SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 1665

ISPITIVANJE SUSTAVA ZASNOVANOG NA USLUGAMA ISPITNIM MODELOM

Jakov Krolo

Zagreb, lipanj 2007.

Zahvaljujem se mentoru Prof. dr. sc. Siniši Srbljiću što mi je omogućio rad na ovom zanimljivom projektu te na vrijednim savjetima i prenesenom iskustvu.

Zahvaljujem se i mr.sc Ivanu Bencu na brojnim korisnim savjetima i uputama prilikom izrade ovog diplomskog rada.

Posebno se zahvaljujem svojoj obitelji na razumijevanju i pomoći koju su mi pružili tijekom studiranja.

Sadržaj

1. Uvod

Programski sustavi šire se u gotovo svim granama našega društva te život postaje iznimno ovisan o njima. Zbog toga kakvoća programske potpore postaje sve važnija. Povećana složenost sustava, kao i sve kraći rokovi isporuke čine postupak ispitivanja programskih sustava sve izazovnijim. Jedan od ključnih problema tradicionalnog ispitivanja sustava je kasno uključivanje procesa ispitivanja u proces izrade programskog sustava. Kasno otkrivanje pogrešaka povećava troškove uzrokovane ispravljanjem i ponovnim ispitivanjem sustava koji mogu doseći i polovicu ukupnih troškova izrade sustava. Dodatni problem je ručno provođenje ispitivanja sustava. Naime, velika složenost današnjih programskih sustava rezultira s iznimno mnogo kombinacija ulaza i događaja koji rezultiraju različitim izlazima sustava. Korištenjem ručnog provođenja ispitivanja sustava nije moguće pokriti sve kombinacije ulaza i ulaznih događaja te tako potvrditi ispravnost i kakvoću sustava.

Ispitivanje složenih programskih sustava po razumnoj cijeni i u razumnom vremenu nije moguće ostvariti bez pomoći automatiziranih alata za ispitivanje sustava. Danas su raširenі komercijalni alati koji podržavaju djelomično automatizirano ispitivanje sustava koje zahtjeva znatan ljudski rad. Djelomično automatizirano ispitivanje označava mogućnost automatiziranog izvođenja ispita, ali ne i mogućnost automatiziranog stvaranja ispita. Međutim, zbog povećane složenosti sustava i sve viših zahtjeva za kakvoćom programskih sustava stvaranje ispita postaje sve složenije te korištenje djelomično automatiziranih alata više nije dovoljno za intenzivno ispitivanje kakvoće programskih sustava. Jedna od novih tehnologija koja taj problem pokušava umanjiti jest *ispitivanje zasnovano na modelu* (engl. *Model Based Testing*, skraćeno MBT).

Ispitivanje zasnovano na modelu koristi model koji opisuje željeno ponašanje programskog sustava. Model ispitivanog sustava služi za automatizirano stvaranje velikih količina ispitnih slučajeva koji se koriste za provedbu ispitivanja programskog sustava. Glavna prednost *ispitivanja zasnovanog na modelu* je potpuna automatizacija ispitivanja. Potpuna automatizacija znači da se pored automatizacije izvođenja ispitnih slučajeva, automatizira i postupak stvaranja ispitnih slučajeva. Dodatno, važna prednost *ispitivanja zasnovanog na modelu* je rano uključivanje ispitivanja u cjelokupni proces izrade sustava. Proces ispitivanja uključuje se u cjelokupni proces izrade sustava odmah tijekom definiranja zahtjeva sustava,

što omogućava bolje razumijevanje zahtjeva i čini ispitivanje učinkovitijim. Statistike pokazuju kako se veliki dio grešaka pojavljuje upravo zbog loše definiranih, nejasnih i nejednoznačnih zahtjeva postavljenih programskim sustavima.

Iz industrijske perspektive, *ispitivanje zasnovano na modelu* obećavajuća je tehnika poboljšanja kakvoće i učinkovitosti ispitivanja te smanjenja troškova. Cilj *ispitivanja zasnovanog na modelu* je rješavanje problema automatskog stvaranja sljedova ispitnih slučajeva na osnovu modela sustava, nadopunjavajući tako automatsko izvođenje ispita. Iz akademske perspektive, *ispitivanje zasnovano na modelu* prirodno je proširenje formalnih metoda i verifikacijskih tehnika. Formalna verifikacija nastoji pokazati da sustav ima određena svojstva dokazujući da model tog sustava zadovoljava ta svojstva. S druge strane, *ispitivanje zasnovano na modelu* započinje s verificiranim modelom te potom pokušava pokazati da se programsko ostvarenje sustava ponaša u skladu s verificiranim modelom.

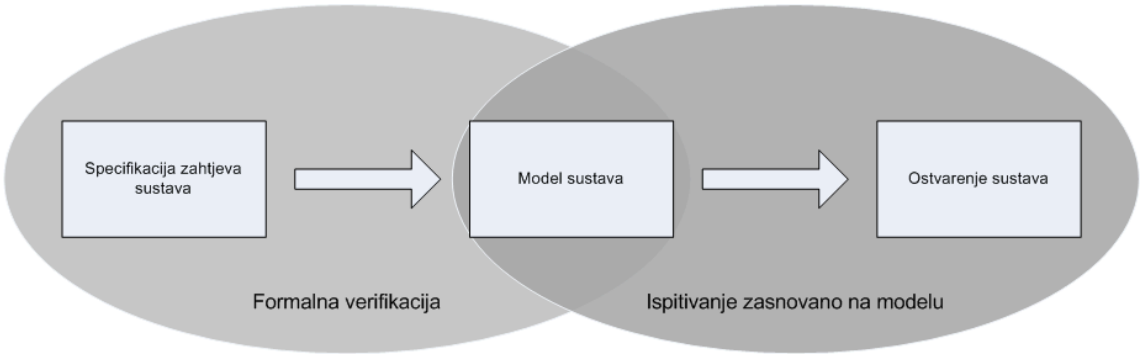
Diplomski rad opisuje tehnike ispitivanja programskih sustava, a posebno razmatra *ispitivanje zasnovano na modelu* kao najnoviju generaciju tehnika ispitivanja. Drugo poglavlje opisuje evoluciju ispitivanja programskih sustava te iznosi razradbu pristupa ispitivanju i tehnika ispitivanja. Treće poglavlje iznosi razradbu različitih pristupa *ispitivanju zasnovanom na modelu*. Na osnovu iznesene razradbe pristupa *ispitivanju zasnovanom na modelu*, opisuju se najčešće korišteni alati *ispitivanja zasnovanog na modelu*. Posebna pažnja posvećuje se alatu CTG (engl. *Conformiq Test Generator*), korištenom tijekom izrade praktičnog dijela diplomskog rada. Također se detaljno opisuje cjeloviti proces *ispitivanja zasnovanog na modelu.* Četvrto poglavlje sadrži opis izvedbe *ispitivanja zasnovanog na modelu* na primjeru jednostavne korisnik-poslužitelj okoline. Peto poglavlje opisuje primjenu *ispitivanja zasnovanog na modelu* u ispitivanju rada sigurnosnog podsustava *Programirljive Internet okoline* (engl. *Programmable Internet Environment*, skraćeno PIE). Šesto poglavlje iznosi zaključak diplomskog rada.

2. Ispitivanje programskih sustava

Svrha ispitivanja programskog sustava je utvrđivanje da li ispitivani programski sustav sadrži greške. Standard IEEE za terminologiju ispitivanja programskih sustava definira grešku (engl. *error*) kao neispravno ponašanje uzrokovano neispravnošću programske komponente [50]. Glavni razlozi ispitivanja sustava ekonomske su prirode. Naime, troškovi neispravnosti sustava (koji sežu od gubitka ugleda sustava do gubitka ljudskih života) te troškovi popravka sustava (koji zahtijevaju izdavanje zakrpa ili novih verzija programa) preveliki su da bi se zanemarili. Na primjer, godine 1998. centar za svemirske programe NASA (engl. *National Aeronautics and Space Administration*) poslao je na Mars svemirsku letjelicu *Mars Climate Orbiter* sa ciljem proučavanja vremena i klime na Marsu. Prilikom ulaska u orbitu Marsa dogodila se nesreća. Sudarom o površinu planeta letjelica se uništila. Razlog nesreće je bila greška programske potpore u pretvorbi jedinica engleskog mjernog sustava u jedinice međunarodnog mjernog sustava (SI jedinice). Procijenjena šteta iznosi 165 milijardi američkih dolara.

Dva najzapaženija pristupa za postizanje visoke kakvoće programske potpore su formalna verifikacija i ispitivanje. Formalna verifikacija provjerava da li formalni model sustava odgovara formalnoj specifikaciji sustava. Dakle, formalna verifikacija garantira samo ispravnu relaciju između modela i specifikacije, a ne i ispravnu relaciju između izvedbe sustava i specifikacije. Glavni nedostatak formalne verifikacija je nemogućnost primjene u praksi na velike sustave [41]. Pored toga, formalna verifikacija zasniva se na pretpostavkama koje ne moraju nužno vrijediti u okolini u kojoj se programski sustav planira koristiti. S druge strane, ispitivanje nastoji utvrditi ispravnost relacije između izvedbe sustava i specifikacije sustava. Ispitivanje je vremenski manje zahtjevno te bolje otkriva kako se programski sustav ponašati u stvarnoj okolini. Koristeći ljudska i računalna sredstva te programske alate za ispitivanje, strategijom ispitivanja smanjuju se ukupni troškovi projekta. Razvojem *ispitivanja zasnovanog na modelu* pokazalo se da formalna verifikacija i ispitivanje sustava modelom nemaju iste, već komplementarne ciljeve. Slika 2.1 uspoređuje ciljeve formalne verifikacije i *ispitivanja zasnovanog na modelu*. Formalna verifikacija koristi model izgrađen na osnovu specifikacija zahtjeva sustava. Cilj formalne verifikacije je pokazati da model sustav zadovoljava specificirana svojstva. S druge strane, *ispitivanje zasnovano na modelu* koristi

verificirani model te potom pokazuje da se izvedba programskog sustava u stvarnoj okolini ponaša u skladu sa modelom.



Slika 2.1 Povezanost formalne verifikacije i *ispitivanja zasnovanog na modelu*

U nastavku poglavlja opisuje se evolucija koncepta ispitivanja programskih sustava kroz povijest. Ispitivanje se evolucijski dijeli na pet razdoblja koji se razlikuju po definiciji i cilju ispitivanja. Potom se uspoređuju dva osnovna pristupa ispitivanju, *pristup bijele kutije* (engl. *white box approach*) i *pristup crne kutije* (engl. *black box approach*). Na kraju se iznosi razradba tehnika ispitivanja. Opisuje se ručna tehnika ispitivanja te četiri generacije automatiziranog ispitivanja sustava.

2.1. Evolucija ciljeva ispitivanja programskih sustava

Ispitivanje programske potpore prisutno je od samih početaka razvoja programske potpore. Koncept ispitivanja s vremenom prolazi kroz razdoblja evolucije. Evolucija definicije i ciljeva ispitivanja programskih sustava usmjeravala je i istraživanje o tehnikama ispitivanja. Razvoj tehnika ispitivanja dijeli se u pet razdoblja, ovisno o definiciji i ciljevima ispitivanja. Prvo razdoblje ispitivanja usmjereno je na *traženje grešaka* (engl. *debugging*), što se danas više ne smatra pravim ispitivanjem. Drugo razdoblje ispitivanje usmjereno je na pokazivanje ispravnosti sustava, tj. stvaranje ispita koji pokazuju da se sustav ponaša u skladu sa zahtjevima. Treće razdoblje ispitivanja usmjereno je na otkrivanje grešaka, a fokusira se na stvaranje ispita koji otkrivaju neispravnost sustava. Četvrto razdoblje ispitivanja usmjereno je na vrednovanje sustava tijekom ciklusa izrade programskog sustava. Konačno, peto razdoblje usmjereno je na sprječavanje pojave grešaka.

2.1.1. Razdoblje usmjereno traženju grešaka

Godine 1950. Turing objavljuje članak [49] koji se smatra prvim člankom o ispitivanju programa. U članku se raspravlja o pitanju „*Kako znamo da program postupa inteligentno?*“. Ako se program gradi na osnovu zahtjeva, tada je to pitanje istovjetno pitanju „*Kako znamo da program zadovoljava postavljene zahtjeve?*“. Turingovo ispitivanje nastoji ustvrditi inteligenciju programa pokazujući da nema razlike između ponašanja programa i čovjeka kao referentnog sustava. Pritom se ispitivani program gradi na osnovu specifikacija zahtjeva koji opisuju ponašanje čovjeka. Takvo Turingovo ispitivanje smatra se početnim oblikom funkcijskog ispitivanja. U tom početnom razdoblju pod ispitivanjem se smatrao sljedeći postupak: nakon što bi se otkrila greška tijekom korištenja sustava, razvijatelji sustava pretraživali su programski kod kako bi našli i ispravili grešku. Prema tome, koncepti ispitivanja i *traženja grešaka* nisu jasno razlikovani tijekom prvog razdoblja.

2.1.2. Razdoblje usmjereno pokazivanju ispravnosti

Drugo razdoblje traje od 1957. do 1978. godine. Tijekom *razdoblja usmjerenog pokazivanju ispravnosti*, na ispitivanje se gledalo kao na tehniku koja pokazuje da programski sustav zadovoljava svoje zahtjeve. Godine 1957. Charles Baker naglašava da ispitivanje ima dva cilja: osigurati da program radi i osigurati da program rješava zadani problem. Od tada se osiguravanje da program rješava zadani problem smatra fokusom ispitivanja, pri čemu se pod osiguravanjem podrazumijeva zadovoljavanje zahtjeva programskog sustava. U tom razdoblju jasno su odvojeni koncepti traženja pogrešaka i ispitivanja. Definicija ispitivanja jasno naglašava da je svrha ispitivanja pokazivanje ispravnosti programa. Tijekom sedamdesetih godina 20. stoljeća, široko se rasprostranjuje ideja o iscrpnom ispitivanju programske potpore. Takva ideja dovodi do početka istraživanja o pokrivenosti (engl. *coverage*) putanja ispitivanja [43].

2.1.3. Razdoblje usmjereno otkrivanju grešaka

Treće razdoblje traje od 1979. do 1982. godine. Tijekom trećeg razdoblja na ispitivanje se gledalo kao na tehniku za otkrivanje grešaka u ostvarenju programskog sustava. Godine 1979. Myers izdaje knjigu *The Art of Software Testing* [45], koja daje osnove za učinkovitije oblikovanje tehnika ispitivanja. Ispitivanje programske potpore po prvi put se opisuje kao proces izvođenja programa s namjerom otkrivanja grešaka. Glavni pomak postiže se

pridjeljivanjem većeg značaja ispitnim slučajevima koji bi pronašli greške sustava. U *razdoblju usmjerenom pokazivanju ispravnosti* često su odabirani ispitni slučajevi koji imaju male vjerojatnosti uzrokovanja grešaka u programskom sustavu. Međutim, u *razdoblju usmjerenom otkrivanju grešaka* ispitivanje želi pokazati da programski sustav ima greške. Takvo razmišljanje dovodi do odabira ispitnih slučajeva koji imaju veću vjerojatnost pronalaska grešaka, što ispitivanje čini učinkovitijim.

2.1.4. Razdoblje usmjereno vrednovanju

Razdoblje usmjereno vrednovanju (engl. *evaluation-oriented*) četvrto je razdoblje koje vremenski traje od 1983. do 1987. godine. Glavni pomak u četvrtom razdoblju ostvaruje se shvaćanjem ispitivanja programske potpore kao tehnike za otkrivanje grešaka, no ne samo u zahtjevima sustava, već i u oblikovanju te ostvarenju programskog sustava. Godine 1983. institut *The Institute for Computer Sciences and Technology of the National Bureau of Standards* izdaje smjernice *Guideline for Lifecycle Validation, Verification, and Testing of Computer Software*. U tim smjernicama opisuje se metodologija koja u jednu cjelinu okuplja verifikacijske, validacijske te tehnike ispitivanja za omogućavanje vrednovanja sustava tijekom životnog ciklusa izrade sustava. Na taj način osigurava se razvoj i održavanje programskog sustava veće kakvoće.

2.1.5. Razdoblje usmjereno sprječavanju pojave grešaka

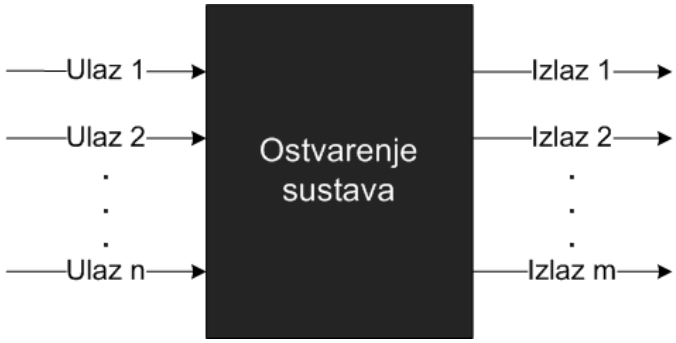
Posljednje razdoblje evolucije ispitivanja programske potpore traje od 1988. godine do danas. Tijekom *razdoblja usmjerenog sprječavanju pojave grešaka*, ispitivanje se smatra tehnikom za sprječavanje pojave grešaka u zahtjevima, oblikovanju i ostvarenju programskog sustava. U vrlo značajnoj knjizi *Software Testing Techniques* [44], Beizer tvrdi da je postupak oblikovanja ispita jedan od najučinkovitijih postupaka sprječavanja grešaka. Definicija ispitivanja proširuje se, pored aktivnosti otkrivanja grešaka, i na aktivnosti sprječavanja pojave grešaka. Godine 1991. Hetzel iznosi sljedeću definiciju ispitivanja: *Ispitivanje je planiranje, oblikovanje, stvaranje, održavanje i izvođenje ispita i ispitnih okolina*. Takva razmišljanja naglašavaju važnost ranog oblikovanja ispita u ciklusu izrade sustava.

2.2. Razradba pristupa ispitivanju

Ispitivanje programskih sustava široko je područje računarske znanosti. Razvijeno je mnogo metoda ispitivanja, no sve se one dijele u dva skupa prema korištenom pristupu ispitivanju: *pristup crne kutije* i *pristup bijele kutije*. Svaki od pristupa ima svoje prednosti i nedostatke koji se opisuju u nastavku odjeljka. Ne postoji općenito rješenje pitanja koji pristup je bolji, već odabir pristupa treba zasnivati na nekoliko faktora, npr. raspoloživo vrijeme, dostupnost unutrašnje strukture sustava, svrha ispitivanja.

2.2.1. Pristup crne kutije

Pristup crne kutije kod ispitivanja programskog sustava fokusira se samo na ulaze i izlaze sustava, uz razumijevanje ponašanja sustava ili funkcijskih svojstava koji se izvode isključivo iz zahtjeva sustava (slika 2.2). Oblikovanje ispita ovisi o promatranju tih svojstava, pritom potpuno zanemarujući strukturu ostvarenja programskog sustava.



Slika 2.2 Pristup crne kutije

Glavna prednost ispitivanja *pristupom crne kutije* je mogućnost ispitivanja sustava od osobe ili tima koji ne zna ništa o tome kako je sustav ostvaren. Konačno, svrha sustava je ponašanje u skladu sa zahtjevima, što je po definiciji neovisno o tome kako je sustav ostvaren. Još jedna prednost ispitivanja *pristupom crne kutije* (poznato i kao *ponašajno ispitivanje*) je dinamička analiza ispitivanog sustava.

U teoriji, alati za ispitivanje *pristupom crne kutije* izvode programski sustav sa svim mogućim kombinacijama ulaza. Međutim, zbog iznimno velikog broja mogućih kombinacija ulaza, u praksi je takvo iscrpno ispitivanje nemoguće kod složenih sustava. Iz navedenog razloga, Myers [45] tvrdi da ispitivanje *pristupom crne kutije* ne može utvrditi je li program bez greške. Kao još jednu od slabosti *pristupa crne kutije*, Howden [46] navodi zanemarivanje

važnih funkcijskih svojstava programa koji su dio oblikovanja ili ostvarenja programskog sustava, a nisu opisani u zahtjevima sustava.

2.2.2. Pristup bijele kutije

Ispitivanje zasnovano na *pristupu bijele kutije* (poznato i kao *strukturno ispitivanje*) upravljano je unutrašnjom strukturom programa. Ispitivanje *pristupom bijele kutije* pri stvaranju ispita uzima u obzir unutrašnja funkcijska svojstva programa. Ispiti stvoreni na osnovu *pristupa bijele kutije* uključuju pokrivenost naredbi, programskih grananja, toka podataka, unutrašnje logike programa, itd.

Prednost *pristupa bijele kutije* je poznavanje unutrašnje strukture programa, što pomaže lakšem uočavanju tipova ispitnih podataka koji rezultiraju učinkovitijim ispitivanjem programskog sustava. Posljedica poznavanja unutrašnje strukture programa je i popravljanje kakvoće programskog koda (čime se također mogu ukloniti neke skrivene greške). Naime, pretraživanjem programskog koda inženjeri ispitivači često savjetuju razvijatelje programa na brisanje nepotrebnog ili promjenu loše napisanog programskog koda.

Međutim, gotovo je nemoguće pretraživanje svakog dijela programskog koda gdje se mogu nalaziti greške. Inženjer ispitivač mora poznavati kako je programski kod napisan kako bi mogao oblikovati prikladne ispite. Još jedan nedostatak ovog pristupa je krivo tumačenje zahtjeva. Na osnovi krivog tumačenja zahtjeva sustava programski se ostvaruje kriva funkcionalnost. Takav propust razvijatelja programa ne može se otkriti ispitivanjem zasnovanom na *pristupu bijele kutije*. Naime, inženjer ispitivač radi ispite na osnovu programskog koda tako da ne zna je li dio funkcionalnosti ispravno ostvaren.

Iako je ispitivanje zasnovano na *pristupu bijele kutije* kroz prošlost bilo raširenije, *pristup crne kutije* polako stječe popularnost u posljednjih desetak godina, a ponajviše zbog razvoja *ispitivanja zasnovanog na modelu.*

2.3. Razradba tehnika ispitivanja

Na razradbu tehnika ispitivanja znatan utjecaj ima evolucija definicije i ciljeva ispitivanja. U početku razvoja ispitivanja postojalo je samo ručno ispitivanje. Pojavom automatiziranih tehnika, ručna ispitivanja gube na značaju. Kako ručna ispitivanja ne mogu pratiti trend povećane složenosti programskih sustava, sve su raširenije automatizirane tehnike. Automatizirane tehnike opisane u nastavku odjeljka prate evoluciju ispitivanja. Dakle, prvo se

razvila tehnika *snimi/ponovi* koja predstavlja prvu generaciju automatiziranih tehnika. Drugu generaciju ispitnih tehnika predstavlja tehnika *ispitivanja ispitnim skriptama*. *Ispitivanje apstrakcijom ispita* predstavlja treću generaciju, dok posljednju generaciju automatiziranog ispitivanja predstavlja tehnika *ispitivanja zasnovanog na modelu*.

Automatsko izvođenje ispita zasniva se na ispitnim skriptama (poznate i kao *ispitni pogonski programi*). Ispitne skripte su programi koji pružaju općenite mehanizme za izvođenje ispitnih slučajeva te spremanje odgovora ispitivanog sustava s ciljem naknadne analize.

2.3.1. Ručno ispitivanje

Iako je ručno ispitivanje najstarija tehnika ispitivanja, ono se i danas često koristi. Međutim, učestalost korištenje ručnog ispitivanja sustava ipak opada zbog nedostataka koje takovo ispitivanje ima u usporedbi s automatiziranim tehnikama. Glavni nedostatak ručnog ispitivanja je izrazito velika podložnost greškama, npr. inženjeri ispitivači vrlo često nenamjerno ponavljaju neke ispitne slučajeve, dok neke druge ostavljaju neispitanima. Osim što je podložno greškama, ručno ispitivanje zahtijeva i znatan rad inženjera ispitivača te troši mnogo vremena.

Tablica 2.1 sadrži aktivnosti ručnog ispitivanja koje uključuju znatan rad inženjera ispitivača.

Tablica 2.1 Aktivnosti ručnog ispitivanja

Aktivnost	Opis izvođenja
Oblikovanje ispita	Ručni odabir ispitnih slučajeva visoke kakvoće koji pokrivaju zahtjeve stavljene pred ispitivani sustav.
Izvođenje ispita	Ručni unos ispitnih slučajeva preko prikladnog korisničkog sučelja.
Pokrivenost ispitivanja	Ručno utvrđivanje pokrivenosti svih logički mogućih kombinacija ulaza, što zahtijeva visoku ljudsku stručnost i znatnu količinu vremena.
Analiza rezultata	Ručna analiza i utvrđivanje jednakosti dobivenog izlaza i očekivanog izlaza sustava.

S obzirom na navedene nedostatke, korištenje ručnih tehnika ispitivanja opada, posebno kod većih i složenijih programskih sustava.

2.3.2. Automatizirano ispitivanje snimi/ponovi

U okviru prve generacije automatizacije ispitivanja razvijeni su alati zasnovani na tehnici *snimi/ponovi*. Tehnika *snimi/ponovi* zasniva se na snimanju nizova operacija inženjera ispitivača. Ova tehnika ne podržava pravu automatizaciju ispitivanja jer ispitivač inženjer ručno oblikuje i pokreće ispitne slučajeve, slično ručnom ispitivanju. Međutim, jednom izveden slijed ispitivanja može se snimiti u ispitnu skriptu. To svojstvo predstavlja glavnu prednost u odnosu na ručno ispitivanje. Ispitna skripta je program koji pruža općenite mehanizme za izvođenje ispitnih slučajeva te spremanje odgovora ispitivanog sustava. Tehnika *snimi/ponovi* koristi jednostavan oblik ispitnih skripti. Jednostavan oblik skripti omogućuje uključivanje snimanja slijeda operacija nad grafičkim sučeljem ispitivanog programa, te ponovnog izvođenja snimljenog slijeda.

Jednom snimljena sjednica u obliku ispitne skripte kasnije se može ponovo pokrenuti po potrebi. U slučaju promjene funkcionalnosti sustava, mora se ponovo provesti ručno stvaranje i snimanje novog skupa ispitnih slučajeva. Takav postupak troši jako puno vremena, kao i tijekom prvog snimanja. Kako su danas rokovi izrade i isporuke sustava sve kraći zbog zahtjeva tržišta, a sustavi sve složeniji, tehnikom *snimi/ponovi* moguće je snimiti tek manji dio putanja u prostoru mogućih ispitnih slučajeva. Zbog toga je pokrivenost ispitivanja sustava tehnikom *snimi/ponovi* ograničena tek na manji dio funkcionalnosti sustava.

2.3.3. Automatizirano ispitivanje ispitnim skriptama

Tehnika automatiziranog ispitivanja ispitnim skriptama pruža potporu razvoja ispitnih skripti na skriptnoj razini. Nedostatak te tehnike je nužna potreba za dobrim vještinama razvijanja programa (skripti) inženjera ispitivača. Poput tehnike *snimi/ponovi*, ispitivanje tehnikom ispitnih skripti također zahtijeva ručno oblikovanje ispita, što je vremenski zahtjevno.

Ako se funkcionalnost sustava s vremenom mijenja, potrebno je ručno održavanje ispitnih skripti. Iako je taj postupak slabost ove tehnike u usporedbi sa novijim tehnikama automatiziranog ispitivanja, ipak se radi o napretku u odnosu na tehniku *snimi/ponovi*. Ručno održavanje skripti je, uz pretpostavljene dobre vještine razvijanja inženjera ispitivača,

jednostavnije i vremenski manje zahtjevno nego ponovno ručno oblikovanje novog skupa ispitnih slučajeva i njegovo snimanje.

Ispitne skripte pišu se standardnim programskim jezicima, kao što su VB, C, C++ , Java, Perl, specijaliziranim jezicima kao što su Tcl i Python, ili nekim drugim jezicima koje podržavaju alati za ispitivanje sustava.

Ispitne skripte ostvaruju ispitivanje sustava kroz nekoliko koraka:

1. inicijalizacija ispitivanog sustava
2. prolazak kroz petlju ispitnih slučajeva te za svaki pojedini slučaj
    a. inicijalizacija odredišta (opcionalno)
    b. inicijalizacija izlaza na vrijednost različitu od očekivane (ako je moguće)
    c. postavljanje ulaza
    d. izvođenje ispitivanog sustava sa postavljenim ulaznim podacima
    e. snimanje izlaza i spremanje rezultata kako bi kasnije bilo moguće uspoređivanje sa očekivanim izlazom

2.3.4. Automatizirano ispitivanje apstrakcijom ispita

Tehnika automatizacije ispitivanja apstrakcijom ispita čini treću generaciju u razvoju automatizacije ispitivanja. Ova tehnika proširuje tehniku ispitivanja *snimi/ponovi* i unapređuje tehniku ispitivanja ispitnim skriptama dodajući im novu dimenziju – apstrakciju ispita. Apstrakcija ispita omogućava inženjeru ispitivaču definiranje ispita na višoj razini apstrakcije od odgovarajuće ispitne skripte. Time se postupak ispitivanja sustava dijeli na dva tipa inženjera ispitivača. Jedna vrsta inženjera ispitivača opisuje zahtjeve sustava i oblikuje ispitne slučajeve na višoj razini apstrakcije koji pokrivaju sve zahtjeve sustava. Druga vrsta inženjera ispitivača mora posjedovati dobre vještine razvijanja programa kako bi definirao ispitne skripte na nižoj razini. Ispitne skripte povezuju se sa odgovarajućim ispitnim slučajevima preko apstrakcijskih mehanizama. Na primjer, takvom raspodjelom omogućeno je oblikovanje jednostavnih ispitnih slučajeva visoke razine koji trebaju ostvariti pristup podacima baze podataka. Takvi pojednostavljeni (apstraktni) ispitni slučajevi prevode se apstrakcijskim mehanizmima u ispitnu skriptu sa stvarnim pozivima pristupa bazi podataka.

