УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА У НОВОМ САДУ

Ђорђе Миљковић

Једно решење алата за аутоматизацију испитивања графичке корисничке спреге

МАСТЕР РАД

Нови Сад, 2013

УНИВЕРЗИТЕТ У НОВОМ САДУ ● **ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА**
21000 НОВИ САД, Трг Доситеја Обрадовића 6

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА

Редни број, **РБР**:	
Идентификациони број, **ИБР**:	
Тип документације, **ТД**:	Монографска документација
Тип записа, **ТЗ**:	Текстуални штампани материјал
Врста рада, **ВР**:	Дипломски – мастер рад
Аутор, **АУ**:	**Ђорђе Миљковић**
Ментор, **МН**:	**Ковачевић др Јелена**
Наслов рада, **НР**:	**Једно решење алата за аутоматизацију испитивања графичке корисничке спреге**
Језик публикације, **ЈП**:	Српски / латиница
Језик извода, **ЈИ**:	Српски
Земља публиковања, **ЗП**:	Република Србија
Уже географско подручје, **УГП**:	Војводина
Година, **ГО**:	**2013**
Издавач, **ИЗ**:	Ауторски репринт
Место и адреса, **МА**:	Нови Сад; трг Доситеја Обрадовића 6
Физички опис рада, **ФО**: (поглавља/страна/ цитата/табела/слика/графика/прилога)	**7/48/0/5/18/1/0**
Научна област, **НО**:	Електротехника и рачунарство
Научна дисциплина, **НД**:	Рачунарска техника
Предметна одредница/Кључне речи, **ПО**:	**ГУИ, ГУИ Плејер, испитивање, ХМЛ, ЛУА**
УДК	
Чува се, **ЧУ**:	У библиотеци Факултета техничких наука, Нови Сад
Важна напомена, **ВН**:	
Извод, **ИЗ**:	**У раду је реализовано решење за аутоматско испитивање графичке корисничке спреге. Постојећи алати за ручно испитивање не одговарају изазовима развоја модерних производа, те је потребно постићи што већу аутоматизацију. Циљ рада је уштеда времена испитивања, као и смањење утицаја људског фактора на појаву грешке приликом испитивања. Функционалност реализованог решења потврђена је потпуном аутоматизацијом испитивања на графичком алату за развој програмске подршке за аудио циљне платформе компаније Cirrus Logic.**
Датум прихватања теме, **ДП**:	
Датум одбране, **ДО**:	

Чланови комисије, **КО**:	Председник:	**Башичевић др Илија**	
	Члан:	**Струхарик др Растислав**	Потпис ментора
	Члан, ментор:	**Ковачевић др Јелена**	

UNIVERSITY OF NOVI SAD ● **FACULTY OF TECHNICAL SCIENCES**
21000 NOVI SAD, Trg Dositeja Obradovića 6

KEY WORDS DOCUMENTATION

Accession number, **ANO**:			
Identification number, **INO**:			
Document type, **DT**:		Monographic publication	
Type of record, **TR**:		Textual printed material	
Contents code, **CC**:		Master Thesis	
Author, **AU**:		**Đorđe Miljković**	
Mentor, **MN**:		**Jelena Kovačević, PhD**	
Title, **TI**:		**One solution of automated Graphical User Interface testing tool**	
Language of text, **LT**:		Serbian	
Language of abstract, **LA**:		Serbian	
Country of publication, **CP**:		Republic of Serbia	
Locality of publication, **LP**:		Vojvodina	
Publication year, **PY**:		**2013**	
Publisher, **PB**:		Author's reprint	
Publication place, **PP**:		Novi Sad, Dositeja Obradovica sq. 6	
Physical description, **PD**: (chapters/pages/ref./tables/pictures/graphs/appendixes)		**7/48/0/5/18/1/0**	
Scientific field, **SF**:		Electrical Engineering	
Scientific discipline, **SD**:		Computer Engineering, Engineering of Computer Based Systems	
Subject/Key words, **S/KW**:		**GUI, GUIPlayer, testing, XML, Lua**	
UC			
Holding data, **HD**:		The Library of Faculty of Technical Sciences, Novi Sad, Serbia	
Note, **N**:			
Abstract, **AB**:		**The paper describes the solution for automated testing of graphical user interfaces. Existing tools for manual testing does not respond to the challenge of developing modern products, and it is needed to achieve greater automation of testing. The aim of the tool is saving time and reducing the human impact on testing process. Functionality is confirmed through completly automated testing tool for graphical software development for Cirrus Logic DSP audio platform.**	
Accepted by the Scientific Board on, **ASB**:			
Defended on, **DE**:			
Defended Board, **DB**:	President:	**Ilija Bašičević, PhD**	
	Member:	**Rastislav Struharik, PhD**	Menthor's sign
	Member, Mentor:	**Jelena Kovačević, PhD**	

