.2007.
18. IKEv2 Project.
    URL: http://ikev2.zemris.fer.hr/, (07/11/07).