Postupci apstrakcije ispita dijele se u četiri kategorije opisane tablicom 2.2. Tablica opisuje kategorije apstrakcija ovisno na čemu se zasnivaju te pruža opis rada apstrakcijskog mehanizma za pojedinu kategoriju apstrakcije.

Tablica 2.2 Kategorije apstrakcija

Kategorija apstrakcije	Opis rada apstrakcijskog mehanizma
Apstrakcija zasnovana na akcijskim riječima	Ispitni slučajevi definirani na apstraktnoj razini pretvaraju se u akcijske riječi sa pripadnim parametrima koje predstavljaju konkretne ispitne ulaze (npr. *Send(100)*).
Apstrakcija zasnovana na prozorima	Ispitni slučajevi definirani na apstraktnoj razini pretvaraju se u operacije nad grafičkim sučeljem (prozorom) ispitivanog programa koje predstavljaju konkretne ispitne ulaze.
Apstrakcija zasnovana na objektima	Ispitni slučajevi definirani na apstraktnoj razini pretvaraju se u atribute i pozive metoda objekata ispitivanog programskog sustava.
Apstrakcija zasnovana na razredima	Isti mehanizam kao kod apstrakcije zasnovane na objektima, samo što koristi istu ispitnu skriptu za sve objekte istog razreda.

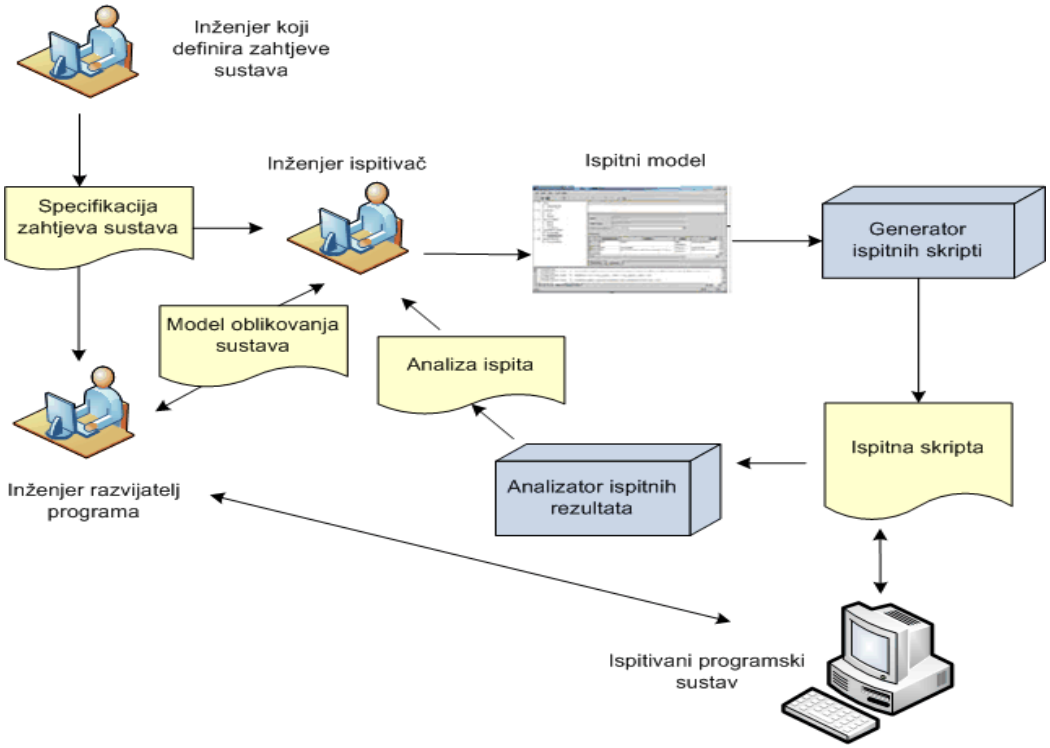
2.3.5. Automatizirano ispitivanje zasnovano na modelu

Prve dvije generacije automatiziranog ispitivanja fokusiraju se na izvođenje ispita te imaju ograničenu potporu za oblikovanje ispita. Tehnika treće generacije automatiziranog ispitivanja uvodi apstrakcijske mehanizme te odvaja definiranje ispitnog scenarija od automatizacije zasnovane na ispitnim skriptama, no još uvijek uz ograničenu potporu za automatiziranu izradu ispita. Tehnika automatiziranog *ispitivanja zasnovanog na modelu* smatra se četvrtom generacijom automatiziranog ispitivanja. *Ispitivanje zasnovano na modelu* pruža potpunu automatizaciju ispitivanja jer podržava automatizirano oblikovanje ispita i automatizirano izvođenje ispita.

Proces izgradnje programskog sustava uz *ispitivanje zasnovano na modelu* uključuje tri uloge:

1. inženjera koji definira zahtjeve sustava (engl. *requirements engineer*)
2. inženjera razvijatelja programa
3. inženjera ispitivača

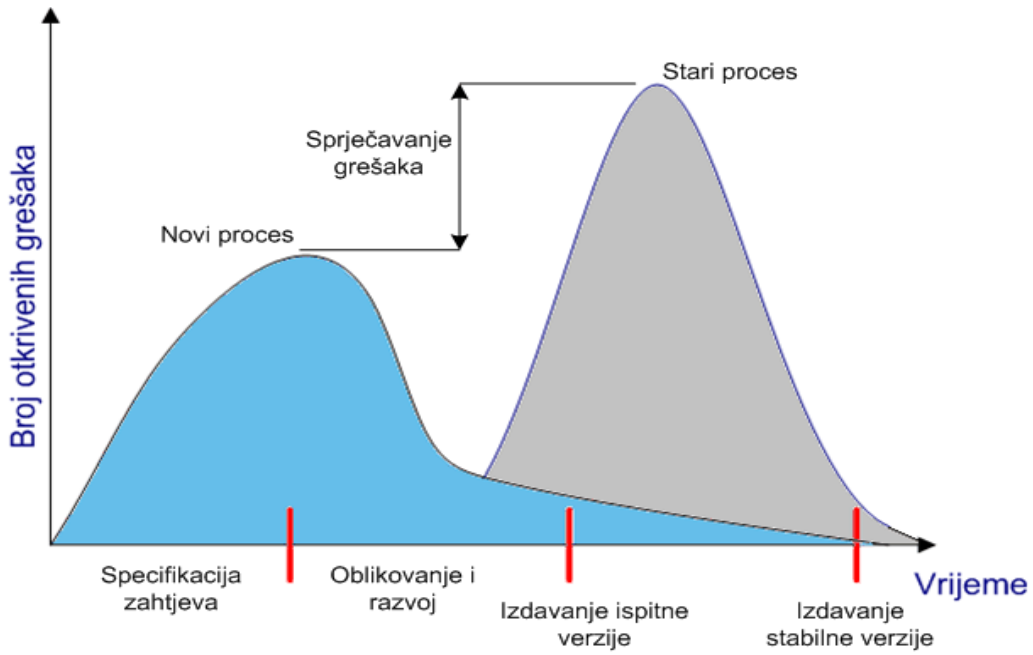
Inženjer koji definira zahtjeve sustava dokumentira zahtjeve sustava u tekstualnom obliku. Razvijatelj programa razvija arhitekturu sustava, komponente i sučelja. Iako je uobičajeno početi proces izgradnje sustava sa slabo definiranim zahtjevima, ulazi u proces mogu uključivati specifikacije zahtjeva sustava, korisničku dokumentaciju, dokumentaciju o upravljanju sučeljima, dokumentaciju o programskim sučeljima primjenskog sustava (engl. *application*), prethodne verzije oblikovanja te stare ispitne skripte. Inženjeri ispitivači koriste sve postojeće informacije kako bi zahtjeve sustava prikazali modelom.



Slika 2.3 Sažeti konceptualni proces *ispitivanja zasnovanog na modelu*

Slika 2.3 prikazuje pojednostavljeni proces *ispitivanja zasnovanog na modelu*. Na temelju specifikacija zahtjeva (koje gradi inženjer za definiranje zahtjeva sustava) te modela oblikovanja (engl. *design model*) (kojeg gradi inženjer razvijatelj programa u suradnji sa inženjerom ispitivačem) inženjer ispitivač gradi prikladan ispitni model. Ispitni model služi kao osnova za definiranje ispitnih slučajeva koji pokrivaju sve zahtjeve sustava. Ispitni model prevodi se u ispitnu skriptu za ispitivani sustav. Stvorene ispitne skripte upravljaju izvođenjem ispitivanja sustava. Nakon završetka izvođenja ispitnih slučajeva, analizator ispitnih rezultata automatizirano uspoređuje stvarne i očekivane izlazne vrijednosti te stvara izvještaj o rezultatima ispitivanja.

Kod ranijih tehnika ispitivanja greške se prepoznaju tek kasno u procesu izrade sustava, nekad čak i nakon isporuke sustava, a tada je ispravak grešaka najskuplji. Automatizacijom stvaranja ispita na osnovu modela, greške se otkrivaju brže, tj. ranije u ciklusu izrade sustava. Rano uključivanje *ispitivanja zasnovanog na modelu* u cjelokupni proces izrade sustava moguće je zbog nezavisnosti ispitivanja o programskom razvoju sustava. Prema tome, izgradnja modela na kojem se ispitivanje zasniva može početi odmah sa procesom specifikacije zahtjeva sustava. Kao što je prikazano na slici 2.4, kod *ispitivanja zasnovanog na modelu* mjera otkrivanja grešaka ima brzi rast već na samom početku cjelokupnog procesa izrade sustava, a potom se sve više smanjuje. Mnoge greške otkriju se u razdoblju specifikacije zahtjeva sustava, prije nego što proces krene u razdoblje razvijanja sustava. Greške su najčešće uzrokovane krivim specificiranjem ili krivim tumačenjem zahtjeva. Tijekom razdoblja specifikacije zahtjeva sustava, sprječavanje grešaka je najučinkovitije te su troškovi višestruko manji nego nakon razdoblja razvijanja sustava.



Slika 2.4 Smanjenje troškova *ispitivanjem zasnovanim na modelu*

Slika 2.4 predstavlja konceptualnu razliku između starijih tehnika ispitivanja i tehnike *ispitivanja zasnovane na modelu*. Tradicionalni proces otkrivanja i uklanjanja grešaka programskog sustava predstavljen je krivuljom *Stari proces*, dok je proces *ispitivanja zasnovanog na modelu* predstavljen krivuljom *Novi proces*.

Tehnika *ispitivanja zasnovanog na modelu* središnji je dio diplomskog rada te se detaljno opisuje u sljedećem poglavlju.

3. Ispitivanje zasnovano na modelu

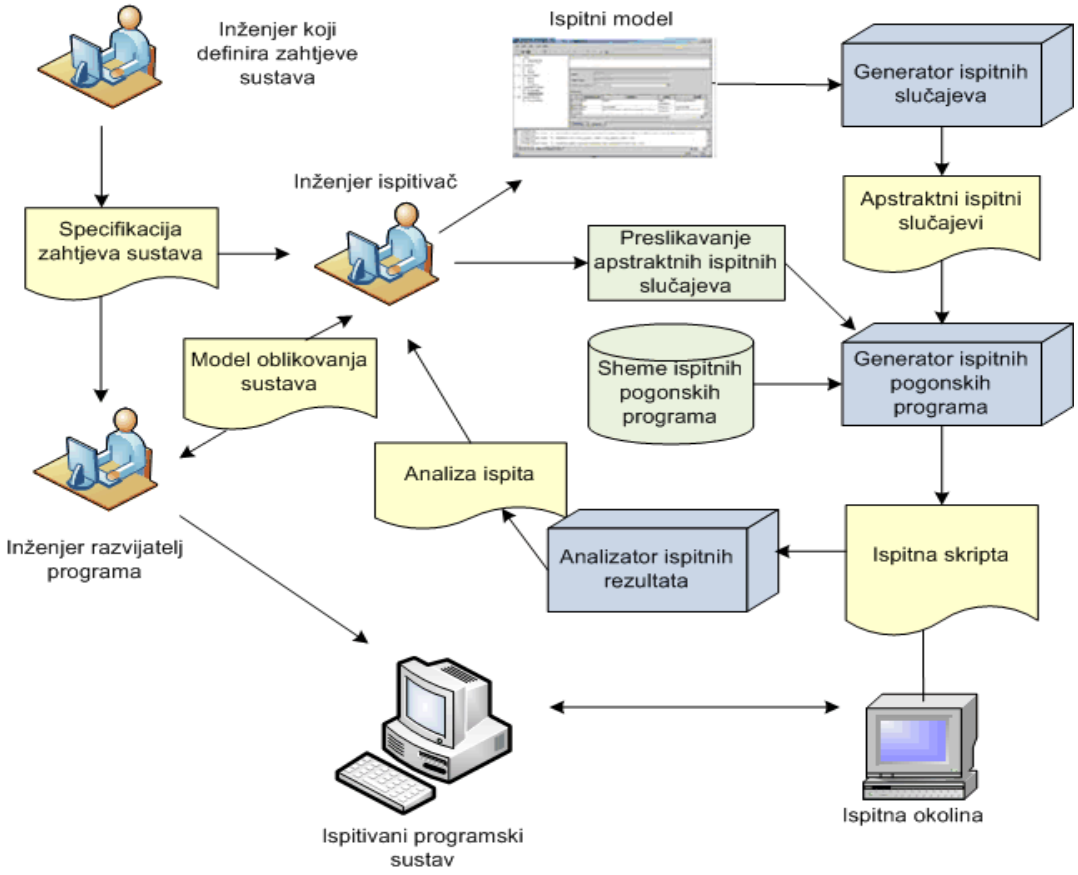
Ispitivanje zasnovano na modelu pruža potporu analizi grešaka u zahtjevima sustava te automatizaciji u oblikovanju i izvođenju ispita. Provjera modela osigurava da svojstva ispitivanog sustava nisu narušena. Model pomaže preradi nejasnih i loše napisanih zahtjeva sustava. Proces ispitivanja sustava modelom uključuje se rano u ciklusu izrade sustava. Takav pristup omogućuje uklanjanje grešaka zasnovanih na nejasno definiranim zahtjevima sustava prije nego započne razdoblje razvijanja programskog sustava. Uz takvu prednost te automatiziranje oblikovanja i izvođenja ispita, proces ispitivanja postaje učinkovitiji uz značajnu uštedu novca.

Slika 3.1 iznosi detaljniji pogled na tijek procesa modeliranja sa slike 2.3. Inženjer koji definira zahtjeve sustav te razvijatelji programskog sustava rade paralelno sa inženjerom ispitivačem kako bi definirali zahtjeve i model oblikovanja sustava. Takav pristup dovodi do prepoznavanja ispravnih sučelja komponenti programskog sustava rano u procesu te utječe na stabilnost arhitekture sustava. Prepoznavanje sučelja komponenti programskog sustava podrazumijeva prepoznavanje ulaza u sustav, u smislu odgovarajućih tipova ulaznih i izlaznih podataka.

Postoje dva zadatka inženjera ispitivača. Jedan zadatak je gradnja ispitnog modela, provedba analize modela (koristeći alate za automatizaciju ispitivanja) i ispravljanje bilo kakve nejednoznačnosti u zahtjevima. Provjereni (engl. *verified*) ispitni modeli prevode se u skup apstraktnih ispitnih slučajeva. Drugi zadatak inženjera ispitivača je definiranje preslikavanja apstraktnih ispitnih slučajeva na shemu ispitnog pogonskog programa za odgovarajuće ispitno okruženje. Shema ispitnog pogonskog programa općenito definira postupak smanjenja apstrakcije ispitnih slučajeva s ciljem da ispitivani sustav razumije ispitne slučajeve. Na osnovu postojećih shema ispitnih pogonskih programa i stvorenih ispitnih slučajeva, generator ispitnih pogonskih programa stvara ispitne skripte za ispitivani sustav. Stvorene ispitne skripte upravljaju izvođenjem ispitivanja sustava.

Sheme pogonskih programa postoje već za mnoge jezike i programe kao što su C, C++, VB, Java, Perl, SQL, PLI, JCL, Ada, XML, HTML, JDBC, ODBC, WinRunner, DynaComm te razne simulatorske jezike. Jednom izgrađena shema ispitnog pogonskog programa vrlo rijetko se mijenja. Svi modeli mogu koristiti istu shemu ispitnih pogonskih programa kako bi

proizveli ispitne skripte za zahtjeve zapisane u svakom modelu. Kada se funkcionalnost sustava promijeni ili nadogradi, logika u modelu se također mijenja, a svi vezani ispiti ponovo se stvaraju koristeći postojeću shemu ispitnog pogonskog programa. Ako se promijeni ispitna okolina, samo se treba promijeniti shema ispitnog pogonskog programa. Ispitni pogonski programi vezani za pojedini model mogu se ponovo stvoriti bez promjena u modelu.



Slika 3.1 Konceptualni proces *ispitivanja zasnovanog na modelu*

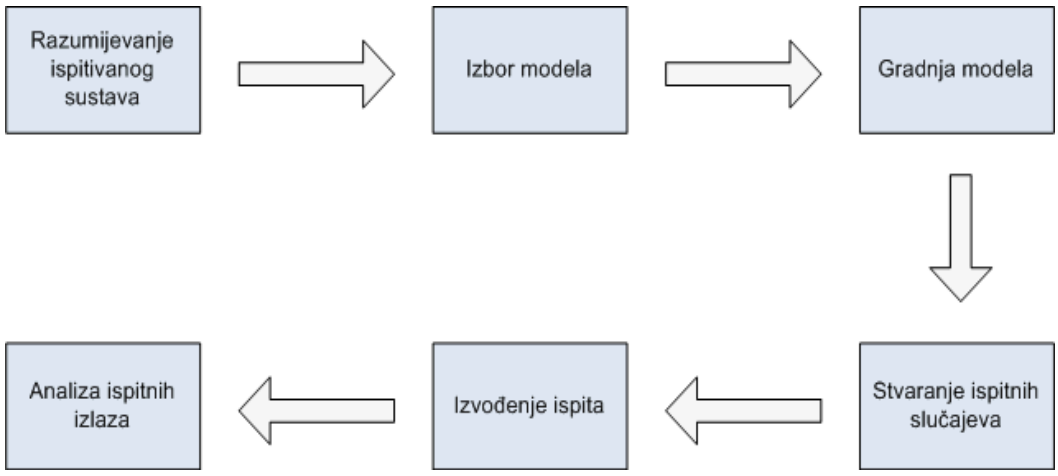
Sučelja ispitnog modela sustava povezana su na programska sučelja programskog ostvarenja (engl. *implementation*) sustava putem tehnike preslikavanja objekata pa stvoreni ispitni pogonski programi podržavaju automatsko izvođenje ispita. Razvijatelj programskog ostvarenja sustava često ima na raspolaganju ispitne pogonske programe prije nego što završi programskog ostvarenje. Time se smanjuje ispitivanje koje obično razvijatelj programskog ostvarenja izvodi tijekom svog rada.

U nastavku poglavlja detaljno se opisuje čitav proces *ispitivanja zasnovanog na modelu*, podijeljen u šest koraka. Postoji nekoliko pristupa *ispitivanju zasnovanom na modelu* koji se

razlikuju ovisno o sedam različitih kriterija vezanih uz model, stvaranje i izvršavanje ispita. Na kraju se iznosi razradba najzastupljenijih alata te nešto detaljniji opis alata CTG koji je korišten u sklopu praktičnog dijela diplomskog rada.

3.1. Proces ispitivanja zasnovanog na modelu

Proces *ispitivanja zasnovanog na modelu* dijeli se u nekoliko koraka opisanih slikom 3.2. Prvi korak procesa ispitivanja je *razumijevanje ispitivanog sustava*. Za provedbu ispitivanja visoke kakvoće, najprije je potrebno stvoriti mentalni prikaz funkcionalnosti sustava na osnovi specifikacije zahtjeva sustava. Nakon pribavljene dokumentacije i proučene funkcionalnosti sustava, *odabire se prikladan model*. Slijedi korak *gradnje modela* upotrebom jednog od dostupnih alata koji podržavaju automatizaciju *ispitivanja zasnovanog na modelu* za odabrani tip modela . Na osnovu izgrađenog modela, automatski se *stvaraju ispitni slučajevi*. Stvoreni ispitni slučajevi izvode se na ispitivanom programskom sustavu. Tijekom *izvođenja ispita*, prikupljaju se odgovori ispitivanog sustava te se potom vrši *analiza ispitnih izlaza* kako bi se utvrdila valjanost sustava.



Slika 3.2 Pojednostavljeni prikaz uobičajenih aktivnosti *ispitivanja zasnovanog na modelu*

3.1.1. Razumijevanje ispitivanog sustava

Uobičajeni zahtjev za većinu tehnika ispitivanja je dobro razumijevanje ciljeva rada programskog sustava. Stvaranje mentalnog prikaza funkcionalnosti sustava je preduvjet za stvaranje modela. Stvaranje mentalnog prikaza nije trivijalan zadatak pošto sustavi danas uobičajeno imaju zamršena sučelja i složenu funkcionalnost. Pored toga, programski sustav

postavljen je unutar operacijskih sustava pored mnoštva drugih primjenskih sustava, dinamčki povezanih knjižnica te datotečnih sustava s kojima međudjeluje ili utječe na njih na neki način.

Da bi se razvilo razumijevanje nekog primjenskog sustava, inženjeri ispitivači trebaju se upoznati sa funkcionalnošću sustava, ali i njegovom okolinom. Koristeći prikladne istraživačke tehnike i pregledavajući dostupne dokumente, inženjeri ispitivači skupljaju informacije potrebne pri izgradnji prikladnog modela. Prilikom razvoja mentalnog prikaza funkcionalnosti sustava potrebno je ostvariti nekoliko aktivnosti.

Na osnovu ciljeva ispitivanja, odrediti komponente/svojstva koja se trebaju ispitati. Nijedan model nije idealan za opisivanje složenog ili velikog sustava. Odluka o tome što će se modelirati prvi je koraku održavanju *ispitivanja zasnovanog na modelu* praktičnim.

Prikupiti korisnu dokumentaciju. Kao većina inženjera ispitivača, inženjeri ispitivači koji se bave *ispitivanjem zasnovanom na modelu* moraju naučiti što je više moguće o sustavu koji se ispituje. Pregledavanje zahtjeva, dijagrama korištenja (engl. *use case diagram)*, specifikacija, korisničkih uputa te bilo kakve dostupne dokumentacije važni su za razjašnjavanje nejasnoća oko toga što i kako programski sustav treba raditi.

Ako je moguće, uspostaviti komunikaciju sa timom za definiranje zahtjeva, timom za oblikovanje te timom za razvoj. Razgovor sa timovima na projektu štedi puno vremena i truda, posebno kad je riječ o izboru i izgradnji modela. Postoje tvrtke koje grade nekoliko tipova modela tijekom razdoblja definiranja zahtjeva i oblikovanja. No, nepotrebno je raditi model ispočetka ako se može ponovo iskoristiti ili prilagoditi za svrhe ispitivanja, pritom štedeći vrijeme i druge resurse. Štoviše, mnoge nejednoznačnosti u prirodnom jeziku i/ili formalnim dokumentima bolje se razrješavaju izravnim kontaktom nego čitanjem izvještaja i dokumenata.

Prepoznati korisnike sustava. Svaki entitet koji pruža ili koristi podatke sustava, ili utječe na sustav na neki način mora se zabilježiti. Primjerice, ti entiteti mogu biti korisnička sučelja, unosi sa tipkovnice ili mišem, jezgra operacijskog sustava, mreža, datoteke, baze podataka i ostala vanjska skladišta podataka, programirljiva sučelja koje sustav pruža ili koristi. Takvo prepoznavanje korisnika na prvi pogled djeluje zbunjujuće jer mnogi inženjeri ispitivači te razvijatelji programa nisu zadovoljni idejom da programski sustav ima korisnike koji nisu ljudi niti programirljiva sučelja. Prepoznavanje korisnika sustava prvi je korak za proučavanje događaja, što pomaže inženjerima ispitivačima da razviju sposobnost dijagnosticiranja

neočekivanih rezultata ispitivanja. Na kraju, treba izdvojiti one korisnike čije ponašanje treba biti simulirano prema ciljevima ispitivanja.

Pobrojati ulaze i izlaze svakog korisnika. Na prvi pogled ovo se čini kao iznimno težak posao, uzevši u obzir sve korisnike, te zamorno da bi se radilo ručno. Međutim, podjela posla prema korisnicima, komponentama ili svojstvima značajno smanjuje zamornost posla. Uz to, dostupni su komercijalni alati koji olakšavaju rad tako da automatski uočavaju korisničke kontrole u grafičkom korisničkom sučelju te moguće funkcije u programskim sučeljima.

Proučiti domene svakog ulaza. Kako bi se stvorili korisni ispiti u kasnijim koracima, trebaju se proizvesti stvarne, smislene vrijednosti za ulaze. Treba se provesti istraživanje graničnih, neispravnih te ispravnih vrijednosti za svaki ulaz. Ako je ulaz poziv funkcije, tada je potrebna slična analiza za povratnu vrijednost te svaki od parametara. Nakon toga, provodi se prikladna apstrakcija ulaza kako bi se pojednostavio proces modeliranja. Ulazi koji se mogu simulirati na isti način sa identičnim domenama mogu se shvatiti kao jedan. Također, moguć je i obrnuti slučaj: pozivi iste funkcije sa različitim parametrima mogu se shvatiti kao više različitih ulaza.

Dokumentirati informacije o primjenjivosti ulaza. Za stvaranje korisnih ispita, model treba sadržavati informacije o uvjetima koji određuju da li neki ulaz može biti primijenjen od strane korisnika. Na primjer, čovjek ne može pritisnuti gumb na nekom prozoru ako taj prozor nije otvoren ili aktivan. Uz to, inženjeri ispitivači trebaju primijetiti promjene u ovim uvjetima aktiviranih od strane jednog od tih ulaza.

Dokumentirati uvjete pod kojima sustav odgovara na ispitne ulaze. Odgovor sustava je izlaz prema jednom od svojih korisnika ili promjena u njegovim unutrašnjim podacima koja utječe na njegovo ponašanje u nekom budućem trenutku. Trebaju se proučiti uvjeti pod kojima ulazi uzrokuju odgovarajuće odgovore. Ne samo da to pomaže inženjerima ispitivačima da evaluiraju rezultate ispitivanja, već i oblikuju ispite koji uzrokuju određene odgovore.

Proučiti sljedove ulaza koji se trebaju modelirati. Proučavanjem sljedova ispitnih ulaza, tj. ispitnih slučajeva koje treba modelirati, traži se odgovor na sljedeća pitanja. Da li su svi ulazi primjenjivi tijekom cijelog vremena? Pod kojim uvjetima sustav očekuje ili prihvaća određene ulaze? Kojim redoslijedom bi sustav trebao obrađivati povezane ulaze? Koji su uvjeti da određeni sljedovi ulaza proizvedu određene izlaze? Odgovarajući na takva pitanja, otkriva se i sprječava većina pogrešnih shvaćanja funkcionalnosti sustava.

Shvatiti strukturu i semantiku vanjskih skladišta podataka. Ova aktivnost posebno je važna kada sustav drži podatke u velikim datotekama ili relacijskim bazama podataka. Poznavanje

izgleda i značenja podataka uzrokuje mogućnost da slaba i riskantna područja budu izložena analiziranju. Takva aktivnost pomaže u izgradnji modela koji stvaraju ispite koji uzrokuju kvar vanjskih podataka ili koji aktiviraju greške u sustavu sa čudnim kombinacijama podataka i ulaznih vrijednosti.

Shvatiti unutarnja međudjelovanja i računanje podataka. Kao kod prethodne aktivnosti, ova aktivnost pomaže inženjeru ispitivaču u shvaćanju ispitivanog sustava te sposobnosti modela da stvori ispitne podatke koji otkrivaju neispravnosti sustava. Tok internih podataka među različitim komponentama važan je za gradnju modela više razine. Interni izračuni koji su posebno aritmetički riskantni, poput dijeljenja ili operacija sa brojevima sa pomičnim zarezom visoke preciznosti, uobičajeno su skloni greškama.

Održavati model. Malo je razloga za stvaranje dokumenta sa svim važnim podacima, osim ako to ne zahtijevaju propisi organizacije. Održavanje skupa pokazivača na sve dokumente koji sadržavaju potrebnu dokumentaciju uglavnom je poželjno. Neke aktivnosti, poput proučavanja primjenjivosti ulaza i ulaznih sljedova, možda nisu standardne u ranijim razdobljima inženjerskog procesa (engl. *engineering process*), iako pružaju značajne informacije za proces modeliranja pa su vrijedne dokumentiranja. Također, preporučuje se i dokumentiranje problema nastalih tijekom modeliranja te logike za donesene odluke. U nedostatku bilo kakve dokumentacije, značajno je i dokumentiranje svih istraživačkih nalaza. Na kraju, posve je razumno uz model zabilježiti i komentare te zapažanja, posebno ako nema puno dokumentacije te ako alati za modeliranje to dopuštaju. Konačno, to je i cilj modela: da pokaže razumijevanje ponašanja programskog sustava.

3.1.2. Izbor modela

Izbor modela svodi se na naše znanje o modelu i o području primjene. Ne postoje modeli programskog sustava koji su dobri za sve namjene, već se za svaku situaciju mora donijeti odluka koji model (ili skup modela) je najprikladniji.

Teorija automata daje klasifikaciju automata i vrste jezika koje prihvaćaju. Ako shvaćamo što neki primjenski sustav radi, osnovna teorija automata može preporučiti model ili, obratno, može nam reći koji dio sustava možemo modelirati koristeći određeni automat. Teorija automata pruža načine odlučivanja o istovjetnosti različitih modela. Na primjer, dijagrami stanja nisu istovjetni sa konačnim automatima. Konačni automati strukturno su istovjetni izravnim grafovima. Kako su ispiti stvoreni na osnovu modela zapravo putovi u grafu, teorija

grafova nudi algoritme pretraživanja grafa te kriterij pokrivenosti ispitivanja u obliku strukture grafa. U ispitivanju računalnog sklopovlja već duže vrijeme koriste se automati.

Gramatike su također dio osnovne teorije automata. O gramatikama se zna jednako mnogo kao i o automatima, kada je u pitanju koji programski sustav se može modelirati sa kojim tipom gramatike. Što se tiče stvaranja ispita, najjednostavniji oblik generatora ispita je *unazadni prevoditelj* (engl. *reverse compiler*) koji proizvodi strukturno i semantički odgovarajuće sljedove ulaza umjesto da ih parsira. Kako su alati za prevođenje široko dostupni, izgradnja generatora ispita zasnovanog na gramatici nije teška. Nažalost, ne postoji teorija koja pomaže pri upravljanu ispitivanjem i postizanju kriterija pokrivenosti ispitivanja kada se koristi gramatika.

Različita područja primjene bolje odgovaraju određenim vrstama modela. Ne postoje važnije studije širih razmjera koje pokazuju koje modele treba upotrebljavati za koja područja primjene. Međutim, postoje intuitivna zapažanja.

Za modeliranje HTML datoteka Web preglednika ili matematičkih izraza za stručni kalkulator, izbor gramatika najvjerojatnije je najbrži i najjednostavniji pristup, pošto se gramatike često koriste za opis takvih jezika.

Telefonski sustavi imaju veliki broj stanja te se mnoge greške u takvim sustavima otkrivaju ispitivanjem opterećenja kroz duže vremensko razdoblje, čineći tako automate idealnim rješenjem za takve namjene.

Paralelni sustavi uglavnom imaju komponente koje se izvode konkurentno. Ako se pojedinačne komponente mogu modelirati korištenjem automata, dijagrami stanja su dobar odabir za modeliranje takvih sustava. Dijagrami stanja mogu biti u više stanja istovremeno te stanje dijagrama stanja može biti novi dijagram stanja.

Ukoliko je sustav uvijek u samo jednom stanju, a prijelazi ovise (osim ulaza) o nekim vanjskim uvjetima, dijagram stanja je odgovarajući izbor. Prijelazi u dijagramima stanja ovise o ulaznom okidaču te vanjskim uvjetima.

Ako se sustav može modelirati konačnim automatom, ali postoji nakana za statističkom analizom pogrešnih podataka, upotreba Markovljevih lanaca je najprikladnija.

Ako pokušavamo saznati da li sustav može izdržati dugačke, složene, neponavljajuće sljedove ulaza, automati su dobar odabir.

Ako nas zanima da li sustav prihvaća sljedove ulaza u određenoj strukturi, npr. da li prevoditelji prihvaćaju programe napisane u jeziku kojeg bi trebali prevoditi, gramatika je najbolji izbor.

Kod primjenskih sustava zasnovanih na nekom mrežnom protokolu preferira se upotreba gramatika. Kod takvih područja primjene, naglasak je na strukturi i semantici paketa, a ne na sadržaju sljedova paketa.

Ako postoji potreba da se predstave uvjeti pod kojima ulazi uzrokuju određeni odgovor, a automati nisu prikladni, tada se obično koriste tablice odlučivanja.

3.1.3. Izgradnja modela

Općenito, inženjeri ispitivači koji rade sa modelima zasnovanim na stanjima definiraju apstrakcije stanja visoke razine koje naknadno prerađuju u stvarni prostor stanja ispitivanog programskog sustava. Ručno pretraživanje prostora stanja je zahtjevno, osim prilikom modeliranja manjih sustava.

Apstrakcije stanja zasnivaju se na ulazima i informaciji o primjenjivosti svakog ulaza te ponašanju kojeg ulaz uzrokuje. Općenita procedura koju koriste mnogi pristupi *ispitivanju zasnovanom na modelu* dijeli se na tri dijela:

1. Napraviti listu ulaza (kao što je opisano u odjeljku 3.1.1)
2. Za svaki ulaz dokumentirati situacije u kojima se ulaz može primijeniti od strane korisnika te obratno, situacije u kojima korisnici nisu u mogućnosti primijeniti određeni ulaz. Te situacije nazivaju se još *ograničenja primjenjivosti ulaza*. Primjerice kod telefona, ulaz „podigni slušalicu“primjenjuje se samo kada je slušalica spuštena. Moderna grafička korisnička sučelja stvaraju mnoge takve situacije u kojima primjerice dijaloški okviri prisiljavaju korisnika da odgovori na taj dijaloški okvir te onemogućuje bilo kakav drugi ulaz.
3. Za svaki ulaz dokumentirati situacije u kojima ulaz uzrokuje stvaranje različitih ponašanja (ili izlaza) ovisno o kontekstu u kojemu se ulaz primjenjuje. Takve situacije poznate su kao *ograničenja ponašanja ulaza*. Primjerice, ulaz „podigni slušalicu“ uzrokuje niz različitih ponašanja. Ako je u kontekstu telefon u stanju mirovanja, rezultirajuće ponašanje je stvaranje signala tonskog biranja. No, ako telefon već zvoni zbog dolaznog poziva, tada sustav spaja ta dva telefona i omogućuje razgovor. Dva različita ponašanja uzrokovana su istim ulazom.

I *ograničenje primjenjivosti ulaza* i *ograničenje ponašanja ulaza* opisuju scenarij koji se treba ispitati. Jedna od glavnih značajki *ispitivanja zasnovanog na modelu* je posljedica da inženjer ispitivač detaljno razmišlja o situacijama navedenih ograničenja. Oba tipa ograničenja predstavljaju se skupovima. Na primjer, sljedeći skupovi dokumentiraju spomenuta ograničenja:

Status telefonske slušalice = {spuštena, podignuta}

Dolazni poziv = {da, ne}

Jednom kada se svako ograničenje dokumentira, množenjem skupova (Kartezijev produkt) dobiva se prostor stanja.

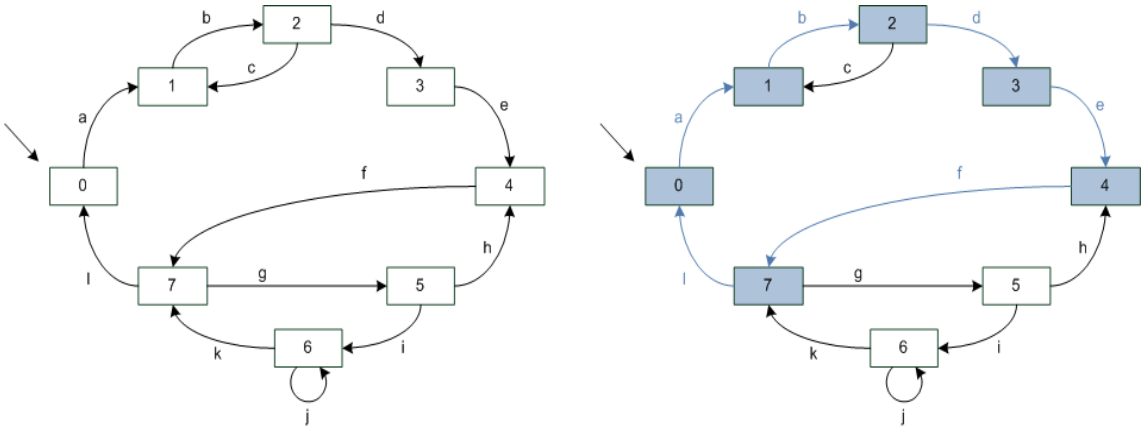
Prowell [23] uvodi jezik TML (engl. *The Modelling Language*) koji praktično opisuje modeliranje programskog sustava na osnovu Markovljevih lanaca. Whittaker [22] piše o gradnji modela zasnovanih na konačnim automatima na hijerarhijski ručni način. El-Far [24] opisuje programski okvir za automatsku gradnju modela zasnovanih na konačnim automatima. Postoje i radovi koji govore o automatizaciji stvaranja vjerojatnosti za Markovljeve lance [25], [26]. Proces razvoja drugih tipova modela (npr. gramatike) sličan je, unatoč razlici u konačnoj prezentaciji modela.

Vremenski trenutak početka aktivnosti ispitivanja izuzetno je važan kod korištenja *ispitivanja zasnovanog na modelu*. Rani početak ispitivanja omogućuje ranu izgradnju ispitnih modela. Naravno, takvi modeli sadrže visoki stupanj apstrakcije, ali se tijekom razvoja (i ranog ispitivanja) sustava modeli mogu proširivati s više detalja. Kada se neko svojstvo sustava više nema namjeru ispitivati, odgovarajući detalji na modelu mogu se ukloniti. Nažalost, rokovi i planiranja u stvarnom svijetu ne dopuštaju uvijek rani početak modeliranja te inženjeri ispitivači završavaju sa zadatkom da modeliraju samo dijelove cijelog složenog sustava.

Vještine inženjera ispitivača nisu dovoljne za vođenje procesa ispitivanja visoke kakvoće. U nedostatku alata koji podupiru razne aktivnosti *ispitivanja zasnovanog na modelu*, troškovi koji će nastati u timu koji se bavi ispitivanjem možda neće biti opravdani iz perspektive broja pronađenih grešaka, čak i nakon duljeg vremenskog razdoblja Organizacije koje žele ostvariti *ispitivanje zasnovano na modelu* za koji nema potpore alata, trebale bi razmisliti o razvoju vlastitog radnog okvira ili odgovarajućih alata s kojima bi mogli graditi i održavati modele.

3.1.4. Stvaranje ispitnih slučajeva

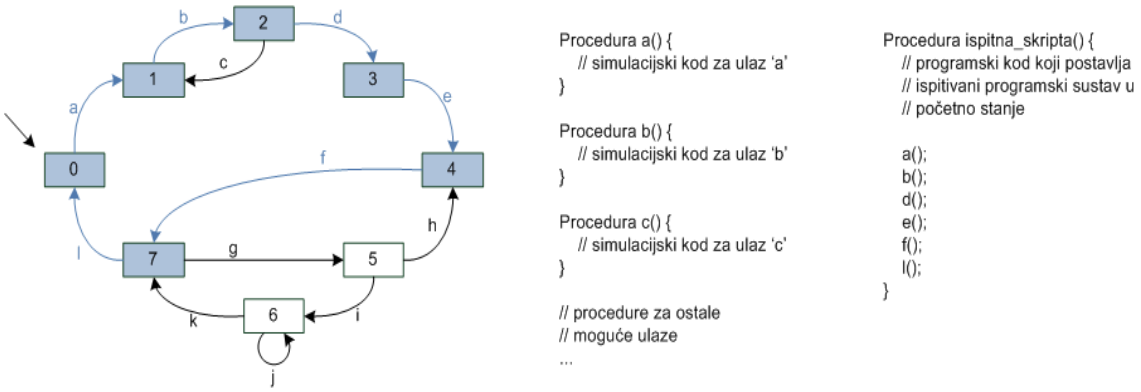
Složenost stvaranja ispita na osnovu modela ovisi o tipu i veličini modela. Modeli koji su korisni za ispitivanje obično posjeduju svojstva koja omogućuju automatsko stvaranje ispita. Za većinu pristupa *ispitivanju zasnovanom na modelu* potrebno je ugraditi znanje kombinatorike jer se ispiti stvaraju prolaskom kroz sve moguće kombinacije uvjeta opisanih u modelu. U slučaju konačnih automata, potrebno je ostvariti algoritam koji obilazi dijagram prijelaza stanja s podjednakom mogućnosti izbora kod nedeterminističkih prijelaza stanja. Tako stvorene putanje, koje se sastoje od sljedova prijelaza među stanjima, predstavljaju jedan ispitni slučaj. Primjerice, na dijagramu prijelaza stanja na slici 3.3, slijed ulaza „a, b, d, e, f, l“ kvalificira se kao jedan ispitni slučaj predstavljenog sustava. Primjer ispitnog slučaja na slici je označen plavom bojom.



Slika 3.3 Automat (lijevo) te ispitna putanja automata (desno)

3.1.5. Izvođenje ispita

S obzirom na izvođenje ispita, moguće je ispitne slučajeve izvoditi čim se izgrade ili nakon što se stvori cijeli skup ispitnih slučajeva koji udovoljava određenom kriteriju prikladnosti. U oba slučaja, prvo se stvaraju simulacijske skripte ulaza korisnika. Simulacijske skripte ulaza sadrže simulacijski kod koji ispitna skripta obavlja za svaki ulaz. Dakle, broj simulacijskih procedura odgovara broju ulaza. Potom generator ispitnih pogonskih programa stvara programski kod ispitnih skripti obilaskom putanje modela, pritom izravno pozivajući simulacijske procedure uz pripadne prijelaze. Na kraju, stvorena ispitna skripta izvodi ispitne slučajeve. Opisani postupak prikazan je slikom 3.4.



Slika 3.4 Ispitna putanja automata (lijevo), simulacijska skripta za svaki ulaz (sredina) te ispitna skripta za navedenu putanju (desno)

3.1.6. Analiza ispitnih izlaza

Vrednovanje rezultata ispitivanja vjerojatno je najteži dio procesa ispitivanja. Inženjeri ispitivači moraju odrediti da li je programski sustav dao ispravne izlaze na osnovu slijeda ispitnih ulaza. U praksi to znači potvrđivanje ispravnosti izlaza ispisanog na ekranu, potvrđivanje ispravnosti izlaza interno spremljenih podataka te utvrđivanje zadovoljavanje zahtjeva sa sredstvima sustava (vrijeme i prostor). *Ispitivanje zasnovano na modelu* ne olakšava tu situaciju. Ispravnost izlaza i internih varijabli moraju se provjeriti usporedbom sa specifikacijama. Međutim, *ispitivanje zasnovano na modelu* dodaje još jednu dimenziju korisnu u praksi – potvrđivanje ispravnosti stanja.

Stanja su apstrakcije internih podataka pa se zato lakše provjeravaju. Na primjer, model će zapamtiti svaku promjenu stanja koja se dogodi (ili bi se trebala dogoditi) u programskom sustavu tijekom ispitivanja. Prema tome, model se ponaša kao vrlo precizna i detaljna specifikacija informirajući inženjera ispitivača koji ulazi bi trebali biti dostupni te koje su vrijednosti svake pojedine apstrakcije podataka, uključujući i stanje ispitivanog sustava.

Kod tradicionalnog ispitivanja, ispiti se stvaraju i izvode jedan po jedan ili u skupini. Pretpostavimo da je inženjer ispitivač proveo 400 ispitnih slučajeva i našao točno 40 grešaka. Na kraju takvog ispitivanja ustanovljuje se da je provedeno 400 ispita i da je nađeno 40 grešaka, no ne može se reći kako bi se sustav ponašao ako se pokrene 401. ispitni slučaj.

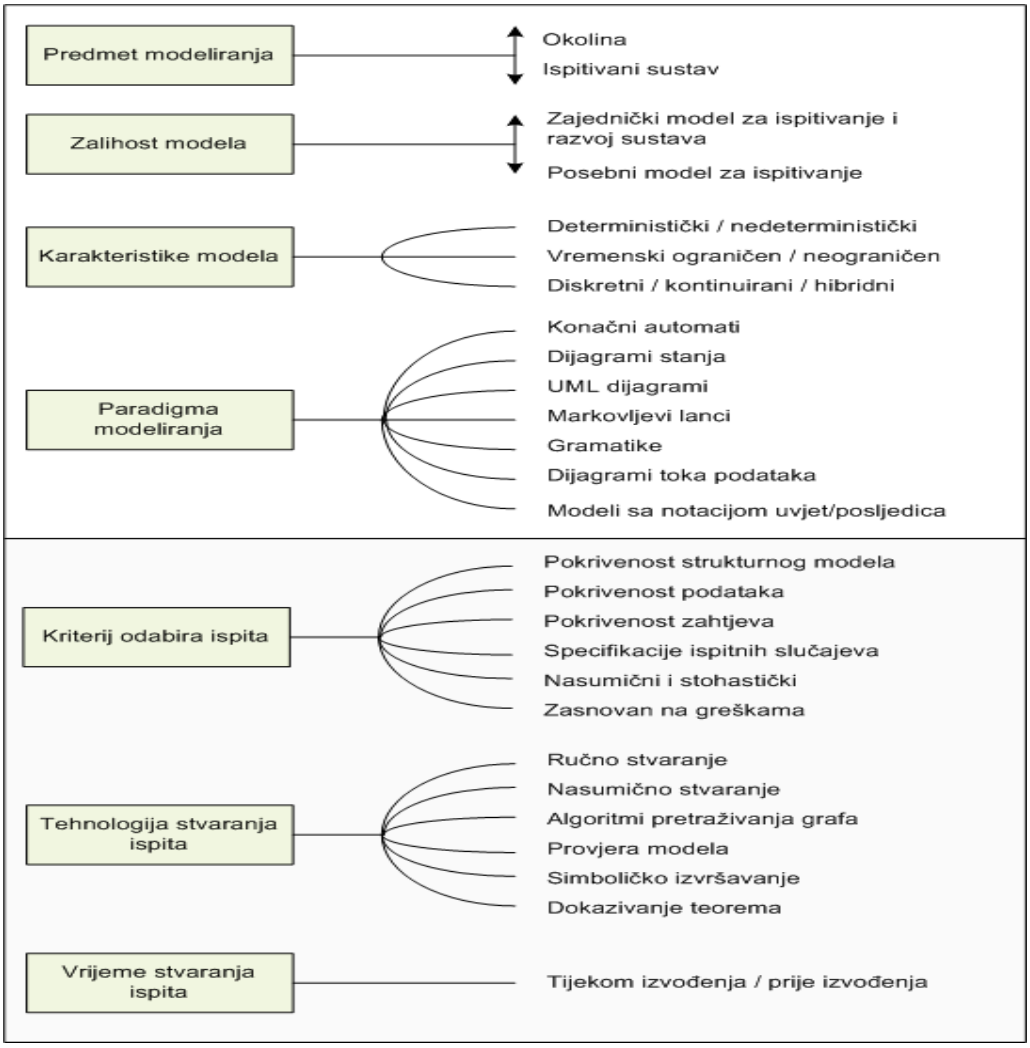
Modeli ne daju samo sliku ispita koji su provedeni, već i uvid u to koji ispiti nisu provedeni. Analiza prirode određivanja razlike između ta dva skupa oblikovala je korisnu metriku za

određivanje potpunosti ispita te pruža kriterij zaustavljanja svojstven modelu [27]. Primjerice, ispitivanje se zaustavlja kada nema više neponavljajućih ispita opisanih modelom.

3.2. Razradba pristupa ispitivanju zasnovanom na modelu

Postoji više različitih pristupa *ispitivanju zasnovanom na modelu*. Pristupi *ispitivanja zasnovanog na modelu* dijele se prema sedam različitih kriterija, kao što je prikazano na slici 3.5. Četiri kriterija prema kojima se vrši razradba pristupa vezani su uz model. Pristupi se dijele ovisno o *predmetu modeliranja*, *razini zalihosti modela*, *karakteristikama modela* te *paradigmi modeliranja*. Preostala tri kriterija vezana su uz stvaranje ispita. Pristupi se razlikuju ovisno o *kriteriju odabira ispita*, *tehnologiji stvaranja ispita* te *vremenu stvaranja ispita*. *Vrijeme stvaranja ispita* definira se relativno u odnosu na izvođenje ispita pa je moguće izvršavati ispite tek nakon što je stvoren skup svih ispitnih slučajeva ili naizmjence stvarati i izvoditi jedan po jedan ispitni slučaj.

Vertikalne strelice na slici 3.5 ukazuju na kontinuirani raspon mogućnosti, dok odabir oblika *A/B* ukazuje na međusobno isključive alternative. Zaobljene linije ukazuju na alternative koje nisu nužno međusobno isključive (npr. neki alati koriste i više od jedne tehnologije stvaranja ispita).



Slika 3.5 Razradba pristupa *ispitivanju zasnovanom na modelu* prema sedam kriterija

Svaki od kriterija po kojima se vrši razradba pristupa *ispitivanju zasnovanom na modelu* opisan je detaljnije u nastavku odjeljka.

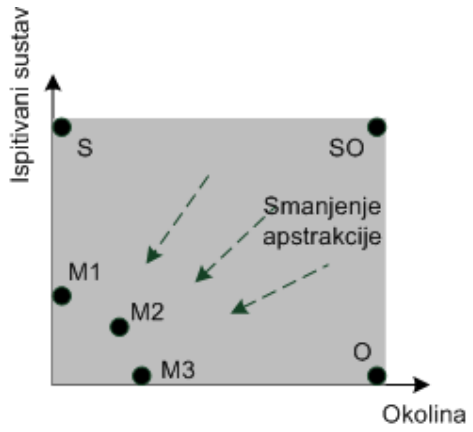
3.2.1. Predmet modeliranja

Prvi kriterij podjele pristupa *ispitivanju zasnovanom na modelu* je predmet modeliranja. S obzirom na predmet modeliranja, *ispitivanje zasnovano na modelu* može koristiti model namjeravanog ponašanja ispitivanog sustava te model mogućeg ponašanja okoline ispitivanog sustava. U praksi, najčešće se koristi kombinacija oba modela.

Model ispitivanog sustava ima dvije svrhe. Prvo, model odražava namjeravano ponašanje ispitivanog sustava. Drugo, njegova struktura upotrebljava se za stvaranje ispitnih slučajeva.

Model okoline ispitivanog sustava koristi se za ograničavanje mogućih ulaza u model. Kao takav, model okoline ograničava skup mogućih ponašanja modela ispitivanog sustava pa se u tom smislu ponaša kao kriterij odabira ispita. Modeli okoline definirani stohastičkim korisničkim profilima opisuju uobičajena međudjelovanja sa ispitivanim sustavom, npr. opisuju uobičajene uzorke podražaja ispitivanom sustavu.

Slika 3.6 prikazuje mogućnosti kombiniranja modela okoline i modela ispitivanog sustava. Vertikalna os pokazuje u kojem obujmu se modelira ponašanje ispitivanog sustava, dok horizontalna os pokazuje u kojem obujmu se modelira ispitna okolina. Zasjenjeno područje pokazuje sve moguće modele koji se mogu koristiti za *ispitivanje zasnovano na modelu*.



Slika 3.6 Razradba pristupa *ispitivanju zasnovanom na modelu*

s obzirom na kriterij predmeta modeliranja

Model na poziciji S je model koji uključuje sve detalje ispitivanog sustava, ali ne govori ništa o očekivanoj okolini. To znači da nisu nametnute granice koje sprječavaju nerazumne ulaze u ispitivani sustav.

Model O upravo je suprotan modelu S. Model O ima potpuno znanje o okolini u koju će biti smješten ispitivani sustav, ali ne zna ništa o željenom ponašanju ispitivanog sustava. To znači da model specificira sve legalne ispitne ulaze, ali ne daje nikakve informacije o očekivanim izlazima ispitivanog sustava.

Pozicija SO najekstremniji je slučaj, modelira se sve vezano uz ispitivani sustav i okolinu. U praksi, to je previše detalja da bi modeliranje bilo praktično jer model postaje jednako složen kao i sam sustav. Apstrakcija je ključna, zato su modeli poput M1, M2 i M3 uobičajeni za *ispitivanje zasnovano na modelu*.

3.2.2. Razina zalihosti modela

Drugi kriterij po kojem se dijele pristupi *ispitivanju zasnovanom na modelu* je razina zalihosti modela. Pristupi ispitivanja ugrubo se razlikuju u razini zalihosti između modeliranja za ispitivanje i modeliranja za programsko ostvarivanje. Dio pristupa *ispitivanju zasnovanom na modelu* koristi zajednički model za stvaranje ispita i za stvaranje programskog koda, dok drugi dio pristupa koristi odvojeni model za ispitivanje.

Zajednički model za stvaranje ispita i programskog koda

U nekim slučajevima izvršni programski kod stvara se iz ponašajnih modela. Primjerice, blok dijagrami Matlabovog Simulinka prevode se u izvršni programski kod. Također postoje i CASE (engl. *Computer-Aided Software Engineering*) alati koji pružaju sredstva za stvaranje simulacijskog ili stvarnog programskog koda iz formalnih modela poput dijagrama stanja. U takvim slučajevima koristi se isti model za stvaranje ispita i programskog koda.

Modeli za stvaranje programskog koda moraju biti vrlo detaljni, tako da nisu uvijek idealni za stvaranje ispita, za što je korisnije imati apstraktnije modele. Kod pristupa ispitivanju koje koriste zajednički model za stvaranje ispita i programskog koda nema zalihosti, tj. sustav se ispituje na sebi. To znači da se odluke opširnije od „bačena je iznimka“ moraju graditi ručno. Na prvi pogled čini se da nedostatak zalihosti ne predstavlja problem jer se zahtijeva da model bude verificiran. Međutim, u praksi se obično greške u modelu i programskom ostvarenju otkrivaju tek tijekom ispitivanja.

Ovaj pristup nije prikladan za ispitivanje funkcionalnosti sustava. Međutim, koristan je za ispitivanje generatora programskog koda ili generatora ispita te stjecanje pouzdanosti u pretpostavke okoline koje su korištene u modelu.

Odvojeni ispitni model

Ideja ovog pristupa je gradnja zasebnog modela u svrhe ispitivanja. Ispitni model ručno se izrađuje na osnovu specifikacija i koristi za stvaranje ispita, dok se ispitivani sustav gradi ručno na osnovu postojeće neformalne specifikacije. Kako se ispitni slučajevi i programski kod ne stvaraju iz istog formalnog dokumenta, stvara se neizbježna zalihost. Ovakav pristup trenutačno se češće koristi u praksi.

Jednom kada se izgradi nezavisni ispitni model, moguće ga je koristiti kao detaljnu specifikaciju sustava. Takav poseban model ima dva oblika iskoristivosti: služi kao specifikacija sustava te kao osnova za *ispitivanje zasnovano na modelu*. Ta karakteristika

posebno je važna u razvojnom kontekstu kada programsko ostvarenje i specifikaciju (ili model) ne izvodi samo jedan tim. Međutim, ispitni modeli vrlo su složene tvorevine pa zato obično zahtijevaju dodatnu dokumentaciju kako bi bili korisni i kao specifikacije. Štoviše, apstrakcije za svrhe ispitivanja i svrhe specifikacija mogu se razlikovati, što objašnjava razliku između jednog posebnog modela za ispitivanje i zajedničkog modela za ispitivanje i stvaranje programskog koda.

3.2.3. Karakteristike modela

Karakteristike modela vezane su uz nedeterminizam, vremenska pitanja te kontinuiranost ili diskretnost prirode modela.

Nedeterminizam se javlja i u modelu i u ispitivanom sustavu. Ako ispitivani sustav pokazuje nedeterminizam, npr. kao rezultat konkurentnosti, moguće je da ispitni podražaj predviđen od modela ovisi o prijašnjim reakcijama ispitivanog sustava. U tom slučaju, model sustava mora omogućavati izražavanje nedeterminizma.

Kod većine sustava za rad u stvarnom vremenu vrlo su važna vremenska svojstva modela. Zbog dodatnog stupnja slobode, ispitivanje takvih sustava iznimno je teško. Primjena ideje *ispitivanja zasnovanog na modelu* sustava za rad u stvarnom vremenu trenutno je predmet intenzivnog istraživanja [34].

Što se tiče dinamike, modeli mogu biti diskretni, kontinuirani ili kombinacija tih dviju vrsta (hibridni modeli). Većina radova kod *ispitivanja zasnovanog na modelu* fokusira se na diskretne sustave upravljane događajima. U mnogim ugrađenim sustavima (engl. *embedded system*) također su uobičajeni kontinuirani i hibridni modeli. Poput *ispitivanja zasnovanog na modelu* sustava za rad u stvarnom vremenu, tako je i ispitivanje kontinuiranih sustava trenutno predmet intenzivnijeg istraživanja.

3.2.4. Paradigma modeliranja

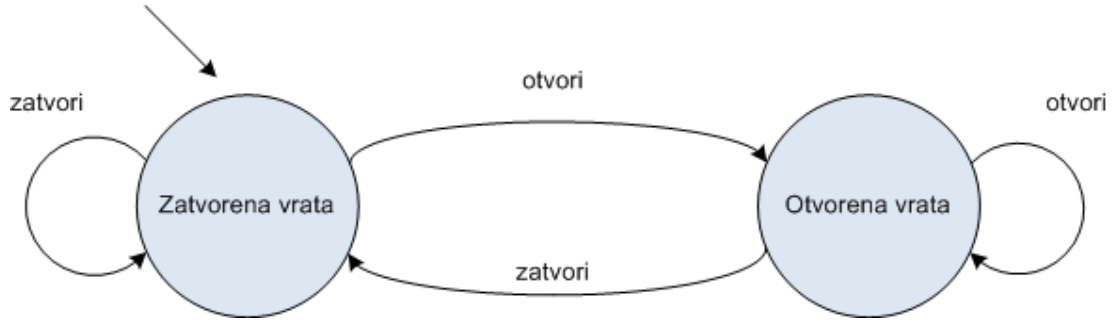
S obzirom na kriterij paradigme modeliranja, pristupi *ispitivanju zasnovanom na modelu* dijelimo prema odabranoj vrsti modela. Najčešće korišteni modeli su konačni automati, dijagrami stanja, UML (engl. *Unified Modelling Language*) dijagrami, Markovljevi lanci, gramatike, dijagrami toka podataka te modeli sa zapisom *uvjet/posljedica* (engl. *precondition/postcondition*).

Konačni automati

Teorija automata pojavila se prije početaka programerskog inženjerstva. Korištenje konačnih automata kao modela u oblikovanju i ispitivanju komponenti računalnog sklopovlja davno je započelo i danas se smatra uobičajenom praksom.

Konačni automati idealni su modeli za opis sljedova ulaza. U kombinaciji sa mnoštvom algoritama obilaska grafova, upotreba automata čini *ispitivanje zasnovano na modelu* manje mukotrpnim nego tradicionalno ispitivanje. Međutim, složeni sustavi povlače i gradnju velikih automata, čija izrada i održavanje nije zanemarivi posao.

Konačni automati primjenjivi su na bilo koji model koji se može precizno opisati sa konačnim brojem stanja. Tijekom ispitivanja, inženjer ispitivač unosi ulaze te potom ocjenjuje rezultate. Zatim odabire novi ulaz, ovisno o prethodnom rezultatu te potom opet ocjenjuje sljedeći skup mogućih ulaza. Inženjer ispitivač uvijek ima određeni skup mogućih ulaza koje bira. Skup mogućih ulaza mijenja se ovisno o stanju programskog sustava. Primjenski sustav uvijek je u točno određenom stanju te trenutno stanje primjenskog sustava određuje iz kojeg skupa ulaza inženjeri ispitivači odabiru ispitne ulaze.



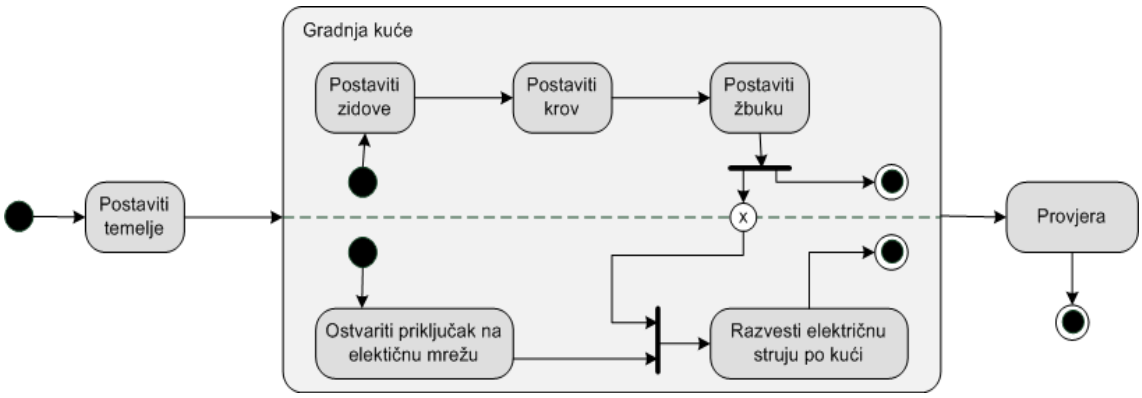
Slika 3.7 Primjer konačnog automata

Konačni automat je matematički model koji se sastoji od konačnog skupa stanja, konačnog skupa ulaznih znakova, funkcije prijelaza, početnog stanja i skupa prihvatljivih stanja. Slika 3.7 prikazuje primjer automat koji modelira zatvaranje i otvaranje vrata. Skup stanja sastoji se od stanja *Zatvorena vrata* i *Otvorena vrata*. Elementi ulazne abecede su *otvori* i *zatvori*. Može se pretpostaviti da su oba stanja prihvatljiva te da je početno stanje *Zatvorena vrata*. Funkcija prijelaza definira prijelaze među stanjima u ovisnosti o ulazu. Ako je automat u stanju *Zatvorena vrata* onda za ulaz *otvori* prelazi u stanje *Otvorena vrata*, dok za ulaz *zatvori*

ostaje u istom stanju. Ako je automat u stanju *Otvorena vrata* onda za ulaz *zatvor*i prelazi u stanje *Zatvorena vrata*, dok za ulaz *otvori* ostaje u istom stanju.

Dijagrami stanja

Dijagrami stanja proširenje su konačnih automata koji se posebno koriste za modeliranje složenih sustava ili sustava u stvarnom vremenu. Oni pružaju okvir za slaganje automata u hijerarhiju, pri čemu jedno stanje može biti prošireno u novi automat niže razine. Dijagrami stanja također pružaju potporu za konkurentne automate. Pored toga, struktura dijagrama stanja uključuje i vanjske uvjete koji utječu na izvršavanje prijelaza iz jednog stanja u drugo, što najčešće smanjuje veličinu modela koji se gradi. Dijagrami stanja intuitivno su istovjetni najširem obliku automata - Turingovom stroju. Međutim, dijagrami stanja su praktičniji, a pritom zadržavaju jednaku mogućnost izražavanja. Dijagrami stanja lakši su za čitanje nego konačni automati, no rad s njima nije trivijalan te je potrebno prethodno učenje i upoznavanje s njima.

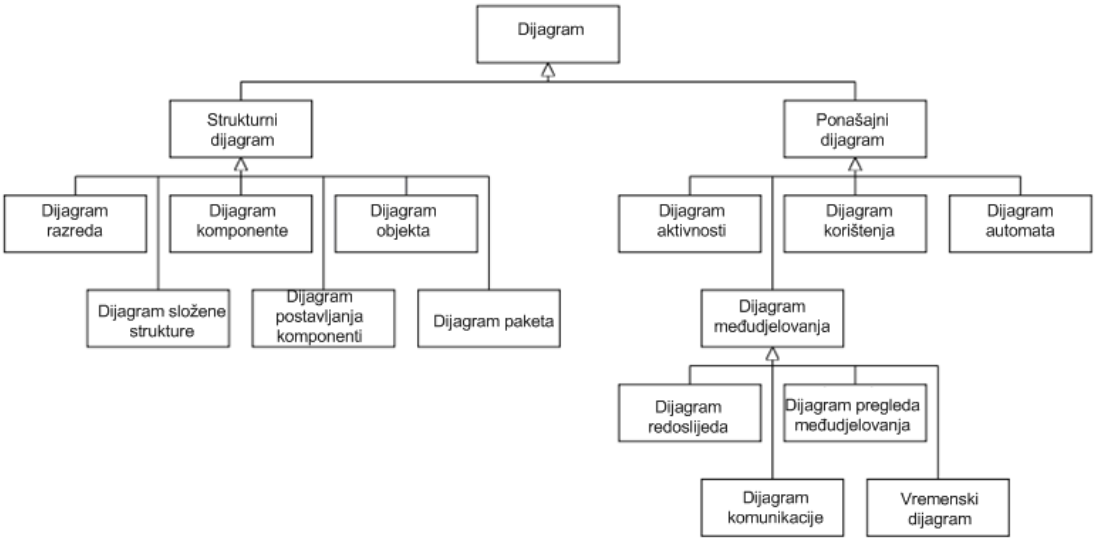


Slika 3.8 Primjer dijagrama stanja

Primjer dijagrama stanja sa slike 3.8 predstavlja pojednostavljeni model gradnje kuće. Stanja na dijagramu stanja označena su pravokutnicima zaobljenih vrhova, a prijelazi strelicama. Kružić popunjen crnom bojom predstavlja početno stanje, a kružić ispunjen crnom bojom u sredini i bijelom uz rub kružnice predstavlja konačno stanje. Stanje *Gradnja kuće* složeno je stanje koje se sastoji od dva automata logički odijeljena crtkanom linijom. Bijeli kružić s oznakom „x“ označava sinkronizacijski mehanizam koji omogućava prijelaz drugom automatu da prijeđe u stanje *Razvesti električnu struju po kući* tek nakon što prvi automat obavi akciju vezanu uz stanje *Postaviti žbuku*.

UML dijagrami

UML je standardizirani jezik za modeliranje objekata opće namjene. UML je za modele isto što i C/C++ za programe – široko prihvaćen način opisa vrlo složenih ponašanja. Prednost UML dijagrama je mogućnost sadržavanja ostalih vrsta dijagrama, npr. u ispitivanju se često koriste dijagrami stanja kao komponente unutar šireg UML okvira. Specifikacije UML 2.0 definiraju 13 vrsta dijagrama koji su hijerarhijski kategorizirani prema slici 3.9.



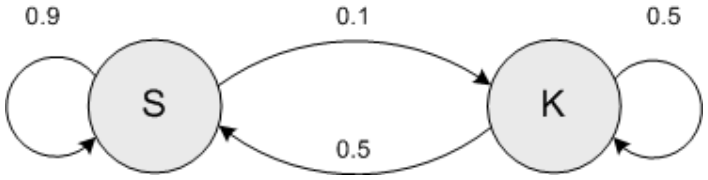
Slika 3.9 Hijerarhija UML dijagrama

U skupinu strukturnih dijagrama spadaju dijagram razreda, dijagram komponenti, dijagram složene strukture, dijagram postavljanja, dijagram objekata i dijagram paketa. Strukturni dijagrami naglašavaju unutrašnje dijelove modeliranog sustava. U grupu ponašajnih dijagrama spadaju dijagram aktivnosti, dijagram stanja i dijagram korištenja. Ponašajni modeli opisuju ponašanje modeliranog sustava. Posljednju skupinu, grupu interaktivnih dijagrama, čine dijagram komunikacije, dijagram pregleda međudjelovanja, dijagram sljedova i dijagram vremena. Interaktivni dijagrami naglašavaju tok upravljanja i podataka unutrašnjih dijelova modeliranog sustava. Slika 3.9 prikazuje hijerarhiju UML dijagrama opisanu dijagramom razreda.

Markovljevi lanci

Markovljevi lanci su stohastički modeli. Za modeliranje programskih sustava koristi se poseban razred Markovljevih lanaca – *neskrativi* (engl. *irreducible*) *Markovljev lanac* konačnih stanja, diskretnih parametara, homogenog vremena. Strukturno su slični konačnim automatima te se mogu shvatiti kao stohastički automat. Značaj neskrativih *Markovljevih*

lanaca nije samo u stvaranju ispita, već i u prikupljanju i analizi grešaka. Njihovom uporabom mogu se procijeniti mjere sustava poput pouzdanosti i srednjeg vremena do kvara sustava (engl. *Mean Time Before Failure*, skraćeno MTBF).



Slika 3.10 Primjer Markovljevog lanca

Markovljev lanac je slijed slučajnih varijabli X1, X2, X3,... sa svojstvom da raspodjela vjerojatnosti sljedećih stanja ovisi samo o trenutnom stanju, tj. nezavisna je o prošlim stanjima. Slika 3.10 prikazuje jednostavni model prognoze vremena sa sljedećom matricom prijelaza:

$$P = \begin{bmatrix} 0.9 & 0.1 \\ 0.5 & 0.5 \end{bmatrix}$$

Ako je danas sunčani dan (stanje *S*), primjer Markovljevog lanca sa slike 3.10 pokazuje da je 90% vjerojatnost da će i sutra biti sunčani dan. Ako je danas kišni dan (stanje *K*), vjerojatnost da će i sutra biti kišni dan je 50%.

Gramatike

Gramatika je precizan opisan formalnog jezika. Različiti razredi gramatika istovjetni su različitim razredima automata, no ponekad su gramatike puno jednostavniji i sažetiji prikaz za modeliranje sustava. Gramatika se zadaje skupom završnih znakova, skupom nezavršnih znakova, skupom produkcija i početnim nezavršnim znakom.

Primjerice, gramatika koja opisuje skup cijelih brojeva opisuje se sljedećim produkcijama:

```
S → '-' B | B
B → D | DB
D → '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

Nezavršni znakovi gramatike označeni su slovima (S, B, D), pri čemu je S početni nezavršni znak gramatike. Završni znakovi nalaze se unutar jednostrukih navodnika (-, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9).

Međutim, postoji jako malo članaka o korištenju gramatika kod *ispitivanja zasnovanog na modelu* te su još uvijek neistražena područja stvaranja ispita i definiranja pokrivenosti ispitivanja.

Dijagram toka podataka

Dijagram toka podataka grafički je prikaz toka podataka kroz informatički sustav. Dijagram toka podataka prikazuje procese, spremišta podataka, vanjske entitete te povezujuće tokove podataka (slika 3.11).

Slika 3.11 Primjer dijagrama toka podataka

Spremišta podataka označava se pravokutnikom bez desne stranice. Dijagram toka podataka ne govori ništa o vremenu spremanja podataka, niti općenito ništa o relativnim vremenima procesa. Procesi koriste ulaze iz spremišta podataka i na osnovu njih stvaraju izlaze. Proces sustava označava se pravokutnikom zaobljenih vrhova. Vanjski entiteti su izvan procesa modeliranja i označavaju se pravokutnicima. Tokovi podataka označavaju se strelicama te pokazuju kako se podaci kreću između spremišta podataka, procesa sustava i vanjskih entiteta.

Model s zapisom *uvjet/posljedica*

Modeli sa zapis *uvjet/posljedica* modeliraju sustav kao skup varijabli koje predstavljaju snimak unutarnjih stanja sustava te operacija koje mijenjaju te varijable. Umjesto definiranja operacija kodom kao programskim jezicima, one se definiraju skupom uvjeta i posljedica. Dakle ne radi se o grafičkom modelu, a primjeri najkorištenijih zapisa *uvjet/posljedica* su Z [52], VDM [53] i JML [54].

3.2.5. Kriterij odabira ispita

Peti kriterij razradbe pristupa ispitivanju definira sredstva koja se koriste za upravljanje stvaranjem ispita. Alati se mogu klasificirati ovisno o tome koju vrstu kriterija odabira ispita podupire. Definiranje općenito najboljeg kriterija nije moguće, već je zadatak inženjera ispitivača da konfigurira sredstva stvaranja ispita i odabere prikladan kriterij odabira ispita te specifikacije ispitnih slučajeva. Slijedi opis najčešće korištenih kriterija.

Kriterij pokrivenosti strukturnog modela

Kriteriji pokrivenosti strukturnog modela koriste strukturu modela. Primjerice strukturu automata ili dijagrama stanja čine čvorovi i lukovi. Kod modela sa zapisom *uvjet/posljedica* strukturu čine uvjetne naredbe.

Paradigma modeliranja često predlaže zasebnu vrstu kriterija strukturne pokrivenosti. Kao kriterij pokrivenosti kod modela sa zapisom *uvjet/posljedica* najčešće se koristi pokrivenost uzročno-posljedičnih veza. Kod modela zasnovanih na prijelazima, poput automata ili dijagrama stanja, postoji mnogo kriterija pokrivenosti grafa koji se koriste za upravljanje stvaranjem ispita. Najčešće korišteni kriteriji pokrivenosti su: svi čvorovi (tj. sva stanja), svi prijelazi, svi parovi prijelaza te svi ciklusi.

Postoji i skup kriterija strukturne pokrivenosti koji se koriste za donošenje složenih logičkih odluka u modelu. Takvi kriteriji u početku su korišteni za ispitivanje *pristupom bijele kutije*. Kasnije se većina tih kriterija preuzela i prilagodila za rad na modelima. Slično su se i mnogi kriteriji pokrivenosti toka podataka [38] za ispitivanje *pristupom bijele kutije* prilagodili za modele.

Kriterij pokrivenosti podataka

Kriteriji pokrivenosti podataka bave se načinom izbora nekoliko ispitnih vrijednosti iz velikog područja podataka. Osnovna ideja je podjela područja podataka na istovjetne razrede te odabir predstavnika iz svakog istovjetnog razreda. Elementi pojedinog razreda trebaju biti istovjetni u smislu njihove sposobnosti da otkriju pogrešku. Za uređene tipove podataka, navedeno dijeljenje prostora obično je dopunjeno sa odabirom dodatnih ispita iz graničnih područja intervala. Analize graničnih područja [39] te analize domena široko su prihvaćene kao heuristike otkrivanja grešaka te se koriste kao kriterij pokrivenosti za stvaranje ispita.

Kriterij pokrivenosti zasnovan na zahtjevima

Kada se elementi modela mogu eksplicitno povezati sa neformalnim zahtjevima ispitivanog sustava, pokrivenost se također može primijeniti na zahtjeve. Primjerice, neformalne oznake zahtjeva mogu se pridodati prijelazima UML dijagrama ili predikatima u skupu posljedica modela sa zapisom *uvjet/posljedica*.

Ad-hoc specifikacije ispitnih slučajeva

Izravne specifikacije ispitnih slučajeva koriste se za upravljanje stvaranja ispita. Pored modela, inženjer ispitivač piše i specifikacije ispitnih slučajeva za određivanje ispita koji se

stvaraju. Primjerice, specifikacije ispitnih slučajeva koriste se za ograničavanje putanja kroz model koje se ispituju, za fokusiranje ispitivanja na često korištene slučajeve ili za osiguravanje da će se određene putanje ispitati. Zapisi korišteni za izražavanje tih ciljeva ispitivanja mogu biti isti kao zapisi korišteni za model, ali mogu biti i različiti. Uobičajeni zapisi koji se koriste uključuju konačne automate, regularne izraze te Markovljeve lance.

Nasumični i stohastički kriterij

Nasumični i stohastički kriteriji najbolje se primjenjuju na modele čiji je predmet modeliranja okolina jer okolina određuje uzorke korištenja ispitivanog sustava. Vjerojatnosti akcija korištenja modeliraju se izravno ili neizravno [26]. Nakon izgrađenog modela, stvoreni ispiti prate očekivani profil korištenja.

Kriterij zasnovan na greškama

Kriterij zasnovan na greškama uglavnom se primjenjuje na modele čiji je predmet modeliranja ispitivani sustav jer je cilj pronaći greške u ispitivanom sustavu. Jedan od najčešće korištenih kriterija zasnovanih na greškama je pokrivenost mutacija. Takav pristup uključuje mutaciju modela, a potom stvaranje ispita koji će razlikovati mutirani model od originalnog modela. Pretpostavka je postojanje korelacije između grešaka u modelu i grešaka u ispitivanom sustavu te između mutacija i grešaka u stvarnom svijetu.

3.2.6. Tehnologija stvaranja ispita

Šesti kriterij razradbe pristupa ispitivanju je tehnologija koja se koristi tijekom stvaranja ispita. U mnogo slučajeva, modeli ispitivanog sustava svode se na *ručno stvaranje* ispitnih slučajeva (obično kod grafičkih modela koji se grade sofisticiranim CASE alatima). Jedna od glavnih karakteristika *ispitivanja zasnovanog na modelu* je mogućnost automatizacije stvaranja ispitnih slučajeva. Ispitni slučaj predstavljaju putanju u modelu koja označava ispitivani zahtjev sustava. Da bi se stvorili prikladni ispitni slučajevi, pretraživanje putanja mora biti upravljano. Tehnike koje upravljaju pretraživanjem putanja modela su nasumično stvaranje, algoritam pretraživanja namjenskih grafova, provjera modela, simboličko izvođenje i dokazivanje teorema.

Nasumično stvaranje ispita provodi se uzorkovanjem prostora ulaznih vrijednosti sustava. Svaka ulazna vrijednost ima jednaku vjerojatnost odabira. Nasumični prolazak kroz model može rezultirati skupinom ispitnih slučajeva različitih karakteristika, od kojih su neki manje a neki više važni. Kako kod složenih modela postoji jako puno putanja, toliko da se ne mogu

sve pokriti u realnom vremenu, postoji mogućnost da neki važni ispitni slučajevi budu izostavljeni.

Algoritam pretraživanja namjenskih grafova sadržava algoritme pokrivenosti čvorova ili lukova. Primjer takvog algoritma je *Chinese Postman algoritam* [37], koji pokriva svaki luk barem jednom.

Provjera modela je tehnika za provjeru svojstava sustava u modelu. Provjerom pojedinog svojstva sustava u modelu potvrđuje se ispravnost svojstva ili se daje protuprimjer. Potvrda ispravnosti očituje se putanjom modela kojom je svojstvo zadovoljeno, dok je protuprimjer predstavljen putanjom modela kojom svojstvo nije zadovoljeno. Glavna ideja stvaranja ispitnih slučajeva provjerom modela je formulirati *ad-hoc* specifikacije ispitnih slučajeva kao svojstva dohvatljivosti, npr. „konačno će doći u određeno stanje ili će se dogoditi određeni prijelaz“. Provjerom modela dobivaju se putanje kojima se dolazi u traženo stanje ili koje konačno uzrokuju prijelaz.

Ideja *simboličkog izvođenja* je izvođenje modela sa skupom ulaznih vrijednosti, umjesto sa jednom vrijednosti. Koriste se simboličke varijable umjesto konkretnih vrijednosti. Na taj način stvaraju se simboličke putanje modela, pri čemu se jedna simbolička putanja sastoji od više stvarnih putanja modela. Nakon stvorenih simboličkih putanja, simboličke varijable se popunjavaju stvarnim vrijednostima kako bi se dobile stvarne putanje, tj. ispitni slučajevi. Simboličko izvođenje popunjava simboličke varijable vrijednostima iz *ad-hoc* specifikacija ispitnih slučajeva, koje uobičajeno sadrže granične vrijednosti.

Na kraju, stvaranje ispita provodi se i korištenjem tehnike *dokazivanja teorema*. Ako se model sustava može pretvoriti u skup logičkih formula, tada se iz njih mogu izvoditi teoremi koji predstavljaju određeno ponašanje sustava. Ispitni slučajevi stvaraju se kao dio postupka dokazivanja teorema koje predstavlja provjeru ispravnosti određenog zahtjeva (tj. ponašanja) sustava. Dokazivanje teorema uspješno se koristi kod modela sa zapisom *uvjet/posljedica*.

3.2.7. Vrijeme stvaranja ispita

Posljednji kriterij razradbe pristupa *ispitivanju zasnovanom na modelu* vezan je uz relativno vrijeme stvaranja i izvođenja ispitnih slučajeva.

Kod stvaranja ispita tijekom izvođenja (engl. *on line testing*), algoritmi stvaranja ispita reagiraju na stvarne izlaze ispitivanog sustava. Ispitni slučajevi se jedan po jedan stvaraju i odmah izvode. To je potrebno ako je ispitivani sustav nedeterministički jer generator ispita

vidi koju putanju je ispitivani sustav uzeo pa može primijeniti istu putanju i u modelu. Stvaranje ispita prije izvođenja (engl. *off line testing*) označava da se čitav skup ispitnih slučajeva stvara striktno prije nego se izvodi. Takvo stvaranje ispita iz nedeterminističkog modela je često iznimno teško te uključuje stvaranje ispitnih slučajeva u obliku stabla i grafova, a ne linearnih sljedova.

Prednosti stvaranja ispita prije izvođenja, kada se ono može primijeniti, uglavnom su praktične. Stvoreni ispiti mogu se upravljati i izvoditi koristeći postojeće alate za upravljanje ispitivanjem, što znači da je potrebno manje promjena u procesu ispitivanja. Prednost je i mogućnost stvaranja skupa ispita odjednom te potom izvođenja više puta na ispitivanom sustavu. Također, stvaranje ispita i izvođenje ispita mogu se obaviti na različitim računalima ili u različitim okolinama, kao i u različito vrijeme. Konačno, ako je proces stvaranja ispita sporiji od procesa izvođenja ispita, očite su prednosti obavljanja koraka stvaranja ispita samo jednom.

3.3. Razradba alata za ispitivanje zasnovano na modelu

Različiti pristupi *ispitivanju zasnovanom na modelu* ostvareni su u različitim alatima koji podupiru ispitivanje zasnovano na modelu. Razmatraju se najpopularniji alati: TorX [29], LTG [55], Matlab Simulink V&V [56], JUMBL [57], AETG [58] i CTG [6], [7], [8]. Navedeni alati su reprezentativni primjerci najpopularnijih pristupa *ispitivanju zasnovanom na modelu* prema razradbi opisanoj u odjeljku 3.2. Također se pokazuje za koja područja primjene su alati prikladni.

3.3.1. TorX

Sustav TorX primjer je automatiziranog stvaranja ispita na osnovu ponašajnog modela ispitivanog sustava. TorX je akademski alat za *ispitivanje zasnovano na modelu* razvijeno krajem devedesetih godina 20. stoljeća. Svrha TorX-a je ostvarenje teorije ispitivanja usklađenosti između modela i programskih ostvarenja [30]. TorX je reprezentativni primjerak alata za stvaranje ispita zasnovanih na modelu ostvarenom automatima. Sustav TorX upravlja nedeterminizmom te pruža mogućnost stvaranja ispita tijekom izvođenja. Odabir ispita TorX obavlja prema *ad-hoc* specifikacijama ispitnih slučajeva. Ispiti se stvaraju nasumičnim kretanjem putanjama modela uz ograničenja definirana specifikacijama ispitnih slučajeva..

Alati sa sličnim pristupom su TGV [31], STG [32] te AutoLink [33]. Uobičajena područja primjene tih alata su telekomunikacijski i protokolni sustavi.

Predmet modeliranja: S obzirom na predmet modeliranja, opisuje se ponašanje ispitivanog sustava. Međutim, uzimaju se u obzir i neki aspekti okoline.

Razina zalihosti modela: Studije izvodivosti (engl. *case studies)* sustava TorX pokazuju zalihost modela korištenjem odvojenog modela za stvaranje ispita.

Karakteristike modela: Sustav TorX pruža upravljanje nedeterminističkim, vremenski neograničenim, diskretnim modelima.

Paradigma modeliranja: TorX koristi model zasnovan na automatima.

Kriterij odabira ispita: Kao kriterij odabira ispita TorX koristi ad-hoc specifikacije ispitnih slučajeva.

Tehnologija stvaranja ispita: Automatizirano stvaranje ispitnih slučajeva koristeći pretraživanje putanja modela koje nije posve nasumično, već ograničeno *ad-hoc* specifikacijama ispitnih slučajeva.

Vrijeme stvaranja ispita: Alat Torx podržava stvaranje ispita prije i tijekom izvođenja.

3.3.2. LTG

Alat LTG (engl. *Leirios Test Generator*) tvrtke *Leirios Technologies*, komercijalno je dostupan alat za *ispitivanje zasnovano na modelu*. Ispitni slučajevi stvaraju su na osnovu ponašajnog modela ispitivanog sustava koristeći pokrivenost modela kao kriterij odabira ispita. Alat LTG prima dvije vrste ulaznog zapisa: UML dijagrami te modeli zasnovani na zapisu *uvjet/posljedica*.

Također su korišteni i sljedeći dostupni alati sa sličnim pristupom ispitivanju: T-VEC Tester za Simulink [59] i Reactis tvrtke Reactive Systems [60]. Uobičajena područja primjene tih sustava su reaktivni sustavi, ugrađeni sustavi, pametne kartice ili e-transakcijski primjenski sustavi.

Predmet modeliranja: S obzirom na predmet modeliranja, opisuje se ponašanje ispitivanog sustava.

Razina zalihosti modela: Koristi se odvojen model za stvaranje ispita.

Karakteristike modela: Modeli moraju biti deterministički, vremenski neograničeni i diskretni.

Paradigma modeliranja: Alat LTG podržava UML dijagrame te modeliranje zasnovano na zapisu *uvjet/posljedica* (poput B apstraktnih automata).

Kriterij odabira ispita: Podržano je nekoliko kriterija pokrivenosti strukturnog modela. Za UML modele kriterij uključuje pokrivenost stanja, svih prijelaza, svih proširenih prijelaza te svih parova prijelaza. Za modele sa zapisom *uvjet/posljedica* također je podržano nekoliko vrsta pokrivenosti (npr. pokrivenost svih posljedica ili svih parova posljedica). U oba slučaja, složeni uvjeti u sklopu uvjeta prijelaza (engl. *guard*) ili predikata (koji na osnovu uvjeta omogućuje određenu akciju) detaljnije se ispituju korištenjem kriterija strukturne pokrivenosti (npr. MC/DC). Vrijednosti podataka mogu se izabrati na osnovu kriterija pokrivenosti podataka: jedna ili više vrijednosti, slučajne vrijednosti, granične vrijednosti ili sve vrijednosti.

Tehnologija stvaranja ispita: Automatizirano stvaranje ispitnih slučajeva koristi simboličko izvođenje zasnovano na ograničavanju modela te algoritme pretraživanja. Stvoreni apstraktni ispitni slučajevi se potom prevode u izvršne ispitne skripte koristeći prilagodnik poseban za ispitnu okolinu.

Vrijeme stvaranja ispita: LTG podržava stvaranje ispita prije i tijekom izvođenja.

3.3.3. Matlab Simulink V&V

Svrha Simulinkovog modula za verifikaciju i validaciju je ispitivanje kontinuiranih i hibridnih Simulinkovih modela. Glavna funkcionalnost Simulinkovog modula V&V je mogućnost praćenja od zahtjeva sustava do Simulinkovih modela te analiza pokrivenosti modela.

Predmet modeliranja: Simulinkovi modeli specificiraju namijenjeno ponašanje ispitivanog sustava, kao i ispitne okoline.

Razina zalihosti modela: Simulinkovi modeli obično se koriste za stvaranje ispitnih slučajeva i programskog koda.

Karakteristike modela: Simulink podržava determinističke i nedeterminističke modele, vremenski ograničene modele sa kontinuiranom funkcijom i tipovima podataka. Podržani su i hibridni sustavi.

Paradigma modeliranja: Simulink koristi paradigmu toka podataka (Simulinkovi funkcijski blokovi).

Kriterij odabira ispita: Simulinkovi blokovi zasnivaju se na namjenskom kriteriju poput pokrivenosti preglednih tablica (engl. *lookup table*), koje snimaju učestalost pretraživanja tablice u bloku. Podržani su i ostali kriteriji strukturne pokrivenosti za pokrivenost podataka (granične vrijednosti, analize raspona signala) te složene logičke odluke (pokrivenost odluka, pokrivenost uvjeta, MC/DC).

Tehnologija stvaranja ispita: Ispiti se izvode ručno, a potom se podvrgavaju automatiziranoj analizi pokrivenosti na razini modela.

Vrijeme stvaranja ispita: Ispitni slučajevi izvode se na samom modelu. Da bi se ispiti izvodili na ispitivanom sustavu, prvo se moraju snimiti ispiti, a potom prilagoditi sučelju ispitivanog sustava. Dakle, radi se o pristupu stvaranju ispita prije izvođenja.

3.3.4. JUMBL

JUMBL (engl. *Java Usage Model Builder Library*) je akademski alat za statističko *ispitivanje zasnovano na modelu*, razvijeno na američkom sveučilištu *Universtity of Tennessee*. Alat JUMBL podržava razvoj modela korištenja (engl. *usage model*) zasnovan na Markovljevim lancima. Model korištenja zasnovan na Markovljevim lancima ima jedinstveno početno stanje, jedinstveno konačno stanje, skup prijelaznih stanja korištenja te prijelazne lukove između stanja. Prijelazni lukovi označeni su odgovarajućim događajem te vjerojatnošću pojavljivanja. Vjerojatnosti prijelaza zasnivaju se na očekivanom korištenju ispitivanog sustava. Ispitni ulaz stvara se obilaskom modela na osnovu vjerojatnosti prijelaznih lukova. Prema tome, ispitni slučajevi sa najvećom vjerojatnosti stvaraju se prvi. Model zasnovan na Markovljevim lancima ne pruža očekivani odgovor ispitivanog sustava.

Poznatiji alati sa sličnim pristupom ispitivanju su Matelo tvrtke ALL4TEC [61] i CleanTest tvrtke CleanSoft [62].

Predmet modeliranja: Koristi se model očekivane okoline.

Razina zalihosti modela: JUMBL koristi odvojen model za stvaranje ispita.

Karakteristike modela: Modeli su vremenski neograničeni i diskretni. Izbor između determinizma i nedeterminizma nije bitan jer se samo stvaraju ispitni ulazi i ne modelira se ponašanje ispitivanog sustava.

Paradigma modeliranja: Modeli alata JUMBL grade se jezikom TML, koji služi za stohastički zapis opisa modela korištenja zasnovanih na Markovljevim lancima.

Kriterij odabira ispita: Statističko ispitivanje pruža nasumični i statistički kriterij (zasnovan na vjerojatnosti prijelaznog luka modela korištenja).

Tehnologija stvaranja ispita: Automatizirano stvaranje ispitnih ulaza koristeći statističke algoritme pretraživanja.

Vrijeme stvaranja ispita: Stvoreni ispitni slučajevi moraju se prevesti u skriptni jezik okoline izvođenja ispita (ili mogu biti izvršeni ručno). Alat JUMBL koristi pristup stvaranju ispita prije izvođenja i pruža API (engl. *Application Programming Interface*) za povezivanje sa okolinama izvršavanja ispita.

3.3.5. AETG

U kombinatornom ispitivanju (engl. *combinatorial testing*) svrha je smanjiti broj mogućih kombinacija ulaznih varijabli (kojih je u praksi jako puno) na nekoliko reprezentativnih. AETG (engl. *Automatic Efficiant Test Generator*) je generator ispitnih ulaza zasnovan na modelu za kombinatorno ispitivanje. Da bi smanjio broj ispitnih podataka, generator AETG koristi algoritam uparivanja (engl. *pair-wise algorithm*). Za svaki par mogućih ulaznih parametara u sustav, algoritmom uparivanja ispituju se sve moguće diskretne kombinacije tih ulaznih parametara. Korištenjem pažljivo izabranih ispitnih slučajeva, ispitivanje postaje mnogo brže nego prilikom iscrpnog pretraživanja svih mogućih kombinacija svih ulaznih parametara. Smisao ispitivanja zasnovanog na algoritmu uparivanja zasniva se na pretpostavci da su najjednostavnije greške u programu općenito uzrokovane jednim ulaznim parametrom. Nešto manje jednostavna kategorija grešaka uzrokovana je međudjelovanjem dvaju ulaznih parametara (što se otkriva algoritmom uparivanja). Greške uzrokovane međudjelovanjem tri ili više ulaznih parametara su rjeđe, ali istodobno i vremenski zahtjevnije za otkriti. Generator AETG također podržava ispitivanje svih trojki (engl. *all-triples*) ili svih četvorki (engl. *all-quadruples*), ali tada veličina stvorenog skupa ispita brzo raste. Izvor za svaki ulazni ispit mora biti zadan ručno.

Pored alata AETG, postoji mnogo alata namijenjenih ispitivanju zasnovanom na uparivanju [63]. Uobičajena područja primjene za ovakav pristup je ispitivanje različitih konfiguracija ili kombinacija uređaja.

Predmet modeliranja: Ispitivanje zasnovano na uparivanju koristi jednostavni statički model ulaznih podataka ispitivanog sustava, definirajući domene varijabli i sve nedopuštene kombinacije vrijednosti. Dakle, koristi se model okoline.

Razina zalihosti modela: Odvojen model za stvaranje ispitnih ulaza.

Karakteristike modela: Modeli su vremenski neograničeni i diskretni. Izbor između determinizma i nedeterminizma nije bitan jer generator AETG modelira samo ispitne ulaze, a ne ponašanje ispitivanog sustava.

Paradigma modeliranja: Obično nema modeliranja ponašanja, već samo modeliranje domene statičkih podataka pa se koristi model za kombinatorno ispitivanje.

Kriterij odabira ispita: Razred alata čiji predstavnik je AETG koristi pokrivenost svih parova kao kriterij pokrivenosti podataka.

Tehnologija stvaranja ispita: Automatizirano stvaranje ispitnih ulaza koristeći algoritme pretraživanja u *n* smjerova (engl. *n-way search algorithm*).

Vrijeme stvaranja ispita: Koristi se pristup stvaranju ispita prije izvođenja.

3.3.6. CTG

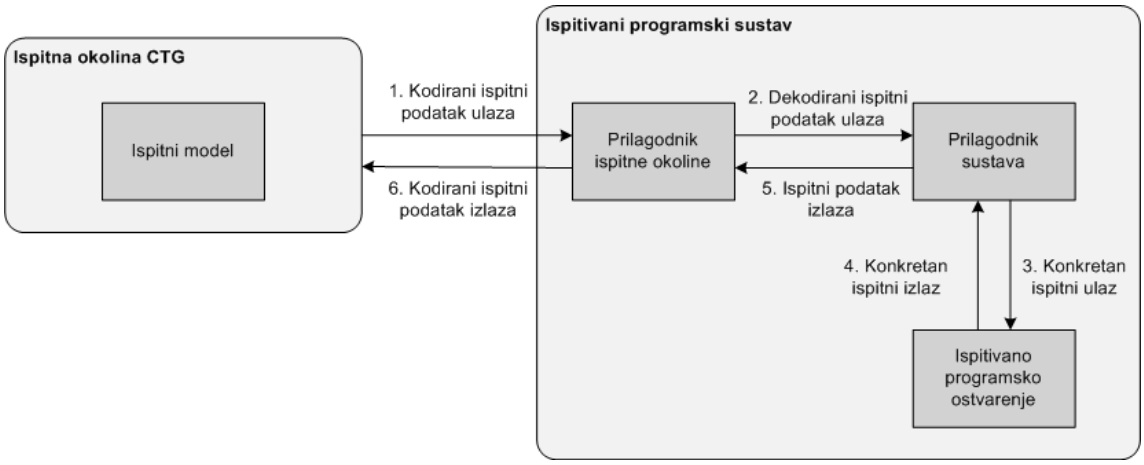
Alat CTG tvrtke *Conformiq* korišten je tijekom izrade praktično dijela diplomskog rada. CTG pruža mogućnost stvaranja velikih količina ispita iz modela zasnovanih na UML dijagramima i izvođenja ispita na ispitivanom sustavu. UML standard ne definira tekstualni zapis za akcije vezane uz prijelaze i stanja dijagrama ispitnog modela, te zato alat CTG definira zaseban proceduralni programski jezik nazvan *CTG akcijski jezik*. Glavna namjena *CTG akcijskog jezika* je definiranje akcija koje su pridijeljene prijelazima i stanjima UML modela. *CTG akcijski jezik* sastoji se od dva glavna dijela: zapisa koji opisuju prijelaze i definicija. Zapisi koji opisuju prijelaze sastoje se od događaja, uvjeta prijelaza i akcija. Događaji su opisani tipom podataka koji se zove *record*, dok su uvjeti prijelaza opisani logičkim izrazima. Akcije su liste naredbi koje sadrže akcijske naredbe i izraze. Definicije sadržavaju definicije potprograma, deklaracije varijabli i deklaracije konstanti.

Alat CTG pruža ispitnu okolinu koja je spojena sa ispitivanim sustavom (engl. *System Under Test*, skraćeno SUT) preko prilagodnika ispitne okoline (engl. *testbed adapter*) i prilagodnika sustava (engl. *system adapter*). Prilagodnik ispitne okoline i prilagodnik sustava su programske komponente koje ulazne ispite iz ispitne okoline pretvaraju u oblik koji je

razumljiv ispitivanom sustavu. Prilagodnik ispitne okoline i prilagodnik sustava dio su istog računalnog procesa, a ispitivani sustav obično radi u odvojenom procesu (iako nije nužno). Obično su povezani preko mreže zasnovane na protokolu IP (engl. *Internet Protocol*) i komuniciraju koristeći poseban protokol primjenske razine iznad protokola TCP (engl. *Transmission Control Protocol*). Standardna rješenja poput CORBA (engl. *Common Object Request Broker Architecture*) ili XML RPC (engl. *eXtensible Markup Language Remote Procedure Call*) ne koriste se za komunikaciju između generatora ispita i ispitivanog sustava jer ti protokoli zahtijevaju više sredstava nego što je dostupno u nekim sustavima koji se ispituju.

Mehanizam za određivanje ispravnosti ispitnih izlaza (engl. *test oracle*) ostvaruje matematičke algoritme koji upravljaju stvaranjem ispitnih slučajeva tijekom izvođenja. Ispitivanje je nezavisno od načina ostvarenja komunikacije sa ispitivanim sustavom.

Slika 3.12 prikazuje uobičajenu ispitnu konfiguraciju tijekom ispitivanja sa pristupom stvaranju ispitnih slučajeva tijekom izvođenja.



Slika 3.12 Konfiguracija ispitivanja sa pristupom stvaranju ispitnih slučajeva tijekom izvođenja

Alat CTG upravlja izvođenjem ispita tako da vrednuje ispitni scenarij tijekom izvođenja sustava i šalje podražaj ispitivanom programskom ostvarenju (engl. *Implementation Under Test*, skraćeno IUT) putem ispitnog podatka (engl. *datum)*. *Ispitni podatak* je podatkovni element koji predstavlja protokolnu poruku ili funkciju koja se treba pozvati, ovisno o ispitivanom programskom ostvarenju. U slučaju odgovora, *ispitni podatak* predstavlja ispitni izlaz programskog ostvarenja (kao odgovor na *ispitni podatak* ulaza) u obliku razumljivom ispitnoj okolini. *Ispitni podaci* kodiraju se binarno i prenose putem komunikacijskog medija

(TCP/IP) prilagodniku ispitne okoline. Prilagodnik ispitne okoline prikriva sve detalje kodiranja *ispitnog podatka* i protokola između sebe i ispitne okoline. Također pruža i čvrsto sučelje preko kojeg prilagodnik sustava komunicira sa ispitnom okolinom. Prilagodnik sustava, nakon primljenog *ispitnog podatka* ulaza, pretvara *ispitni podatak* u konkretan ispitni ulaz kojeg prepoznaje ispitivano programsko ostvarenje. Ovisno o ispitivanom programskom ostvarenju, ulaz može biti protokolna poruka, funkcijski poziv, niz znakova jezika SQL, poruka operacijskog sustava i slično. Vrsta pretvorbe zavisna je o ispitivanom sustavu. Zbog toga je potreban različiti prilagodnik sustava za različita ostvarenja ispitivanog sustava. Iz navedenih razloga, alat CTG može komunicirati samo sa jednim ispitivanim sustavom tijekom jedne sjednice ispitivanja.

Koraci opisani na slici 3.12 obrnuti su kada ispitivano programsko ostvarenje odgovara na podražaj od ispitne okoline. U tom slučaju ispitivano programsko ostvarenje šalje ispitni izlaz prilagodniku sustava. Prilagodnik sustava vrši pretvorbu konkretnog ispitnog izlaza u *ispitni podatak* izlaza kojeg šalje prilagodniku ispitne okoline. Nakon kodiranja u binarni prikaz, *ispitni podatak* izlaza putuje od prilagodnika ispitne okoline do ispitne okoline gdje alat CTG vrši rekonstrukciju *ispitnog podatka.* Nakon primljenog odgovora, ispitna okolina odlučuje je li odgovor bio ispravan ili ne.

Prilagodnik ispitne okoline, prilagodnik sustava i ispitivano programsko ostvarenje zajedno se nazivaju ispitivani sustav.

Predmet modeliranja: Ispitni modeli u CTG-u opisuju ponašanje ispitivanog sustava.

Razina zalihosti modela: Koristi se odvojen model za stvaranje ispita.

Karakteristike modela: CTG pruža potporu nedeterminizmu u modelu. Modeli moraju biti vremenski neograničeni i diskretni.

Paradigma modeliranja: Alat CTG podržava UML dijagrame, a dodatno pruža i *CTG akcijski jezik* za definiranje akcija vezanih uz prijelaze i stanja.

Kriterij odabira ispita: Podržano je četiri kriterija pokrivenosti strukturnog modela. Kriteriji uključuje pokrivenost svih stanja, pokrivenost svih prijelaza, pokrivenost raspodjela (engl. *partition criterion*) i pokrivenost strukturnih ulaza/izlaza. Detaljnija analiza pojedinih kriterija odabira ispita iznesena je u odjeljku 5.3.3.

Tehnologija stvaranja ispita: Automatizirano stvaranje ispitnih slučajeva koristi algoritme pretraživanja. Pretraživanje ispitnog modela provodi se nasumično ili optimirano. Ako je

odabrano optimirano pretraživanja, koriste se tehnike pretraživanja unaprijed (engl. *forward search*) i stohastičkog planiranja. Stvoreni apstraktni ispitni slučajevi prevode se u izvršne ispitne skripte koristeći prilagodnik za ispitnu okolinu.

Vrijeme stvaranja ispita: CTG podržava stvaranje ispita prije i tijekom izvođenja.

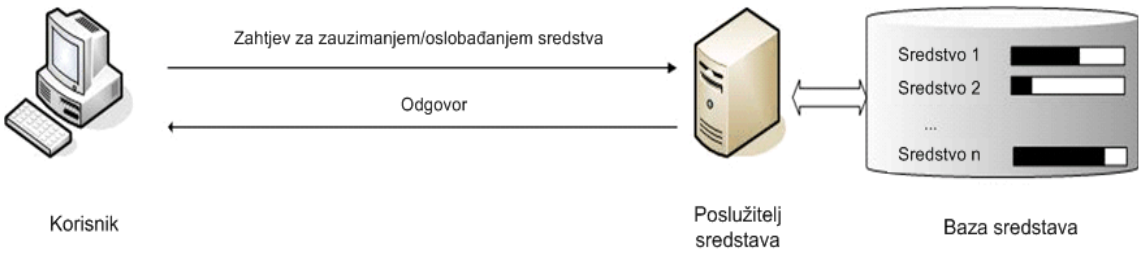
4. Ispitivanje jednostavne korisnik-poslužitelj okoline ispitnim modelom

U prethodnim poglavljima opisana je teorija *ispitivanja sustava modelom*. Ovo poglavlje daje primjer izvođenja ispitivanja jednostavne korisnik-poslužitelj okoline ispitnim modelom. Svrha ovog primjera je lakše shvaćanje koncepta *ispitivanja zasnovanog na modelu* s ciljem razumijevanja ispitivanja sigurnosnog podsustava okoline *PIE* ispitnim modelom, opisanog u sljedećem poglavlju. Ispitivanje se izvodi uz pomoć alata CTG, koji je korišten i u ispitivanju sigurnosnog podsustava okoline *PIE*.

Alat CTG omogućava izradu ispitnog modela te pruža ispitnu okolinu za izvođenje ispitivanja rada poslužitelja. U ispitivanoj korisnik-poslužitelj okolini, korisnik je predstavljen ispitnim modelom kojim se ispituje rad poslužitelja. Rad poslužitelja ispituje se na dvije razine. Prvo se gradi i verificira model oblikovanja poslužitelja, a potom se vrši ispitivanje programskog ostvarenja poslužitelja.

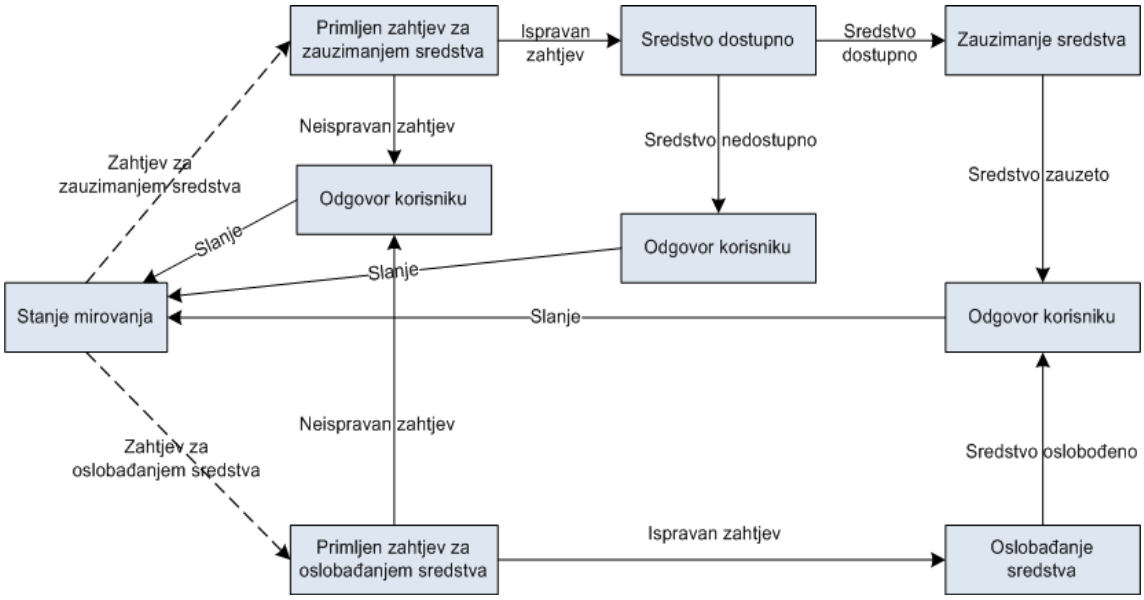
4.1. Opis funkcionalnosti ispitivane korisnik-poslužitelj okoline

Ispitivana korisnik-poslužitelj okolina sastoji se od korisnika koji poslužitelju šalje zahtjeve za zauzimanjem ili oslobađanjem sredstava te poslužitelja koji sadrži bazu sredstava. Svako sredstvo sastoji se od unaprijed definiranog broja jedinica pa korisnik zauzima ili oslobađa određen broj jedinica pojedinog svojstva. Opisani korisnik-poslužitelj sustav prikazan je slikom 4.1.



Slika 4.1 Opis korisnik-poslužitelj okoline

Ponašanje poslužitelja opisano je modelom na slici 4.2.



Slika 4.2 Model ponašanja poslužitelja

Isprekidane strelice na slici predstavljaju vanjske pozive korisnika, dok pune strelice označavaju akcije poslužitelja. Poslužitelj je u stanju mirovanja do pojave zahtjeva korisnika.

Ako korisnik zahtijeva zauzimanje sredstva, poslužitelj prvo provjerava ispravnost zahtjeva i dostupnost sredstva. Ukoliko zahtjev nije sintaksno ispravan ili traženo sredstvo ne postoji, korisniku se šalje odgovarajuća poruka o neispravnom zahtjevu. Ako je zahtjev ispravan i sredstvo dostupno, provodi se zauzimanje traženog broja jedinica traženog sredstva. U slučaju da poslužitelj nema traženi broj jedinica sredstva slobodno, korisniku se šalje odgovarajuća poruka o neuspjelom zauzimanju sredstva. U suprotnom, uspješno se obavlja zauzimanje sredstva i dojavljuje korisniku.

Ako korisnik zahtijeva oslobađanje sredstva, poslužitelj najprije provjerava ispravnost zahtjeva. Ukoliko zahtjev nije sintaksno ispravan, korisniku se šalje poruka o neispravnom zahtjevu. U slučaju ispravnog zahtjeva, pokušava se osloboditi zahtijevani broj jedinica sredstva. Ako se želi osloboditi više jedinica traženog sredstva od onoliko koliko ih je zauzeto, tada se korisniku šalje odgovor o neuspješnom oslobađanju. U suprotnom, korisniku se šalje odgovor o uspješnom oslobađanju sredstva.

Na osnovu opisanog ponašanja poslužitelja, definira se protokol komunikacije korisnik-poslužitelj. Protokol se definira zapisom ABNF (engl. *Augmented Backus-Naur Form*) prema specifikaciji RFC 4234 :

```
CRLF = %d13.10   ; \r\n
DIGIT = “0” / “1” / “2” / “3” / “4” / “5” / “6” / “7” / “8” / “9”   ; [0-9]
Number = 1*DIGIT   ; [0-9]+
Message = Request / Response
Request = Type Resource Quantity CRLF
Type = “Reserve” | “Release”
Resource = Number
Quantity = Number
Response = “Invalid request” / “Resource unavailable” / “OK”
```

Ključnom riječi “Request” definira se zahtjev korisnika, dok se ključna riječ “Response” predstavlja odgovor poslužitelja. Zahtjev za zauzimanjem sredstva definiran je ključnom riječi “Reserve”, dok je zahtjev za oslobađanjem sredstva definiran sa “Release”. Redni broj sredstva i broj jedinica definiraju se znamenkama dekadskog sustava. Odgovor korisniku dojavljuje se sa jednom od sljedećih poruka: “Invalid request”, “Resource unavailable” ili “OK”. Ako se zauzimanje ili oslobađanje sredstva uspješno obavila, poslužitelj odgovara korisniku sa “OK”. U slučaju da korisnik traži zauzimanje sredstva kojeg nema u poslužiteljevoj bazi sredstava, poslužitelj odgovara sa “Resource unavailable”. Odgovor “Invalid request” moguć je ako traženo sredstvo ne postoji ili zahtjev korisnika nije sintaksno ispravan ili ako je zatraženo oslobađanje više jedinica traženog sredstva nego što ih je zauzeto.

Na primjer, ako poslužitelj ima 10 sredstava i svako sredstvo ima 10 slobodnih jedinica, komunikacija između korisnika i poslužitelja može se odvijati na sljedeći način:

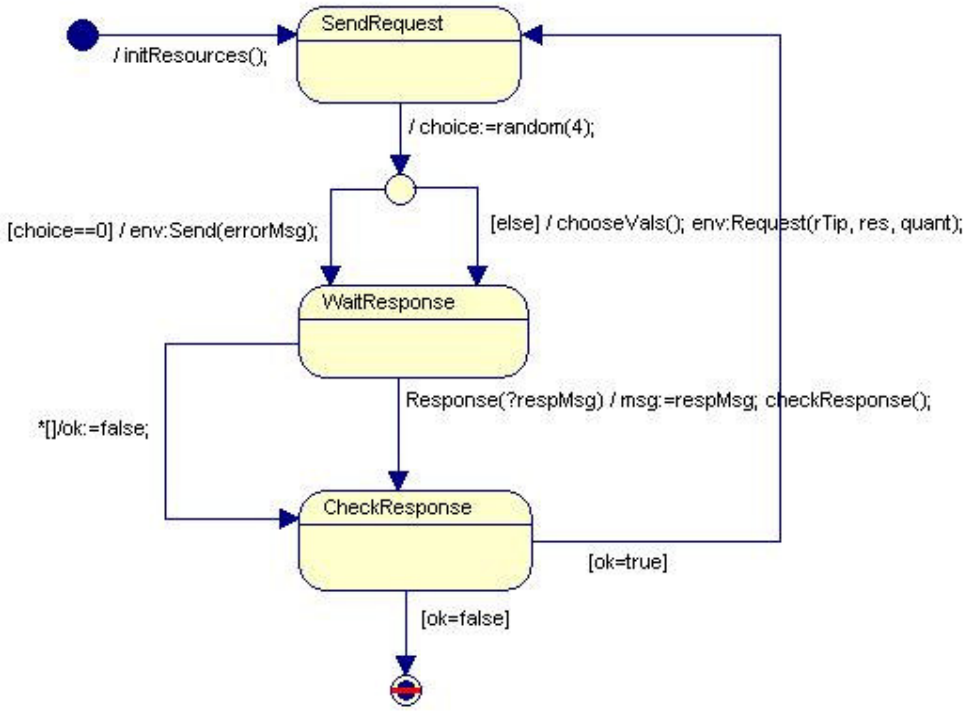
```
→ Reserve 2 9 \r\n
← OK

→ Reserve 2 2 \r\n
← Resource unavailable

→ Release 2 5 \r\n
← OK

→ Reserve 2 2 \r\n
← OK
```

U navedenom primjeru, korisnik zahtijeva zauzimanje 9 jedinica sredstva „*2*“. Kako početno ima 10 slobodnih jedinica svakog sredstva, poslužitelj odgovara porukom “OK” koja označava da je dopustio korisniku traženo zauzimanje sredstva. Nakon nekog vremena korisnik zahtijeva 2 jedinice sredstva „*2*“. Međutim, ovaj put poslužitelj odgovara porukom da nema dovoljno raspoloživih jedinica tog sredstva. Sljedećim zahtjevom, korisnik traži oslobađanje 5 jedinica sredstva „*2*“, na što poslužitelj odgovara potvrdno. U ovom trenutku poslužitelj ima 7 slobodnih jedinica sredstva „*2*“, zbog čega omogućava zauzimanje 2 jedinice tog sredstva u idućem zahtjevu korisnika.

4.2. Ispitni model

Ispitni model predstavlja moguća ponašanja korisnika, te služi za ispitivanje ispravnosti rada poslužitelja. Ispitni model dio je ispitne okoline podržane alatom CTG te se koristi za verifikaciju modela oblikovanja poslužitelja i za ispitivanje programskog ostvarenja poslužitelja. Model oblikovanja poslužitelja također se gradi alatom CTG te se komunikacija ispitnog modela i modela oblikovanja ostvaruje mehanizmima unutar alata CTG. S druge strane, komunikacija sa programskim ostvarenjem poslužitelja ostvaruje se putem mreže protokolom TCP/IP.

Ispitni model prikazan je na slici 4.3. Blokovi pravokutnog oblika zaobljenih vrhova na ispitnom modelu predstavljaju stanja. Početno stanje označeno je kružićem ispunjenim plavom bojom. Strelice predstavljaju prijelaze, popraćene naredbama *CTG akcijskog jezika*. Neke naredbe kod prijelaza predstavljaju pozive procedura opisane u zasebnoj datoteci. Izraz u uglatim zagradama predstavlja uvjet prijelaza. Ako je uvjet u uglatim zagradama zadovoljen, prijelaz je omogućen.



Slika 4.3 Ispitni model

Ispitivanje počinje inicijalizacijom sredstava opisanom procedurom *initResources*. Potom se prelazi u stanje *SendRequest*, označavajući da slijedi korisničko slanje zahtjeva poslužitelju. Kako bi se što realnije simuliralo ponašanje korisnika, četvrtina zahtjeva oblikuje se neispravno. Ispravno oblikovani zahtjevi sadrže nasumične podatke o broju traženih jedinica te rednom broju sredstva stvorene procedurom *chooseVals*.

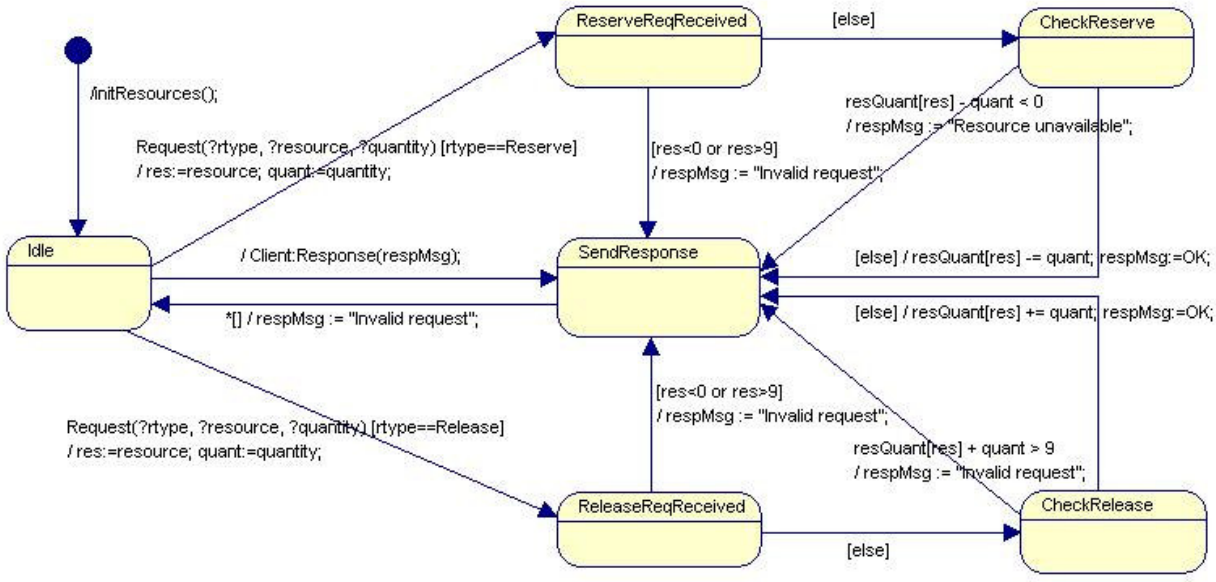
Slanje poruke poslužitelju postiže se naredbom *env:Request(rTip, res, quant)*. Ključna riječ *env* označava da se poruka šalje scenariju okoline (engl. *environment scenario*). Scenarij okoline je postavka u alatu CTG koja označava ispitivano programsko ostvarenje ili model oblikovanja koji se verificira. Pritom se ispitni model označava kao glavni scenarij (engl. *main scenario*). Izraz *Request(rTip, res, quant)* označava strukturu podataka i konkretne vrijednosti koje se šalju ispitivanom poslužitelju.

Nakon slanja poruke poslužitelju, prelazi se u stanje *WaitResponse*, u kojem se čeka odgovor od poslužitelja. Ako poslužitelj odgovori sa ispravnim tipom podatka, procedurom *checkResponse* provjerava se je li odgovor poslužitelja jednak očekivanom. Ako je poslužitelj odgovorio neispravnom porukom ili je odgovor poslužitelja drukčiji od očekivanog ispitivanje

završava odlukom *FAIL*, što znači da je nađena greška u radu poslužitelja. U suprotnom, ispitivanje se nastavlja dok se ne dosegne uvjet zaustavljanja (npr. 100% pokrivenost prostora mogućih ispitnih slučajeva). Ako se tijekom ispitivanja nijednom ne donese odluka *FAIL*, konačna odluka je *PASS*. U slučaju verifikacije modela oblikovanja poslužitelja, odluka *PASS* označava da se model sustava ponaša u skladu sa zahtjevima sustava. U slučaju ispitivanja programskog ostvarenja poslužitelja, odluka *PASS* označava da se stvarni sustav ponaša u skladu sa zahtjevima.

4.3. Verifikacija modela oblikovanja poslužitelja

Alat CTG pruža mogućnost verifikacije modela sustava. Model oblikovanja poslužitelja gradi se alatom CTG, slično kao i ispitni model. Da bi se alatu CTG naznačilo da se želi verificirati model oblikovanja sustava, tada se taj model mora u postavkama alata označiti kao scenarij okoline (engl. *environment scenario*). Model oblikovanja poslužitelja (slika 4.4), izgrađen alatom CTG, zasniva se na modelu ponašanja poslužitelja sa slike 4.2. Model oblikovanja sastoji se od sedam stanja označenih pravokutnim blokovima zaobljenih vrhova te početnog stanja označenog popunjenim kružićem. Strelice predstavljaju prijelaze među stanjima. Prijelazi su opisani *CTG akcijskim jezikom*.



Slika 4.4 Model oblikovanja (poslužitelj)

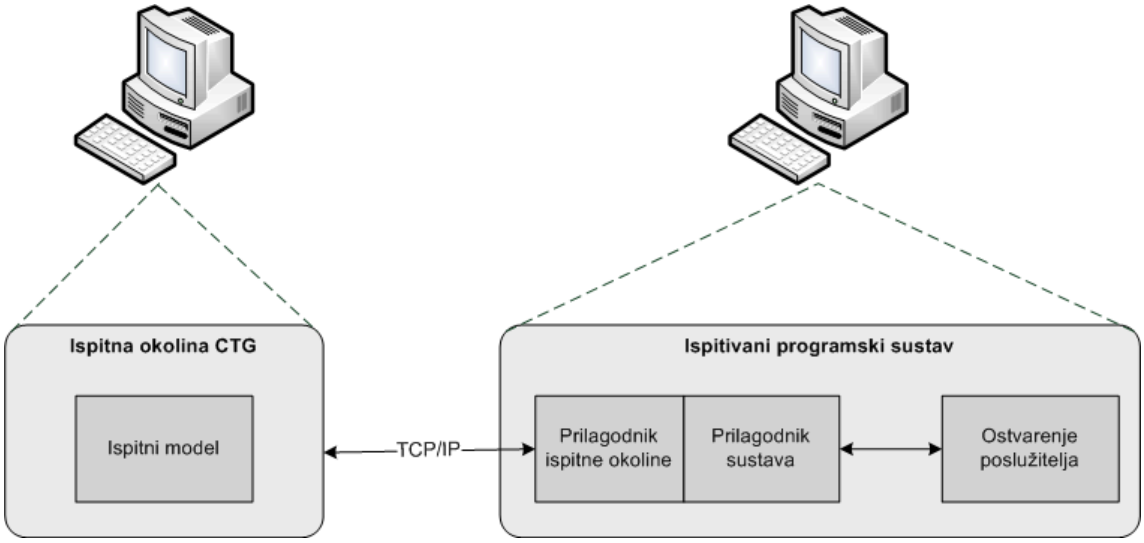
Pri prijelazu iz početnog stanja u stanje mirovanja *Idle* postavljaju se početne vrijednosti baze sredstava (broj sredstava i broj jedinica pojedinog sredstva). U stanju *Idle* poslužitelj čeka na zahtjev korisnika. U stanje *ReserveReqReceived* prelazi se ako dođe zahtjev za zauzimanjem sredstva, a u stanje *ReleaseReqReceived* ako dođe zahtjev za oslobađanjem sredstva. Ako su zahtjevi ispravni i postoji traženo sredstvo, tada se iz stanja *ReserveReqReceived* prelazi u stanje *CheckReserve*, odnosno (u slučaju oslobađanja sredstva) iz stanja *ReleaseReqReceived* u stanje *CheckRelease*. Ako je zahtjev bio sintaksno neispravan ili traženo sredstvo ne postoji u bazi sredstava poslužitelja, prelazi se u stanje *SendResponse*. Stanje *SendResponse* šalje odgovarajuću poruku korisniku, ovisno o ispravnosti zahtjeva te uspješnosti zahtijevane akcije. Stanja *CheckReserve* i *CheckRelease* provjeravaju je li moguće obaviti traženo zauzimanje, odnosno oslobađanje sredstva. Nakon obavljene provjere, prelazi se u stanje *SendResponse* i korisniku šalje odgovor o uspješnosti zahtijevane akcije.

Obavljena verifikacija pokazala je da se model poslužitelja ponaša u skladu sa zahtjevima.

4.4. Ispitivanje programskog ostvarenja poslužitelja

Poslužitelj je programski ostvaren jezikom C. Ostvarenje poslužitelja ne mora se nalaziti na istom računalu na kojem se nalazi ispitna okolina pa se komunikacija ispitivanog programskog ostvarenja i ispitne okoline odvija mrežnim protokolima TCP/IP.

Ispitna okolina i ispitivano programsko ostvarenje poslužitelja povezani su preko prilagodnika ispitne okoline i prilagodnika sustava (slika 4.5). Prilagodnik ispitne okoline isporučuje se zajedno sa alatom CTG te predstavlja shemu ispitnih pogonskih programa. Inženjer ispitivač ne treba mijenjati prilagodnik ispitne okoline jer je isti za sve ispitne modele. Zadatak inženjera ispitivača je definiranje preslikavanja apstraktnih ispitnih slučajeva iz ispitnog modela u konkretne ispitne ulaze programskog ostvarenja sustava kako bi alat CTG mogao stvoriti ispitne skripte. Taj zadatak inženjer ispitivač ostvaruje u obliku prilagodnika sustava. Prilagodnik sustava ostvaren je programskim jezikom C te zajedno sa prilagodnikom ispitne okoline čini cjelinu koja povezuje ispitnu okolinu i ispitivani sustav. Metode prilagodnika sustava pozivaju se iz prilagodnika ispitne okoline.



Slika 4.5 Komunikacija ispitne okoline i ispitivanog ostvarenja poslužitelja

Prilagodnik sustava čeka na vezu na određenoj priključnici (engl. *port*), pretpostavljena vrijednost je 9876. Parametar *SUT TCP/IP port number* u alatu CTG treba postaviti na istu vrijednost. Pored toga, u alatu CTG mora biti odgovarajuće postavljen parametar *SUT host name* na adresu računala na kojem se nalazi prilagodnik sustava (ili *localhost* ako je prilagodnik sustava postavljen na istom računalu kao ispitna okolina, tj. alat CTG).

Preko prilagodnika ispitne okoline, prilagodnik sustava prima od ispitne okoline sve tipove podataka definirane unutar CTG projekta koji predstavljaju apstraktne ispitne ulaze. Osim modela, CTG projekt sastoji se i od korisnički tipova podataka, varijabli i procedura, definiranih *CTG akcijskim jezikom*. Da bi prilagodnik sustava znao s kojim tipovima podataka može upravljati, korisnik treba ostvariti sljedeću proceduru *pass_types_to_datahandler* :

```
static DataType type_Send = NULL;
static DataType type_Response = NULL;
static DataType type_Request = NULL;

/* types je pokazivač na polje tipova ispitnih podataka koje procedura čita i sprema u
 *  tipove razumljive prilagodniku sustava (type_Send, type_Response, type_Request)
*/
int pass_types_to_datahandler(DataType *types, size_t nTypes)
{
   size_t i;

   for (i = 0; i < nTypes; i++)
   {
      if (0 == strcmp(dt_name(dmgr, types[i]), "Send"))
      {
         type_Send = types[i];
      }
      else if (0 == strcmp(dt_name(dmgr, types[i]), "Request"))
      {
         type_Request = types[i];
      }
     else if (0 == strcmp(dt_name(dmgr, types[i]), "Response"))
     {
         type_Response = types[i];
     }
   }
   return 1;
}
```

Prilagodnik sustava preko argumenata procedure *pass_types_to_datahandler* prima od prilagodnika ispitne okoline polje tipova podataka koji opisuju strukturu ulaznih i izlaznih ispitnih podataka. Na osnovu procedure *pass_types_to_datahandler*, prilagodnik sustava zna kakve su strukture poruke koje prima i koje treba slati ispitnoj okolini preko prilagodnika ispitne okoline.

Osim procedure *pass_types_to_datahandler*, potrebno je ostvariti i proceduru *reply_new_datum*. Procedura *reply_new_datum* na osnovu *ispitnih podataka* dobivenih od prilagodnika ispitne okoline i na osnovu procedure *pass_types_to_datahandler* stvara konkretne ispitne ulaze za ispitivani sustav. Programsko ostvarenje poslužitelja potom vraća ispitne izlaze koje prilagodnik sustava prosljeđuje prilagodniku ispitne okoline. Prilagodnik ispitne okoline ispitne izlaze pretvara u *ispitne podatke* razumljive ispitnoj okolini te ih šalje

na analizu ispitnoj okolini. Ispitna okolina uspoređuje dobivene sa očekivanim odgovorima te donosi odluku o ispravnosti rada ispitivanog ostvarenja poslužitelja.

Ispitivanje je pokazalo da programsko ostvarenje poslužitelja ispravno radi u skladu sa definiranim zahtjevima.

5. Ispitivanje sigurnosnog podsustava okoline PIE ispitnim modelom

Prethodna poglavlja sadrže teorijsku osnovu tehnike *ispitivanja zasnovanog na modelu* te primjer ispitivanja jednostavne korisnik-poslužitelj okoline. Ovo poglavlje fokusira se na programsko ostvarenje ispitivanja sustava zasnovanog na uslugama ispitnim modelom. Sustav koji se ispituje predstavlja sigurnosnu infrastrukturu *Prividne logičke mreže* (engl. *overlay network*) koja čini osnovicu arhitekture *Programirljive Internet okoline* (*PIE*). U nastavku poglavlja opisuje se arhitektura *Programirljive Internet okoline* koja omogućuje izgradnju i izvođenje raspodijeljenih primjenskih sustava zasnovanih na uslugama. Potom se opisuje *Prividna logička mreža* te arhitektura ispitivanog sigurnosnog podsustava okoline *PIE*. Središnji dio poglavlja sadrži opis ostvarenja ispitivanja sigurnosnog podsustava okoline *PIE* ispitnim modelom. Na kraju se razmatra analiza rezultata provedenog ispitivanja.

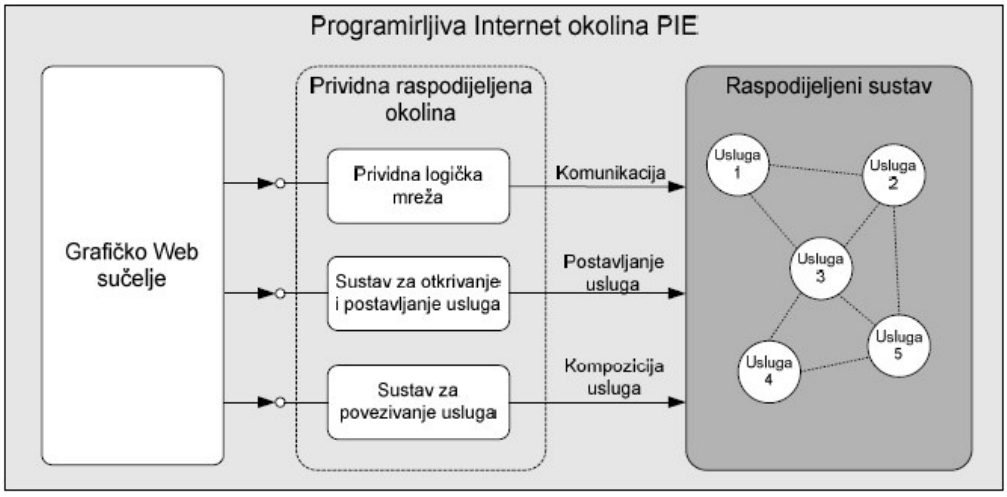
5.1. Programirljiva Internet okolina

Programirljiva Internet okolina korisnicima omogućuje izgradnju i izvođenje raspodijeljenih primjenskih sustava zasnovanih na programskom modelu zasnovanom na uslugama [2]. Osnovni gradivni elementi pri oblikovanju i izgradnji raspodijeljenih programskih sustava uporabom programskog modela zasnovanog na uslugama su programske usluge (engl. *services*). Okolina *PIE* razlikuje tri vrste programskih usluga: primjenske usluge (engl. *application services*), raspodijeljeni programi (engl. *distributed programs*) te usluge za suradnju i natjecanje (*engl. coopetition services, coopetition = cooperation + competition*).

Primjenske usluge ostvaruju funkcionalnosti svojstvene pojedinim područjima primjene, npr. mogućnost obavljanja bankovnih transakcija.

Raspodijeljeni programi omogućavaju povezivanje pojedinačnih primjenskih usluga u nove i složenije primjenske usluge. Raspodijeljeni programi sadrže programsku logiku potrebnu za usklađivanje rada primjenskih usluga uporabom usluga za suradnju i natjecanje. Programska logika svojstvena području primjene raspodijeljenog programskog sustava izdvojena je iz raspodijeljenih programa i u potpunosti je ostvarena u primjenskim uslugama raspodijeljenim diljem globalne mreže Internet.

Usluge za suradnju i natjecanje služe za povezivanje primjenskih usluga u raspodijeljene programe te omogućavanje njihove međusobne komunikacije i usklađivanje rada (engl. *synchronization*). Prividna raspodijeljena računalna okolina koristi četiri vrste usluga za suradnju i natjecanje: binarni semafor, opći semafor, poštanski pretinac i usmjernik događaja. Navedene usluge za suradnju i natjecanje detaljnije se opisuju u odjeljku 5.1.3.



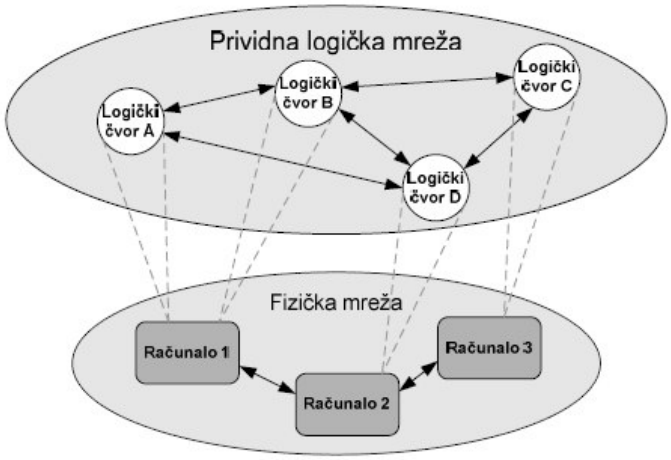
Slika 5.1 Arhitektura Programirljive Internet okoline

Arhitektura okoline *PIE* prikazana je slikom 5.1. Okolina *PIE* sastoji se od dva osnovna dijela: *Grafičkog Web sučelja* i *Prividne raspodijeljene okoline*. *Prividna raspodijeljena okolina* sastoji se od tri podsustava: *Prividne logičke mreže*, *Sustava za otkrivanje i postavljanje usluga* i *Sustava za povezivanje usluga.* Slojevitost arhitekture *Prividne raspodijeljene okoline* naglašava zavisnost pojedinih podsustava. Podsustav na višoj razini koristi funkcionalnosti podsustava na nižim razinama. *Prividna logička mreža* [3] omogućuje logičko adresiranje usluga. *Sustav za otkrivanje i postavljanje usluga* [4] ostvaruje funkcionalnost registra usluga te dodatno omogućuje postavljanje i uklanjanje usluga sa čvorova *Prividne logičke mreže*. *Sustav za povezivanje usluga* [4] ostvaruje osnovna načela programskog modela zasnovanog na uslugama, kao što su sinkronizacijske i komunikacijske usluge, te prevođenje i izvođenje raspodijeljenih programa.

Grafičko Web sučelje [2] omogućuje pisanje raspodijeljenih programa primjenom jezika *SSCL* (engl. *Simple Service Composition Language,* skraćeno SSCL) [5] te pokretanje postupka njihovog prevođenja i izvođenja putem sustava interaktivnih izbornika.

5.1.1. Prividna logička mreža

Osnova arhitekture okoline *PIE* je *Prividna logička mreža* koja se nadovezuje na fizičku računalnu mrežu. Cilj *Prividne logičke mreže* je ostvariti nezavisnost raspodijeljenog primjenskog sustava o raznorodnoj fizičkoj infrastrukturi, podržati jednostavno adresiranje te omogućiti komunikaciju usluga u takvom okruženju. Slika 5.2 prikazuje primjer računalne mreže sa tri računala koja se primjenom *Prividne logičke mreže* preslikava u logičku mrežu sa četiri logička čvora.



Slika 5.2 Preslikavanje fizičke mreže računala u mrežu logičkih čvorova

Logički čvorovi nesmetano ulaze i izlaze iz *Prividne logičke mreže*. Jedna od prednosti korištenja *Prividne logičke mreže* je u prenosivosti raspodijeljenog primjenskog sustava s jednog skupa računala na drugi. Umjesto promjena na raspodijeljenom primjenskom sustavu, prilikom prijenosa raspodijeljenog primjenskog sustava potrebno je samo postaviti odgovarajuću topologiju *Prividne logičke mreže.* Nezavisnost komunikacije među uslugama o fizičkoj računalnoj mreži ostvaruje se uvođenjem logičkih čvorova. Osim toga, uvođenje logičkih čvorova omogućuje logičko adresiranje usluga smještenih na logičkim čvorovima. Tako je svaka usluga raspodijeljenog sustava jedinstveno određena imenom logičkog čvora na kojem se nalazi, te imenom usluge na tom čvoru. Logičko adresiranje usluga i čvorova ostvaruje se primjenom *WS-Addressing* standarda. Koristeći opisani način adresiranja, *Prividna logička mreža* obavlja zadaću usmjeravanja zahtjeva između usluga, a osim toga zadužena je i za sigurnost i tajnost komunikacije koja prolazi sustavom.

5.1.2. Sustav za otkrivanje i postavljanje usluga

Sustav za otkrivanje i postavljanje usluga nadovezuje se na *Prividnu logičku mrežu*. Otkrivanje sredstava nužna je funkcionalnost za izgradnju i izvođenje raspodijeljenih primjenskih sustava u dinamičkim okolinama. Zbog toga *Sustav za otkrivanje i postavljanje usluga* pruža funkcionalnost registra usluga u koji se spremaju lokacije aktivnih usluga u *Prividnoj logičkoj mreži*. *Sustav za otkrivanje i postavljanje usluga* automatizira postupak postavljanja usluga na logičke čvorove *Prividne logičke mreže*. Automatizacija postupka postavljanja usluga smanjuje složenost postupka postavljanja primjenskih sustava te pruža potporu upravljanju opterećenjem raspodijeljenog primjenskog sustava. Nadalje, *Sustav za otkrivanje i postavljanje usluga* zadužen je za vođenje popisa stvorenih i uništenih primjeraka. Dakle, *Sustav za otkrivanje i postavljanje usluga* sadrži listu svih spremljenih i postavljenih usluga i svih primjeraka usluga.

5.1.3. Sustav za povezivanje usluga

Sustav za povezivanje usluga pruža potporu oblikovanju i izvođenju složenih raspodijeljenih primjenskih sustava primjenom programskog modela zasnovanog na uslugama. *Sustav za povezivanje usluga* ostvaruje sinkronizacijske i komunikacijske usluge te usluge prevođenja i izvođenja raspodijeljenih programa.

Prevođenje i izvođenje raspodijeljenih programa podržano je skupom mrežnih usluga koje ostvaruju *SSCL prevoditelje* te *CL interpretatore*. Raspodijeljeni programi pisani jezikom SSCL prevode se u programski kod jezika CL (engl. *Coopetition Language*) koji se dalje prevodi u C# programski kod. Izvođenje *C#* raspodijeljenog programa ostvareno je primjenom *.NET Framework* sustava. Kako bi raspodijeljeni program počeo sa izvođenjem, potrebno je putem sučelja mrežne usluge raspodijeljenog programa stvoriti primjerak. Svaki raspodijeljeni program može se izvoditi u više primjeraka, s time da je izvođenje primjeraka istog raspodijeljenog programa međusobno neovisno.

Dostupne usluge sinkronizacije i komunikacije su binarni i opći semafor, poštanski pretinac te usmjernik događaja. Usluge sinkronizacije i komunikacije ostvarene su kao mrežne usluge s pohranom stanja pa je prije korištenja potrebno stvoriti primjerak usluge.

Binarni semafor (engl. *binary semaphore*) koristi se za usklađivanje rada raspodijeljenih programa međusobnim isključivanjem. Binarni semafor u svakom trenutku zauzima najviše jedan raspodijeljeni program.

Opći semafor (engl. *counting semaphore*) zasniva se na proširenju binarnog semafora. U odnosu na binarni semafor, više raspodijeljenih programa može istovremeno zauzeti opći semafor.

Poštanski pretinac (engl. *mailbox*) se, osim za usklađivanje rada raspodijeljenih programa, koristi i za njihovu međusobnu komunikaciju putem razmjene poruka. Komunikacija primjenom poštanskog pretinca vremenski je nepovezana budući da poštanski pretinac sprema primljene poruke sve dok ih neka usluga ne pročita. Razmjenom poruka putem mehanizma poštanskog pretinca ostvaruje se komunikacija među raspodijeljenim programima, dok je blokirajućom operacijom čitanja poštanskog pretinca osigurano usklađivanje njihova rada.

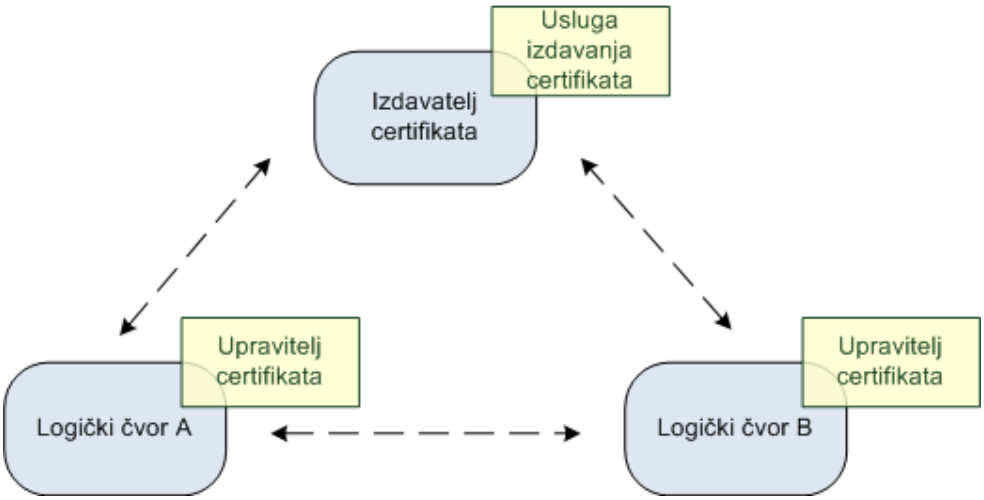
Usmjernik događaja (engl. *event channel*) ostvaruje objava/pretplata (engl. *publish/subscribe*) komunikacijski mehanizam koji pruža napredne mogućnosti tumačenja događaja. Raspodijeljeni programi prijavljivanjem pri usmjerniku događaja iskazuju zanimanje za određenu vrstu događaja koji se mogu pojaviti u raspodijeljenoj okolini. S druge strane, raspodijeljeni programi koji otkrivaju pojavu određenih vrsta događaja dojavljuju pojave događaja usmjerniku događaja. Uparivanjem objavljenih događaja s iskazanim zanimanjima za događaje, usmjernik događaja dojavljuje pojavu događaja raspodijeljenom programu koji je iskazao zanimanje za dotični događaj.

5.2. Sigurnosni podsustav okoline PIE

Osnova arhitekture okoline *PIE* je *Prividna logička mreža*. *Prividna logička mreža* je raspodijeljeni programski sustav čiji se elementi nalaze na svim čvorovima fizičke mreže kojima su pridruženi čvorovi *Prividne logičke mreže*. Programski sustav *Prividne logičke mreže* čini skup programskih usluga. Programska usluga *Prividne logičke mreže* čini jedinstvenu i samostalnu programsku cjelinu koja obavlja funkciju sustava *Prividne logičke mreže* na jednom čvoru fizičke mreže. Jedna od komponenata programske usluge *Prividne logičke mreže* je i sustav infrastrukture javnih ključeva (engl. *Public Key Infrastructure*, skraćeno PKI). Predmet diplomskog rada je ispitivanje upravo tog sustava ispitnim modelom.

Sigurnosni podsustav okoline *PIE* sastoji se od *Izdavatelja certifikata* (engl. *Certificate Authority*, skraćeno CA) i logičkih čvorova (korisnika). *Izdavatelj certifikata* je entitet koji izdaje digitalne certifikate za korištenje od strane korisnika. To je primjer pouzdane treće strane u sigurnoj komunikaciji. *Izdavatelji certifikata* su karakteristični za mnoge sheme infrastrukture javnih ključeva. Svaki logički čvor koji postane dijelom okoline *PIE* dobiva

svoj certifikat koji sadrži informacije o asimetričnom paru ključeva, trajanju certifikata, digitalnom potpisu, izdavatelju certifikata i slično. Svaki logički čvor dobiva i certifikat izdavatelja certifikata kako bi mogao provjeriti digitalni potpis digitalnih certifikata ostalih logičkih čvorova. Prije uspostave komunikacije sa drugim čvorom, čvor mora primiti certifikat drugog čvora kako bi dobio informaciju o njegovom javnom ključu te informaciju o sigurnosti tog čvora (provjeravajući digitalni potpis primljenog certifikata). Slika 5.3 prikazuje proces uspostave infrastrukture PKI u okolini *PIE*.



Slika 5.3 Entiteti sigurnosnog podsustava okoline *PIE*

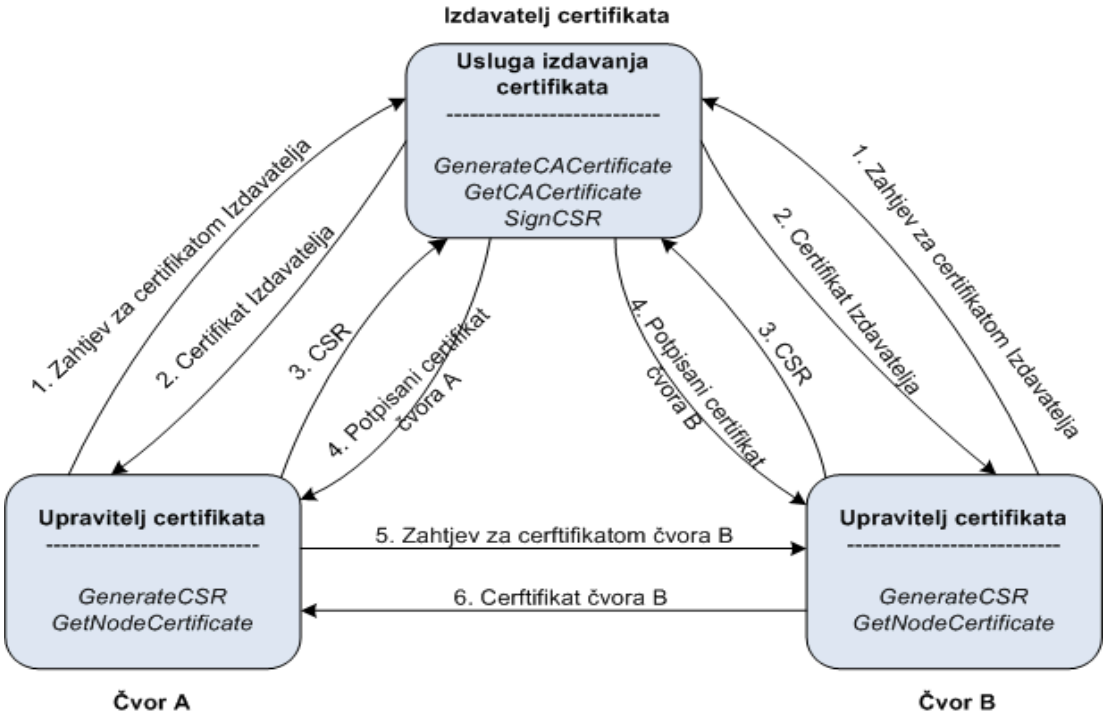
U sigurnosnom podsustavu okoline *PIE* uključena su dva protokola (slika 5.4). Prvi protokol je protokol dohvata certifikata (engl. *Certificate Retrieval Protocol*). Protokol dohvata certifikata definira razmjenu poruka između *Izdavatelja certifikata* i logičkih čvorova:

1. Logički čvor traži digitalni certifikat od *Izdavatelja certifikata*
2. *Izdavatelj certifikata* vraća svoj digitalni certifikat
3. Logički čvor stvara svoj zahtjev za potpisivanjem certifikata (engl. *Certificate Signing Request*, skraćeno CSR) i šalje *Izdavatelju certifikata*
4. *Izdavatelj certifikata* potpisuje zahtjev za potpisivanjem certifikata i vraća ga logičkom čvoru

Drugi protokol je protokol izmjene (engl. *Exchange Protocol*), koji definira početak komunikacije između logičkih čvorova:

5. Logički čvor A želi digitalni certifikat od logičkog čvora B
6. Logički čvor B vraća svoj digitalni certifikat logičkom čvoru A

Nakon što je infrastruktura PKI postavljena, može se provesti izmjena tajnog ključa i početi sigurna komunikacija između čvorova.



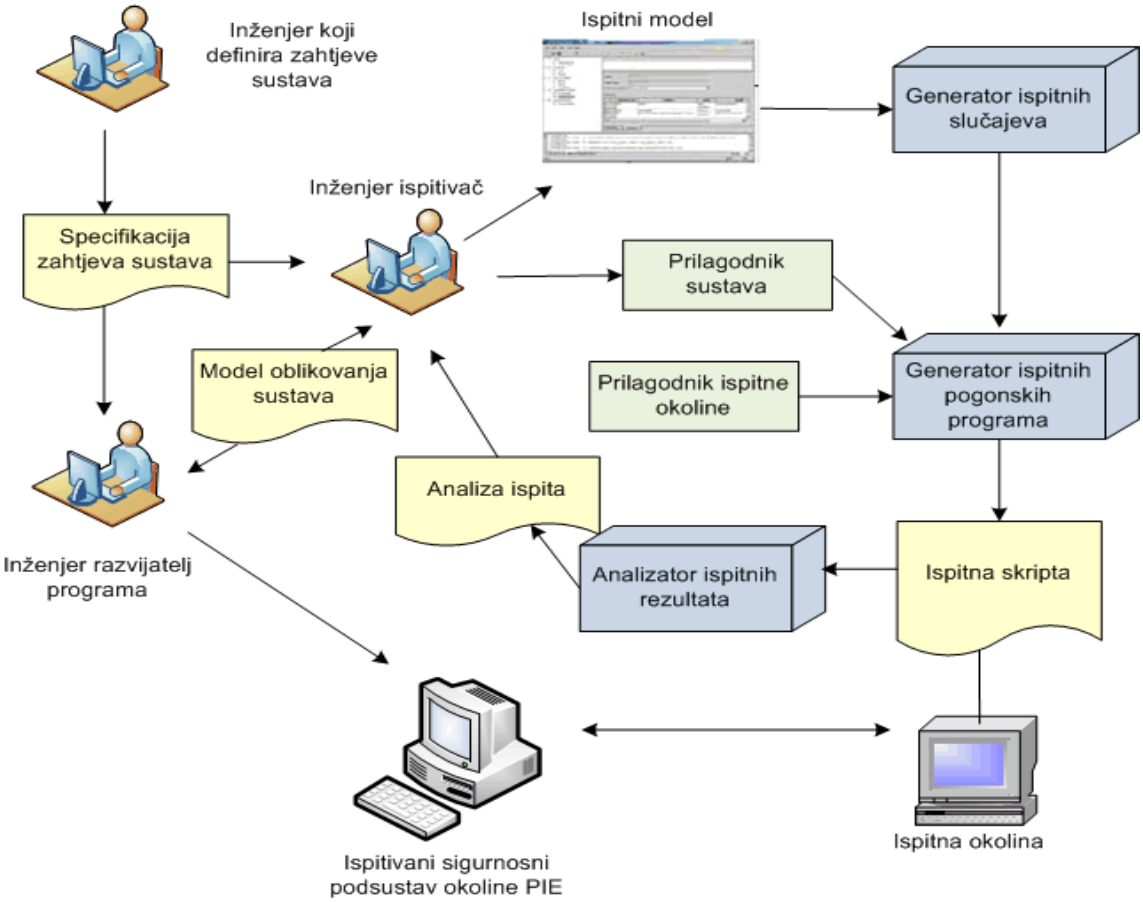
Slika 5.4 Model oblikovanja sigurnosnog podsustava okoline *PIE*

Funkcionalnost sigurnosnog podsustava okoline *PIE* ostvarena je putem dvije Web usluge, *Usluge izdavanja certifikata* (engl. *Certificate Service*) i *Upravitelj certifikata* (engl. *Certificate Manager*). Logički čvorovi koriste uslugu *Upravitelj certifikata*, a *Izdavatelj certifikata* koristi *Uslugu certifikat* (kao što je prikazano na slici 5.3). Obje usluge ostvarene su koristeći radni okvir .NET 1.1. *Upravitelj certifikata* ostvaren je kao Web usluga koja se poziva iz *Prividne logičke mreže* okoline *PIE*, a *Usluga izdavanja certifikata* izložena je kao samostalna Web usluga na nekom poslužitelju neovisnom o okolini *PIE*. Radi se o uslugama bez očuvanja stanja koje se pozivaju samo onda kada je to potrebno.

Potrebno je napomenuti razliku između ovog rješenja infrastrukture PKI i onih koji se uobičajeno koriste. Ovaj sustav izlaže funkcionalnost infrastrukture PKI putem Web usluga, čineći ga jednostavnim za korištenje u raspodijeljenoj okolini usmjerenoj Web uslugama. Stvaranje, raspodjela i postavljanje digitalnih certifikata održava se automatski bez potrebe za korisnikovim učestvovanjem.

5.3. Programsko ostvarenje ispitivanja sigurnosnog podsustava okoline PIE ispitnim modelom

Ispitivano programsko ostvarenje sastoji se od dvije Web usluge opisane u prethodnom odjeljku, *Usluga izdavanja certifikata* i *Upravitelj certifikata*.. Proces ispitivanja sigurnosnog podsustava okoline *PIE* započinje nakon definiranja zahtjeva tog sustava te se gradi paralelno sa procesom razvijanja sustava.



Slika 5.5 Konceptualni prikaz ispitivanja sigurnosnog podsustava okoline PIE ispitnim modelom

Kao što je prikazano na slici 5.5, proces *ispitivanja zasnovanog na modelu* ispitivanog sustava sastoji se od tri koraka:

1. Na osnovu definiranih zahtjeva, u suradnji sa timom za razvoj sigurnosnog podsustava okoline PIE gradi se model oblikovanja sustava. Na temelju modela oblikovanja, korištenjem alata CTG izgrađen je **ispitni model**. Ispitni model izgrađen je tako da se na osnovu njega mogu stvoriti ispitni slučajevi koji će pokriti sva pitanja na koja ispitivani sustav treba odgovoriti.

2. Nakon ispitnog modela, izgrađen je **prilagodnik sustava**. Prilagodnik sustava na odgovarajući način prilagođava stvorene ispitne slučajeve u oblik razumljiv ispitivanom programskom ostvarenju. Prilagodnik sustava ostvaren je upotrebom programskog jezika Java tako da radi sa prilagodnikom ispitne okoline unutar istog računalnog procesa. Prilagodnik ispitne okoline dolazi kao dio alata CTG te ga nije potrebno mijenjati.
3. Nakon izgrađenog ispitnog modela i prilagodnika sustava, alat CTG stvara ispitne skripte koje upravljaju ispitivanjem sa pristupom stvaranju ispitnih slučajeva tijekom izvođenja. Tijekom ispitivanja alat CTG ima sposobnost **analize ispitnih izlaza**, a na kraju ispitivanja stvara izvještaj provedenog ispitivanja.

5.3.1. Ispitni model

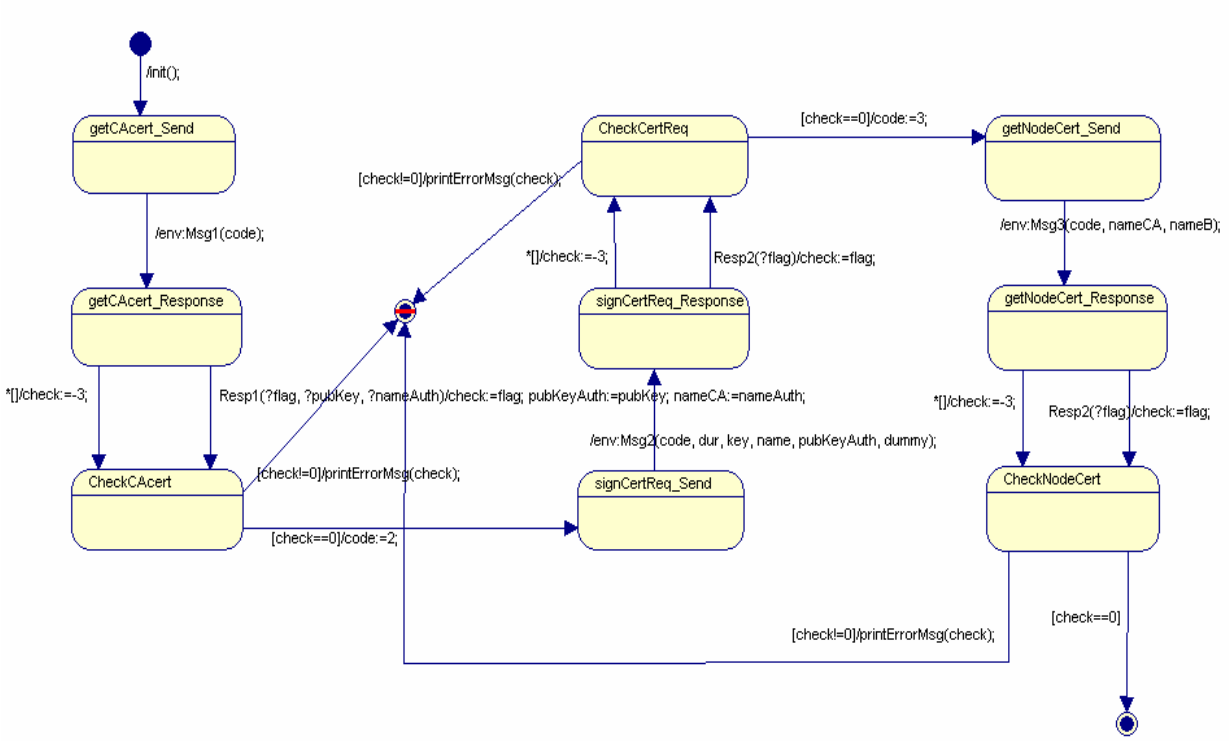
Za gradnju ispitnog modela potrebno je prethodno utvrditi zahtjeve sustava i izraditi model oblikovanja. Izrada zahtjeva sustava zadatak je tima za razvoj sustava, dok se model oblikovanja radio zajedno sa timom za razvoj sustava.

Ispravno izgrađeni ispitni model omogućuje stvaranje ispitnih slučajeva čiji rezultati daju odgovore na sljedeća pitanja:

1. kada čvor Prividne logičke mreže pošalje Izdavatelju certifikata zahtjev za certifikatom Izdavatelj certifikata:
   - vraća li *Izdavatelj certifikata* X509 certifikat?
   - vraća li *Izdavatelj certifikata* vremenski ispravan certifikat?
2. kada čvor *Prividne logičke mreže* pošalje *Izdavatelju certifikata* zahtjev za potpisivanjem certifikata:
   - vraća li *Izdavatelj certifikata* potpisani certifikat upravo tog čvora koji je potpisivanje zatražio?
   - je li duljina ključa u vraćenom potpisanom certifikatu točno ona koja je zatražena zahtjevom?
   - je li javni ključ čvora u vraćenom potpisanom certifikatu točno onaj koji je zatražen zahtjevom?
   - je li vrijeme trajanja potpisanog certifikata točno onakvo kakvo je zatraženo zahtjevom?
   - je li vraćeni potpisani certifikat stvarno potpisan od *Izdavatelja certifikata*?
3. kada čvor *Prividne logičke mreže* pošalje *Izdavatelju certifikata* lažni/neispravni zahtjev za potpisivanjem certifikata, da li *Izdavatelj certifikata* to prepoznaje kao pokušaj prevare ili svejedno vraća nekakav certifikat?

4. kada čvor *Prividne logičke mreže* pošalje drugom čvoru *Prividne logičke mreže* zahtjev za njegovim certifikatom:
   - vraća li drugi čvor vremenski ispravan X509 certifikat?
   - vraća li drugi čvor certifikat potpisan od *Izdavatelja certifikata*?

Ispitni model, na osnovu kojeg se stvaraju ispitni slučajevi koji daju odgovor na tražena pitanja, prikazan je na slici 5.6.



Slika 5.6 Ispitni model

Početak modela označen je početnim stanjem (potpuno obojani krug). Prilikom prijelaza u stanje *getCAcert_Send*, inicijaliziraju se određene varijable procedurom *init*. Procedure, kao i nizovi znakova što određuju prijelaze, pišu se u skladu s pravilima *CTG akcijskog jezika*. Procedura *init* određuje prostor mogućih vrijednosti varijabli koje se koriste za svrhe ispitivanja. Stanje *getCAcert_Send* označava da su se ispitne varijable inicijalizirale na neku od mogućih vrijednosti te može početi komunikacija sa ispitivanim sustavom.

Prvi korak je slanje zahtjeva *Izdavatelju certifikata* za njegovim certifikatom. To je opisano prijelazom */env:Msg1(code)* koji označava da se šalje poruka scenariju okoline (tj. ispitivanom sustavu) sa parametrom *code* koji je postavljen na vrijednost *1*. Na osnovu vrijednosti tog parametra i tipa *ispitnog podatka*, prilagodnik sustava zna što se traži od

ispitivanog sustava. Konkretno, za tip podatka *Msg1* i vrijednost *code=1*, prilagodnik sustava šalje zahtjev *Izdavatelju certifikata* za njegovim certifikatom. Nakon toga, dolazi se u stanje *getCAcert_Response*, koji označava čekanje da ispitivani sustav vrati certifikat *Izdavatelja certifikata*. Kada se to dogodi, prilikom prijelaza u iduće stanje, vrši se ispitivanje dobivene poruke. Ako je pristigla poruka očekivanog formata, sprema se vrijednost varijable *check* na vrijednost koju je odredio prilagodnik sustava (na osnovu odgovora dobivenog od ispitivanog sustava). Ako je vrijednost varijable *check* jednaka nuli, tada je uspješno dobavljen certifikat *Izdavatelja certifikata* pa se interno sprema naziv i javni ključ *Izdavatelja certifikata*. U slučaju da nije došla poruka očekivanog oblika, došlo je do greške u prilagodniku sustava. U sljedećem stanju, *CheckCAcert*, provjerava se varijabla *check*. Ako je varijabla *check* različita od nula, došlo je do pogreške i zaustavlja se trenutni ispitni slučaj sa odlukom *FAIL* i objašnjenjem te se prelazi na ispitivanje novog ispitnog slučaja.

Ako je odgovor ispitivanog sustava u skladu sa očekivanim, tada se nastavlja ispitivanje trenutnog ispitnog slučaja i prelazi u stanje *signCertReq_Send*. Iz tog stanja šalje se *Izdavatelju certifikata* zahtjev za potpisivanjem certifikata (koji naravno može biti potpuno neispravan) predstavljen *ispitnom podatkom* tipa *Msg2*. Potom se prelazi u stanje čekanja odgovora od ispitivanog sustava *signCertReq_Responose*. Kada odgovor pristigne, prilagodnik sustava provjerava sve što treba provjeriti (vremenska ispravnost, ispravni potpis, ispravni naziv, ispravna duljina ključa, ispravna duljina trajanja certifikata) te vraća odgovor glavnom scenariju (tj. ispitnom modelu) sa odgovarajućom vrijednosti varijable *check*. Prelaskom u stanje *CheckCertReq*, ispituje se vrijednost varijable *check*. Ako je vrijednost varijable *check* različita od nule, znači da je došlo do greške. U tom slučaju trenutni ispitni slučaj se zaustavlja odlukom *FAIL* i objašnjenjem greške te se prelazi na ispitivanje novog ispitnog slučaja. Ako je odgovor ispitivanog sustava jednak očekivanom, nastavlja se trenutni ispitni slučaj prelaskom u sljedeće stanje i postavljanjem zastavice *code* na vrijednost *3*. Time se prilagodniku sustava označava da se želi uspostaviti komunikacija sa drugim čvorom *Prividne logičke mreže*.

Stanje *getNodeCert_Send* označava da je čvor tijekom svoje inicijalizacije uspješno dobio svoj certifikat potpisan od *Izdavatelja certifikata* (također posjeduje i certifikat *Izdavatelja certifikata*). Iz tog stanja pokušava se uspostaviti komunikacija sa drugim čvorom *Prividne logičke mreže*. Ispitivanom sustavu šalje se *ispitni podatak* tipa *Msg3* koji označuje zahtjev za certifikatom drugog čvora sa kojim se želi komunicirati. Prelaskom u stanje *getNodeCert_Response* čeka se na odgovor od ispitivanog sustava. Kada odgovor pristigne,

prilagodnik sustava provjerava dobiveni certifikat te vraća odgovarajući odgovor ispitnoj okolini koja zatim postavlja varijablu *check* i prelazi u stanje *CheckNodeCert*. Ako je vrijednost varijable *check* različita od nule, došlo je do greške i prelazi se u završno stanje sa odlukom *FAIL* i objašnjenjem greške te se pokreće novi ispitni slučaj. U suprotnom, ispitni slučaj je prošao odlukom *PASS* i pokreće se novi ispitni slučaj.

Ispitivanje završava kada se dostigne uvjet zaustavljanja, npr. *Stop at 100% coverage* (zaustavi se kada se dostigne 100% pokrivenost prostora mogućih vrijednosti ispitnih podataka).

5.3.2. Prilagodnik sustava

Za razliku od *ispitivanja zasnovanog na modelu* nad jednostavnom korisnik-poslužitelj okolinom (opisano u poglavlju 4), prilagodnik sustava za sigurnosni podsustav okoline *PIE* izrađen je uporabom programskog jezika Java (koristeći razvojnu okolinu *NetBeans*).

Prilagodnik sustava i prilagodnik ispitne okoline zapravo su dva dijela jedne cjeline. API za prilagodnik ispitne okoline dolazi kao dio korištenog alata CTG. Na korisniku, tj. inženjeru ispitivaču, preostaje zadatak da ostvari prilagodnik sustava zaseban za svaki pojedini ispitivani sustav. U tu svrhu izrađeni su zastupnici (engl. *proxy*) Web usluga *Usluga izdavanja certifikata* i *Upravitelj certifikata* te razredi *SystemAdapterPIE_PKI* i *SysAdapterFrame*.

Zastupnik Web usluge *Usluga izdavanja certifikata*

Ispitivano programsko ostvarenje sastoji se od dviju Web usluga te se ispituje pozivom njihovih javnih sučelja. Zbog toga je izgrađen zastupnik Web usluge *Usluga izdavanja certifikata* kao dio prilagodnika sustava kako bi bila moguća komunikacija sa ispitivanim programskim ostvarenjem. Korištenjem razvojne okoline *NetBeans*, izrada zastupnika Web usluge je automatizirana. Takav automatizirani postupak uzrokuje stvaranje nekoliko sučelja i razreda, od čega su posebno važni: *CertificateService*, *CertificateService_Impl* i *CertificateServiceSoap.* Navedeni razredi izravno se koriste u razredu *SystemAdapterPIE_PKI* za ostvarivanje komunikacije sa dijelom ispitivanog programskog ostvarenja zaduženog za izdavanje certifikata.

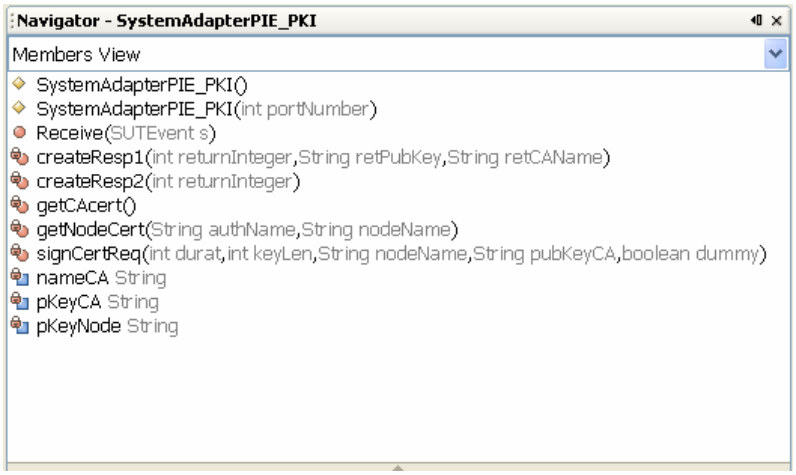
Zastupnik Web usluge *Upravitelj certifikata*

Zastupnik Web usluge *Upravitelj certifikata* izrađen je kao dio prilagodnika sustava kako bi bila moguća komunikacija sa ispitivanim programskim ostvarenjem. Automatizirani postupak

izrade zastupnika uzrokuje stvaranje nekoliko sučelja i razreda, od čega su posebno važni: *CertificateManager*, *CertificateManager_Impl* i *CertificateManagerSoap.* Navedeni razredi izravno se koriste u razredu *SystemAdapterPIE_PKI* za ostvarivanje komunikacije sa dijelom ispitivanog programskog ostvarenja zaduženog za uspostavu sigurnosne komunikacije među čvorovima *Prividne logičke mreže*.

Razred SystemAdapterPIE_PKI

Razred *SystemAdapterPIE_PKI* središnji je razred prilagodnika sustava. Ovaj razred ostvaruje funkcionalnost pretvorbe *ispitnih podataka* poslanih od ispitne okoline u konkretne ispitne ulaze sigurnosnog podsustava okoline *PIE*. Također, prilagodnik sustava čeka na ispitne izlaze dobivene od ispitivanog ostvarenja sustava te ih prilagođuje u izlazne ispitne podatke razumljive prilagodniku ispitne okoline. Razred *SystemAdapterPIE_PKI* nasljeđuje postojeći apstraktni razred *AbsSystemAdapter*. Članovi razreda *SystemAdapterPIE_PKI* prikazani su na slici 5.7.



Slika 5.7 Lista članova razreda SystemAdapterPIE_PKI

Konstruktor razreda *SystemAdapterPIE_PKI definira priključnicu preko koje* prilagodnik sustava komunicira sa ispitnom okolinom. Vrijednost priključnice definira korisnik preko grafičkog korisničkog sučelja ili se koristi pretpostavljena vrijednost (9876). Pritom treba paziti da se parametar *SUT port number* u alatu CTG postavi na istu vrijednost kako bi se uspješno uspostavila veza između ispitne okoline i prilagodnika sustava. Uobičajeno je prilagodnik sustava na istom računalu kao i ispitivano programsko ostvarenje, no može biti i na bilo kojem drugom računalu (u tom slučaju parametri zastupnika Web usluga *Upravitelja certifikata* i *Usluge izdavanja certifikata* moraju biti odgovarajuće promijenjeni).

Kada se uspostavi veza sa ispitnom okolinom, prilagodnik ispitne okoline poziva metodu *Receive* prilagodnika sustava i prenosi primljene ulazne *ispitne podatke* prilagodniku sustava. Na osnovu ulaznih *ispitnih podataka*, prilagodnik sustava odlučuje koju od metoda *getCAcert*, *signCertReq* i *getNodeCert* pozvati. Na osnovu tipa ulaznih *ispitnih podataka*, svaka od navedenih metoda prilagođava ispitne podatke u konkretne ispitne ulaze ispitivanog programskog ostvarenja sustava te čeka na ispitne izlaze.

Ako je primljen *ispitni podatak* tipa *Msg1*, metoda *GetCAcert* preko zastupnika poziva operaciju *getCACertificate* Web usluge *Usluga izdavanja certifikata*:

```
CertificateService cserv = new CertificateService_Impl();
CertificateServiceSoap cservsoap = cserv.getCertificateServiceSoap();
scert = cservsoap.getCACertificate();
```

Nakon što ispitivano programsko ostvarenje vrati odgovor, metoda *GetCAcert* provjerava je li vraćen potpuno ispravan certifikat X509. Po završetku metode *GetCAcert*, metoda *Receive* poziva metodu *createResp1* koja oblikuje odgovarajući izlazni *ispitni podatak* koji se šalje prilagodniku ispitne okoline.

Ako ispitna okolina šalje *ispitni podatak* tipa *Msg2*, poziva se metoda *SignCertReq.* Metoda *SignCertReq* stvara zahtjev za potpisivanjem certifikata te ga potom šalje kao parametar operacije *signCSR* pozvane Web usluge *Usluga izdavanja certifikata*. Zatim se provjerava što je ispitivano programsko ostvarenje vratilo te se pozivom metode *createResp1* oblikuje odgovarajući izlazni *ispitni podatak* koji se šalje kao odgovor ispitnoj okolini:

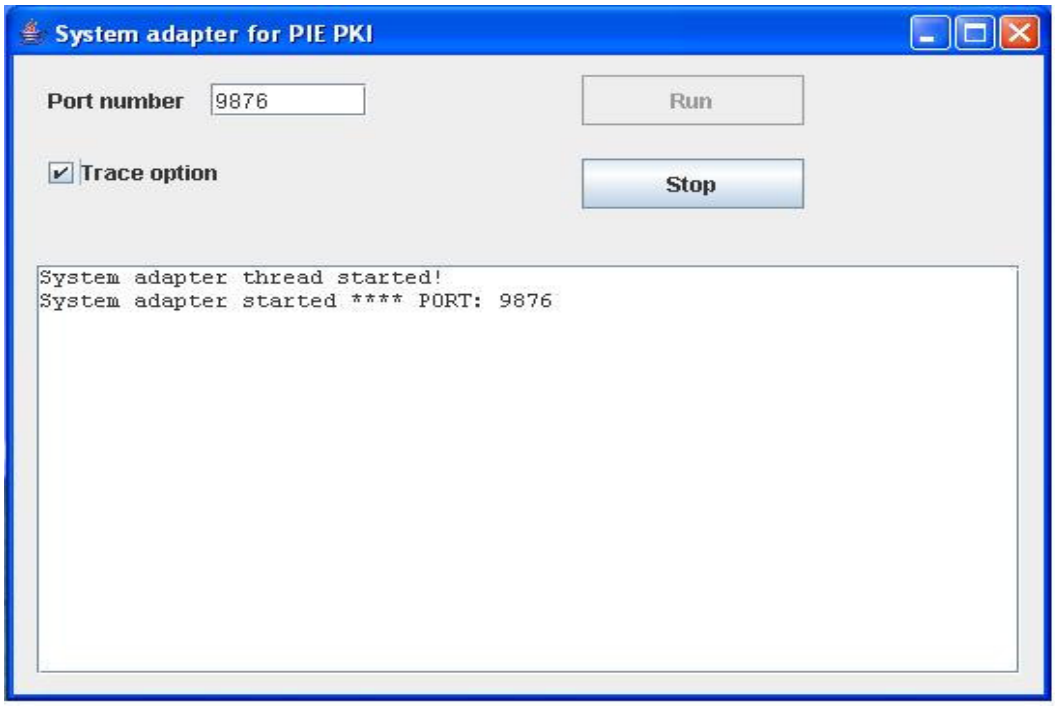
```
DatumType typeMessage = this.mTypes.getType("Resp2");
DatumType typeInteger = this.mTypes.getType("integer");

DatumInteger di = new DatumInteger();
di.setDatumType(typeInteger);
di.setValue(returnInteger);

DatumRecord dr = new DatumRecord(1);
dr.setDatumType(typeMessage);
dr.addField(0, di);
```

U slučaju da ispitna okolina šalje *ispitni podatak* tipa *Msg3*, poziva se metoda *GetNodeCert.* Metoda *GetNodeCert* preko zastupnika poziva operaciju *getNodeCertificate* Web usluge *Usluga izdavanja certifikata*. Nakon što ispitivano programsko ostvarenje vrati rezultat,

metoda *GetNodeCert* provjerava je li vraćen potpuno ispravan X509 certifikat. Po završetku metode *GetNodeCert*, metoda *Receive* poziva metodu *createResp2* koja oblikuje odgovarajući *ispitni podatak* koji se šalje kao odgovor ispitnoj okolini.

Razred SysAdapterFrame

Razred SysAdapterFrame predstavlja grafičko korisničko sučelje preko kojeg se pokreće, zaustavlja ili prati rad prilagodnika sustava (slika 5.8).



Slika 5.8 Grafičko korisničko sučelje prilagodnika sustava

Kada se klikne na gumb *Run*, pokreće se rad prilagodnika sustava u zasebnoj dretvi. Ako je uključena opcija *Trace*, tada prilagodnik sustava tijekom rada ispisuje poruke koje je primio i poslao u za to predviđeno tekstualno polje. Korisnik također upisuje i priključnicu na kojoj prilagodnik sustava čeka na komunikaciju sa ispitnom okolinom. Kada se klikne na gumb *Stop*, prilagodnik sustava čeka da se završi trenutna sjednica ispitivanja te se potom zaustavlja. Ako se klikom na gumb *Stop* prilagodnik sustava zaustavi, za novo otvaranje komunikacije sa ispitnom okolinom mora se kliknuti gumb *Run*.

5.3.3. Rezultati i analiza ispitivanja

Alat CTG pruža nekoliko ispitnih parametara u svrhu što učinkovitijeg ispitivanja. Bez obzira na postavljene vrijednosti parametara analize pokrivenosti i pretraživanja, alat CTG uvijek je uspio prepoznati sljedeće dvije greške na ispitivanom sigurnosnom podsustavu okoline *PIE* :

1. u slučaju kada čvor *Prividne logičke mreže* šalje *Izdavatelju certifikata* zahtjev za potpisivanjem certifikata sa duljinom trajanja certifikata različitom od 365 dana, *Izdavatelj certifikata* vraća pravilno potpisani certifikat, ali sa trajanjem od 365 dana (a ne zahtijevanim brojem dana)
2. u slučaju kada čvor *Prividne logičke mreže* šalje *Izdavatelju certifikata* lažni/neispravni zahtjev za potpisivanjem certifikata, *Izdavatelj certifikata* vraća certifikat jednog od čvorova kojeg ima spremljenog u svom spremištu certifikata, a morao bi javiti grešku.

Tim za razvoj sigurnosnog podsustava okoline *PIE* iskoristio je rezultate ovog ispitivanja kako bi ispravio nedostatke u svom sustavu.

Pravilnim odabirom parametara analize pokrivenosti i pretraživanja postiže se učinkovitije ispitivanje. U nastavku je opisana detaljnija analiza ispitivanja uz različite vrijednosti parametara analize pokrivenosti i pretraživanja. Ispitivanja su provedena na računalu sa signalom takta procesora od 2.6GHz te memorijom RAM od 1GB. Uvjet zaustavljanja je vremensko ograničenje od 15 minuta (900 sekundi) ili postignuta potpuna (100%) pokrivenost prostora mogućih ispita.

Analiza pokrivenosti

Analiza pokrivenosti modela provodi se s ciljem određivanja optimalnih parametara za ispitivanje što većeg broja ispitnih slučajeva u što kraćem vremenu. Postavke za analizu pokrivenosti sastoje se od *kriterija pokrivenosti*, *dubine analize pokrivenosti* i *zalihosti analize pokrivenosti*.

Postoje četiri vrste *kriterija pokrivenosti* podržane od alata CTG: pokrivenost raspodjela, stanja, strukturnih ulaza/izlaza te prijelaza.

Kriterij raspodjele zasniva se na funkciji *partition(X1,...,Xn)* gdje su X1 do Xn logički izrazi, npr. *dur*<365 ili *dummy*==*true*. Pritom varijabla *dur* označava trajanje certifikata u danima, a *dummy* neispravan zahtjev za potpisom certifikata poslan od nekog čvora *Izdavatelju certifikata*. Kriterij raspodjele fokusira se na pokušavanju da pozove proceduru *partition* za

sve moguće logičke kombinacije njenih parametara. Za dani primjer *partition(dur<365, dummy==true)* postoje četiri jasne moguće logičke kombinacije ({*dur*<365, *dummy=true*}, {*dur*<365, *dummy=false*}, {*dur*>=365, *dummy=true*}, {*dur*>=365, *dummy=true*}). Povećanje dubine analize pokrivenosti uzrokuje potrebu za dužim sljedovima poziva funkcije *partition*.

Kriterij stanja fokusira se na kombinacijama svih aktivnih stanja u modelu.

Kriterij strukturnih ulaza/izlaza je kriterij pokrivenosti koji je nezavisan o sintaksi kojom je model opisan. Kriterij strukturnih ulaza/izlaza fokusira se na tipove i veličinu događaja poslanih i primljenih od ispitivanog sustava. Povećanje *dubine analize pokrivenosti* uzrokuje ispitivanje dužih sljedova ulaza/izlaza.

Kriterij prijelaza fokusira se na pokušaju da aktivira sve prijelaze ili duže sljedove prijelaza (ovisno o *dubini analize pokrivenosti*) među stanjima u modelu.

Dubina analize pokrivenosti određuje koliko duge sljedove određenih događaja se pokušava pokriti. Moguće vrijednosti su „zanemariva“, „vrlo mala“, „mala“, „srednja“, „velika“, „vrlo velika“ i „iznimno velika“ dubina. Primjerice, ako se kao *kriterij pokrivenosti* uzme kriterij „prijelaz“ te „zanemariva“ *dubina analize pokrivenosti*, alat CTG pokušava osigurati da se svaki prijelaz u modelu aktivira barem jednom. Ako se poveća dubina, alat će pokušati osigurati da se svaki slijed prijelaza određene duljine (proporcionalan sa dubinom) u modelu aktivira barem jednom.

Ako se promijeni faktor *zalihosti analize pokrivenosti* na neku drugu vrijednost od pretpostavljene, koja iznosi *1*, tada se svaki slijed događaja pokušava pokriti upravo toliko puta. Postavimo li faktor zalihosti na vrijednost *3* i koristimo kriterij „prijelaz“ i „zanemarivu“ dubinu, alat će pokušati aktivirati svaki prijelaz u modelu barem tri puta. Primjerice, ako slijed prijelaza sadrži akcije slanja zahtjeva za certifikatom i primanje certifikata od *Izdavatelja certifikata*, tada će se taj postupak ponoviti tri puta, čineći dobivene rezultate pouzdanijima.

Tablica 5.1 prikazuje ovisnost vremenskog trajanja ispitivanja i broja izvršenih ispita u odnosu na odabrani *kriterij pokrivenosti*. Tijekom provedenog ispitivanja sigurnosnog podsustava okoline *PIE* pokazalo se da je vremensko trajanje ispitivanja i ukupan broj stvorenih ispita neosjetljiv na različite vrijednosti parametra *dubine analize pokrivenosti*. Zbog toga je za ispitivanje utjecaja *kriterija pokrivenosti* odabrana vrijednost „srednja“ parametra *dubine analize pokrivenosti*. Za vrijednost faktora *zalihosti analize pokrivenosti*

odabrana je vrijednost *1* (naravno, veće vrijednosti daju dulja vremena ispitivanja te veći broj stvorenih ispita).

Tablica 5.1 Ovisnost vremenskog trajanja i ukupnog broja ispita o kriteriju pokrivenosti

Kriterij pokrivenosti	Vremensko trajanje (s)	Ukupan broj ispita	Pokrivenost prostora ispita
Prijelaz	270	130	100%
Strukturni ulaz/izlaz	900	207	76.7%
Stanje	286	138	100%
Raspodjela	900	205	34.5%

Kao što je prikazano u tablici 5.1, ispitivanje sigurnosnog podsustava okoline *PIE* traje znatno duže ako se za kriterij pokrivenosti odabere „strukturni ulaz/izlaz“ ili „raspodjela“. Pošto je ispitni model zasnovan na stanjima i prijelazima, potpuna pokrivenost prostora mogućih ispita najbrže se postiže ako se kao kriterij pokrivenosti odabere „prijelaz“ ili „stanje“. Pritom je izbor kriterija „prijelaz“ nešto povoljniji.

Postavke pretraživanja

Postavke pretraživanja utječu na način na koji se pretražuju mogući ispitni slučajevi. Postavke pretraživanja nude odabir parametara za ostvarenje što bolje kakvoće ispitivanja. Kako su danas vremenski rokovi izrade i ispitivanja sustava sve kraći te sustavi sve složeniji, nije uvijek moguće odabrati napredne tehnike pretraživanja mogućih ispitnih slučajeva. Alat CTG nudi parametre s kojima se može odabrati prikladna tehnika pretraživanja ispitnih slučajeva koja će rezultirati završavanjem ispitivanja u željenom vremenskom roku. Postoje dva dostupna parametra: *način pretraživanja* i *sposobnost planiranja ispita*.

Način pretraživanja ima dvije moguće vrijednosti: „optimirani“ i „nasumični“.

Kada je *način pretraživanja* podešen na „optimirani“ (pretpostavljena vrijednost), onda alat koristi kombinaciju tehnika pretraživanja unaprijed i stohastičkog planiranja kako bi optimirao ispitivanje na način da pokrije što veći dio prostora mogućih ispita što je brže moguće, ovisno o korištenom *kriteriju pokrivenosti*. Međutim, optimizacijski algoritam

pretraživanja je sam po sebi heuristički pa postoji teoretska mogućnost da će u nekim ispitnim slučajevima raditi nepravilno (iako je to rijetkost).

Kada je *način pretraživanja* podešen na „nasumični“, tada alat pretražuje model nasumično, rješavajući nedeterminističke izbore tako da svaki mogući put ima jednaku vjerojatnost odabira. „Nasumični“ način zauzima značajno manje memorije nego „optimirani“ način.

Parametar *sposobnost planiranja ispita* koristi se samo kada je postavljen „optimirani“ način pretraživanja, koji između ostalog koristi i tehniku pretraživanja unaprijed. Pretraživanje prostora mogućih ispita unaprijed troši jako puno vremena i tako usporava izvođenje vremenski kritičnih ispita. Zbog toga postoji nekoliko razina pretraživanja: „niska“, „srednja“, „visoka“ i „prilagodljiva“. Te razine pretraživanja ne mogu se eksplicitno kvantificirati u smislu koliko usporavaju ispitivanje. Potencijalno usporavanje može biti uzrokovano na više načina, npr. opterećenost procesora, struktura ispitnog modela, količina memorije na računalu. Vrijednosti „niska“, „srednja“ i „visoka“ odgovaraju fiksnim sposobnostima planiranja tijekom izvođenja ispita tako da su potencijalna doživljena usporavanja konstantna. Ako se postavi vrijednost „prilagodljiva“ tada komponenta za pretraživanje sama mijenja sposobnost pretraživanja na osnovu podatka koji govori koliki dio prostora potencijalnih ispita je već pokriven. Posljedica toga je usporavanje ispitivanja kada se dosegnu visoke vrijednosti pokrivenosti.

Tablica 5.2 prikazuje ovisnost vremenskog trajanja ispitivanja i broja izvršenih ispita u odnosu na odabrani *način pretraživanja i sposobnost planiranja ispita*. Na osnovu rezultata tablice 5.1 odabran je kriterij „prijelaz“ jer daje najučinkovitije ispitivanje. Za parametar *dubine analize pokrivenosti* odabrana je vrijednost „srednja“, a za faktor *zalihosti analize pokrivenosti* vrijednost *1*.

Tablica 5.2 Ovisnost vremenskog trajanja i ukupnog broja ispita o načinu pretraživanja te sposobnosti planiranja ispita

Način pretraživanja	**Sposobnost planiranja ispita**	**Vremensko trajanje (s)**	**Ukupan broj ispita**	**Pokrivenost prostora ispita**
Nasumični	-	80	45	100%
Optimirani	Niska	77	45	100%
	Srednja	166	87	100%
	Visoka	274	134	100%
	Prilagodljiva	270	130	100%

Kao što je prikazano u tablici 5.2, ispitivanje sigurnosnog podsustava okoline *PIE* traje najkraće ako se odabere „optimirani“ *način pretraživanja* sa „niskom“ sposobnosti planiranja ispita. Takve postavke u najkraće vremena i sa najmanjim brojem stvorenih ispita uspijevaju ostvariti stopostotnu pokrivenost prostora mogućih ispita. Veće sposobnosti planiranja ispita troše više vremena pa dublje tehnike pretraživanja unaprijed nisu potrebne za ispitivanje sigurnosnog podsustava okoline *PIE*.

6. Zaključak

Zbog stalnog povećanja složenosti programskih sustava, ispitivanje je danas iznimno bitan dio procesa izrade programskih sustava. S ciljem pojednostavljenja ispitivanja programskih sustava, razvijeno je nekoliko automatiziranih pristupa ispitivanju. Većina automatiziranih pristupa ispitivanju automatizira samo postupak izvođenja ispita, dok se postupak stvaranja ispita prepušta ljudima. Međutim, *ispitivanje zasnovano na modelu*, omogućuje i automatizirano stvaranje ispita te time ostvaruje bolja svojstva u ispitivanju sustava. *Ispitivanje zasnovano na modelu* koristi model za opis željenog ponašanja sustava. Na osnovu modela željenog ponašanja automatizirano se stvaraju ispitni slučajevi te se provodi ispitivanje rada sustava. *Ispitivanje zasnovano na modelu* koristi *pristup crne kutije*, što znači da se fokusira samo na ulaze i izlaze sustava, uz poznavanje očekivanog ponašanja sustava. Prema tome, struktura ostvarenja ispitivanog sustava potpuno je nevažna za *ispitivanje zasnovano na modelu*.

Središnji dio diplomskog rada prikazuje primjenu *ispitivanja zasnovanog na modelu* u ispitivanju sigurnosnog podsustava programirljive internetske okoline *PIE*. Okolina *PIE* ostvaruje programski model zasnovan na uslugama koji omogućuje izgradnju primjenskih sustava zasnovanih na uslugama.

Programsko ostvarenje ispitivanja sigurnosnog podsustava okoline *PIE* sastoji se od izrade ispitnog modela, izrade prilagodnika sustava i analize rezultata. Ispitni model izrađen je upotrebom alata CTG koji koristi UML modele i posebno definiran *CTG akcijski jezik*. Prilagodnik sustava omogućuje komunikaciju ispitne okoline s ispitivanim ostvarenjem sigurnosnog podsustava okoline *PIE*. Prilagodnik sustava prilagođuje apstraktne ispitne slučajeve automatski generirane alatom CTG u konkretne ispitne ulaze razumljive ispitivanom ostvarenju programskog sustava. Prilagodnik sustava ostvaren je upotrebom programskog jezika Java.

Analiza rezultata ispitivanja sigurnosnog podsustava okoline *PIE* otkrila je nekoliko pogreški ispitivanog sustava. Izvještaj i analiza provedenog ispitivanja poslužili su razvojnom timu sigurnosnog podsustava okoline *PIE* za ispravljanje pronađenih grešaka u ostvarenju programskog sustava.

Ispitivanje zasnovano na modelu pokazalo se kao korisna tehnika u ispitivanju sustava. Na osnovu iskustva stečenog tijekom izrade diplomskog rada, smatram *da ispitivanje zasnovano na modelu* treba uključiti u svaki proces izrade složenijih sustava. Učinkovitost ove tehnike posebno je značajna za sigurnosne sustave, gdje je prisutnost svake greške skupocjena.

7. Literatura

[1] SRBLJIĆ, S. 2002. *Jezični procesori 1 – Uvod u teoriju formalnih jezika, automata i gramatika*, Element, Zagreb.

[2] MILANOVIĆ, A. 2005. *Programski model zasnovan na uslugama*, doktorska disertacija, Fakultet elektrotehnike i računarstva, Sveučilište u Zagrebu, Zagreb.

[3] ŠKVORC, D. 2006. *Prividna mreža računalnih sustava zasnovanih na uslugama*, magistarski rad, Fakultet elektrotehnike i računarstva, Sveučilište u Zagrebu, Zagreb.

[4] PODRAVEC, M. 2006. *Otkrivanje i postavljanje usluga u sustavima zasnovanim na uslugama***,** magistarski rad, Fakultet elektrotehnike i računarstva, Sveučilište u Zagrebu, Zagreb.

[5] GAVRAN, I. 2006. *Korisnički jezik programskog modela zasnovanog na uslugama*, magistarski rad, Fakultet elektrotehnike i računarstva, Sveučilište u Zagrebu, Zagreb.

[6] CONFORMIQ SOFTWARE, 2005. *Conformiq Test Generator User Guide.*

[7] CONFORMIQ SOFTWARE, 2005. *Conformiq Test Generator Action Language Specification.*

[8] CONFORMIQ SOFTWARE, 2005. *Conformiq Test Generator Integration APIs.*

[9] Java 2 Platform Standard Edition 5.0 API Specification, (http://java.sun.com/j2se/1.5.0/docs/api/).

[10] OpenSSL, (http://www.openssl.org/).

[11] LACKOVIĆ, M. 2006. *Model Driven Architecture and Model Based Testing*, Technical Report, Ericsson ETK.

[12] DALAL, S. R., JAIN, A., KARUNANITHI, N., LEATON, J.M., LOTT, C.M., PATTON, G.C. 1999, *Model-Based Testing in Practise*, Proceedings ICSE'99 (ACM Press), Los Angeles.

[13] UTTING, M., PRETSCHNER, A., LEGEARD, B. 2006. *A Taxonomy of Model-Based Testing*, Working Paper Series ISSN 1170-487X.

[14] EL-FAR, I.K., WHITTAKER, J.A. 2001. *Model-based Software Testing*, Encyclopedia on Software Engineering, Wiley.

[15] BLACKBURN, M.R., BUSSER R.D., NAUMAN, A.M. 2003. *The Generations of Test Automation Tools: Understanding the Differences*, Software Productivity Consortium, NFP.

[16] BLACKBURN, M., BUSSER R., NAUMAN, A. 2004. *Why Model-Based Test Automation is Different and What You Should Know to Get Started*, Software Productivity Consortium, NFP.

[17] RUMPE, B. 2003. *Model-based Testing of Object-Oriented Systems*, Springer, Berlin.

[18] HARTMAN, A., KATARA, M., OLVOVSKY, S. 2006. *Choosing a Test Modeling Language: a Survey*, Proceedings Haifa Verification Conference, LNCS, Springer.

[19] UTTING, M. 2005. *Model-Based Testing*, The University of Waikato, New Zealand.

[20] ROBINSON, H. 2003. *Obstacles and opportunities for model-based testing in an industrial software environment*, Enterprise Management Division, Microsoft Corporation.

[21] DACS, 2003. *Model-Based Testing*, DACS Gold Practice Document Series.

[22] WHITTAKER, J.A. 1997. *Stochastic software testing,* Annals of Software Engineering, 4:115-131, J. C. Baltzer AG, Science Publishers, Red Bank, New Jersey, USA.

[23] PROWELL, S.J. 2000. *TML: a description language for Markov chain usage models*, Information and Software Technology, 42(12):835-844.

[24] EL-FAR, I.K. 1999. *Automated Construction of Software Behavior Models*, Master's Thesis, Florida Institute of Technology, Florida, USA.

[25] WALTON, G.H. 1995. *Generating Transition Probabilities for Markov Chain Usage Models*, Ph.D. Thesis, Department of Computer Science, University of Tennessee, Knoxville, Tennessee, USA.

[26] WALTON, G.H., POORE, J.H. 2000. *Generating transition probabilities to support model-based software testing*, Software: Practice and Experience, 30(10), pp. 1095-1106.

[27] WHITTAKER, J.A., AGRAWAL, K. 1994. *The application of cleanroom software engineering to the development of embedded control systems software*. Proceedings of the 1994 Quality Week.

[28] WHITE, L.J., COHEN, E.I. 1980. *A Domain Strategy for Computer Program Testing*. IEEE Transactions on Software Engineering, 6(3).

[29] TRETMANS, J., BRINKSMA, E. 2002. *Automated Model Based Testing*, Progress 2002 – 3rd Workshop on Embedded Systems.

[30] TRETMANS, J. 1996. *Test Generation with Inputs, Outputs and Repetetive Quiescence*, Concepts and Tools 17 (3).

[31] JARD, C., JERON, T. 2005. *TGV: theory, principles and algorithms*, Software Tools for Technology Transfer 7 (4).

[32] CLARKE, D., JERON, T., RUSU, V., ZINOVIEVA, E. 2002. *STG: A symbolic test generation tool*, Vol. 2280 of Springer LNCS.

[33] KOCH, B., GRABOWSKI, J., HOGREFE, D., SCHMITT, M. 1998. *AutoLink – a tool for automatic test generation from SDL specifications*, Proceedings IEEE Intl. Workshop on Industrial Strength Formal Specification Techniques (WIFT 1998).

[34] BERKENKOTTER, K., KIRNER, R. 2005. *Real-Time and Hybrid Systems Testing*, Broy et al. [2005].

[35] BROY, M., JONSSON, B., KATOEN, J.P., LEUCKER, M., PRETSCHNER, A. 2005. *Model-Based Testing of Reactive Systems*, LNCS no. 3472, Springer-Verlag.

[36] VAN LAMSWEERDE, A. 2000. *Formal specification: a roadmap*, Proceedings ICSE'00.

[37] KWAN, M. 1962. *Graphic programming using odd and even points*, Chinese Mathematics 1.

[38] FRANKL, P., WEYUKER, E. 1988. *An Applicable Family of Data Flow Testing Criteria*, IEEE TSE 14 (10).

[39] KOSMATOV, N., LEGEARD, B., PEUREUX, F., UTTING, M. 2004. *Boundary coverage criteria for test generation from formal models*, Proceedings 15th Intl. Symp. on SW Reliability Engineering.

[40] WALTON, G., POORE, J. 2000. *Generating transition probabilities to support model-based software testing*, Software: Practice and Experience 30 (10).

[41] MILLER, E. 1981. *Introduction to software testing technology*, Tutorial: Software Testing & Validation Techniques, IEEE Computer Society Press, pp:4-16

[42] DIJKSTRA, E. W. 1969. *Structural programming*, Software Engineering Techniques, Buxton and Randell, pp. 84-88.

[43] GOODENOUGH, J.B., GERHART, S.L. 1975. *Toward a theory of test data selection*, IEEE Transactions on Software Engineering, pp. 156-173.

[44] BEIZER, B. 1990. *Software Testing Techniques*, 2. izdanje, Van Nostrand Reinhold Company Limited, ISBN 0-442-20672-0.

[45] MYERS, G.J. 1979. *The Art of Software Testing*, Wiley.

[46] HOWDEN, W.E. 1980. *Functional Program Testing*, IEEE Transactions on Software Engineering, 6(2):162-169.

[47] HOWDEN, W.E. 1982. *Weak mutational testing and completeness of test sets*, IEEE Transactions on Software Engineering, 8(4):371-379.

[48] STOCKS, F., CARRINGTON, D. 1996. *A framework for specification-based testing*, IEEE Transactions on Software Engineering, 22(11):777-793.

[49] TURING, A.M. 1950. *Computing machinery and intelligence*, Mind, 59, 433-460.

[50] IEEE standard, (http://www.ieee.org/).

[51] SAFFORD, E.L. 2000. *Software Technology Conference: Software and Systems - Managing Risk, Complexity, Compatibility and Change*, Salt Lake City, Utah, USA.

[52] SPIVEY, M. 1992. *The Z Notation: a reference manual*, Prentice Hall, USA.

[53] JONES, C. 1990. *Systematic Software Development using VDM*, Prentice Hall, USA.

[54] LEAVENS, G.T., BAKER A.L., RUBY, C. 1999. *JML: A Notation for Detailed Design*, Kluwer Academic Publishers, Iowa, USA.

[55] BOUQUET, F., LEGEARD, B., PEUREUX, F., TORREBORRE, E. 2004. *Mastering Test Generation from Smart Card Software Formal Models*, Proceedings International Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices, Vol. 3362 of Springer LNCS, pp. 70–85.

[56] Matlab Simulink Verification and Validation, (http://www.mathworks.com/products/simverification/).

[57] PROWELL, S. 2003. *Jumbl: A tool for model-based statistical testing*, Proceedings HICSS'03, IEEE.

[58] COHEN, D., DALAL, S., FREDMAN, M., PATTON, G. 1997. *The AETG System: An approach to testing Based on Combinatorial Design*, IEEE TSE 23 (7):437–444.

[59] T-VEC Tester za Simulink, (http://www.t-vec.com/solutions/simulink.php).

[60] REACTIVE SYSTEMS, 2006. *Model-Based Testing and Validation with Reactis*.

[61] MaTeLo, (http://www.all4tec.net/).

[62] CleanTest, (http://www.cleansoft.com/).

[63] Pairwise Testing, (http://www.pairwise.org/).

8. Dodatak

8.1.1. Primjer korištenja CTG akcijskog jezika

Akcije koje se izvršavaju tijekom prijelaza ili stanja ispitnog modela mogu se predstaviti pozivom procedura. Procedure se opisuju *CTG akcijskim jezikom*. Procedura *printErrorMsg* ispisuje objašnjenje greške uz odluku *FAIL* kada se događa ispad ispitivanog sustava. Na primjeru dane procedure uočava se da je sintaksa *CTG akcijskog jezika* vrlo slična sintaksi programskog jezika C pa inženjeru ispitivaču nije potrebno trošiti puno vremena za upoznavanje sa *CTG akcijskim jezikom*.

```
integer proc printErrorMsg(integer checkCode)
{
   if (checkCode == 1)
      printf("IUT replied with wrong CA certificate format!");
   else if (checkCode == 2)
      printf("On dummy request IUT replied with certificate!");
   else if (checkCode == 3)
      printf("IUT replied with wrong nodeName value!");
   else if (checkCode == 4)
      printf("IUT replied with wrong keyLength value!");
   else if (checkCode == 5)
      printf("IUT replied with wrong duration value!");
   else if (checkCode == 6)
      printf("IUT replied with wrong publicKeyNode value!");
   else if (checkCode == 7)
      printf("IUT replied with signatureCA that could not be verified!");
   else if (checkCode == 8)
      printf("IUT replied with invalid CA certificate!");
   else if (checkCode == 9)
      printf("IUT replied with wrong NODE certificate format!");
   else if (checkCode == 10)
      printf("IUT replied with invalid NODE certificate!");
   else if (checkCode == 11)
      printf("IUT replied with wrong nodeB nodeName value!");
   else if (checkCode == 12)
      printf("IUT replied with wrong CA nodeName value in nodeB certificate!");
   else
      printf("CTG received unknown error code!");
   return 0;
};
```

8.1.2. Popis kratica

ABNF	*Augmented Backus-Naur Form*	prošireni oblik Backus-Naur
AETG	*Automatic Efficiant Test Generator*	automatski učinkoviti generator ispita
API	*Application Programming Interface*	programirljivo sučelje primjenskih sustava
CA	*Certificate Authority*	izdavatelj certifikata
CASE	*Computer-Aided Software Engineering*	računalno potpomognuto inženjerstvo programske potpore
CL	*Coopetition Language*	jezik za suradnju i natjecanje
CORBA	*Common Object Request Broker Architecture*	arhitektura posrednika zahtjeva jednostavnih objekata
CSR	*Certificate Signing Request*	zahtjev za potpisivanjem certifikata
CTG	*Conformiq Test Generator*	generator ispita Conformiq
HTML	*HyperText MarkUp Language*	označeni jezik hiperteksta
IP	*Internet Protocol*	Internet protokol
IUT	*Implementation Under Test*	ispitivano programsko ostvarenje
JDBC	*Java Data Base Connectivity*	Java pristup bazama podataka
JML	*Java Modeling Language*	Java jezik za modeliranje
JUMBL	*Java Usage Model Builder Library*	Java knjižnica za izgradnju modela korištenja
LTG	*Leirios Test Generator*	generator ispita Leirios
ODBC	*Open Data Base Connectivity*	otvoreni pristup bazama podataka
MBT	*Model Based Testing*	ispitivanje sustava modelom
MC/DC	*Modified Condition/Decision Coverage*	promijenjeni uvjet/pokrivenost odluke
MTBF	*Mean Time Before Failure*	srednje vrijeme do pojave kvara
NASA	*National Aeronautics and Space Administration*	državna centar za aeronautiku i svemir
PIE	*Programmable Internet Environment*	programirljiva Internet okolina
PKI	*Public Key Infrastructure*	infrastruktura javnih ključeva
RAM	*Random Access Memory*	memorija sa slobodnim pristupom
RFC	*Request for Comments*	zahtjev za komentarima
RPC	*Remote Procedure Call*	udaljeni poziv procedure
SQL	*Structured Query Language*	strukturirani jezik upita
SSCL	*Simple Service Composition Language*	jezik za jednostavnu kompoziciju usluga
STG	*Symbolic Test Generator*	generator simboličkih ispita

SUT	*System Under Test*	sustav koji se ispituje
TCP	*Transmission Control Protocol*	protokol upravljanja prijenosa
TGV	*Test Generation with Verification technology*	stvaranje ispita sa verifikacijskom tehnologijom
TML	*The Modelling Language*	jezik za modeliranje
T-VEC	*Test Vector Generator*	generator ispitnih vektora
UML	*Unified Modelling Language*	opći jezik za modeliranje
VDM	*Vienna Development Method*	bečka razvojna metoda
XML	*eXtensible Markup Language*	jezik sa dodatnim označavanjem

8.1.3. Popis stranih izraza

all-quadruples	sve četvorke
all-triples	sve trojke
application	primjenski sustav
application service	primjenska usluga
binary semaphore	binarni semafor
black box approach	pristup crne kutije
case studies	studije izvodivosti
Certificate Manager	Upravitelj certifikata
Certificate Service	Usluga izdavanja certifikata
combinatorial testing	kombinacijsko ispitivanje
coopetition service	usluga za suradnju i natjecanje
counting semaphore	opći semafor
coverage	pokrivenost
datum	ispitni podatak
debugging	traženje grešaka
design model	model oblikovanja
distributed program	raspodijeljeni program
embedded system	ugrađeni sustav
environment scenario	scenarij okoline
engineering process	inženjerski proces
error	greška
evaluation-oriented	usmjeren vrednovanju
Exchange Protocol	protokol izmjene
event channel	usmjernik događaja
forward search	pretraživanje unaprijed
guard	uvjet prijelaza
implementation	programsko ostvarenje
irreducible	neskrativi
lookup table	pregledna tablica
mailbox	poštanski pretinac
main scenario	glavni scenarij
n-way search algorithm	algoritam pretraživanja u *n* smjerova
on line testing	stvaranje ispita tijekom izvođenja

off line testing	stvaranje ispita prije izvođenja
overlay network	prividna logička mreža
pair-wise algorithm	algoritam uparivanja
partition criterion	kriterij raspodjele
port	priključnica
precondition/postcondition	uvjet/posljedica
proxy	zastupnik
publish/subscribe	objava/pretplata
requirement engineer	inženjer koji definira zahtjeve sustava
reverse compiler	unazadni prevoditelj
service	usluga
synchronization	usklađivanje rada
system adapter	prilagodnik sustava
testbed adapter	prilagodnik ispitne okoline
test oracle	mehanizam provjere ispravnosti izlaza
test purpose	ispitni sadržaj
usage model	model korištenja
use case diagram	dijagram korištenja
verified	provjeren
white box approach	pristup bijele kutije