SADRŽAJ

SPISAK SLIKA

SPISAK TABELA

SKRAĆENICE

GUI	- *Graphical User Interface*, Grafička korisnička sprega
DSP	- *Digital Signal Processor*, Digitalni signal procesoor
XML	- *Extensible Markup Language,* Proširivi jezik za označavanje
API	- *Application programming interface,* Sprega za programiranje aplikacija
AUT	- *Application Under Test,* Aplikacija koja se ispituje
BBT	- *Black Box Testing,* Ispitivanje metodom crne kutije
DUT	- *Device Under Test,* Uređaj koji se ispituje
DLL	- *Dinamic Link Library*, Biblioteka za dinamičko povezivanje

1. Uvod

U okviru ovog rada realizovan je alat za automatizaciju ispitivanja grafičke korisničke sprege na Microsoft Windows XP operativnom sistemu. Alat se sastoji od tri nezavisna alata razvijena kako bi se lakše izvršila potpuna automatizacija ispitivanja.

Grafička korisnička sprega (*eng. GUI – Graphical User Interface*) je način komunikacije sa računarom manipulacijom grafičkim elementima i dodacima u vidu tekstualnih poruka i obaveštenja. Kako bi korisnicima komunikacija bila što lakša, GUI se bazira na objektima poznatim u stvarnom životu kao što su prozori, ikonice i meniji kojima se može manipulisati mišem ili tastaturom.[1]

Glavna prednost grafičke korisničke sprege u odnosu na komandnu spregu je da čini računarske operacije intuitivnijim, što omogućava lakše učenje i korišćenje računarskih operacija. Na primer, mnogo je lakše za novog korisnika da premesti datoteku iz jednog direktorijuma u drugi prevlačenjem ikone mišem, nego da pamti i upisuje komande u terminalski prostor. Takođe još jedna od prednosti grafičke korisničke sprege je činjenica da pruža korisnicima informaciju o efektima svake izvršene akcije. Na primer, kada korisnik izbriše jednu ikonu koja predstavlja datoteku, ikona odmah nestane, što potvrđuje da je datoteka izbrisana (ili bar poslata u kantu). Ovo je u suprotnosti sa komandnom spregom u kojoj korisnik unosi komandu *Delete* (uključujući ime datoteke koju je potrebno izbrisati), ali ne prima nikakvu povratnu informaciju koja pokazuje da je datoteka izbrisana.

GUI korisnicima omogućava veliku slobodu u komunikaciji sa aplikacijama. Time se korisnicima olakšava rad, ali otežava posao razvojnim programerima i ispitivačima. Budući da korisnici na GUI-u mogu vršiti interakciju na razne načine, teško je osigurati da program ispunjava sve funkcionalne zahteve (ispravnost) i nefunkcionalne zahteve (upotrebljivost) za svaku moguću interakciju.

Cilj samog rada je bio da se smanji vreme izvršavanja velikog broja ispitnih slučajeva, smanjenje uticaja ljudskog faktora na pojavu greške prilikom ispitivanja i da se omogući automatsko izvršavanje regresivnih ispitnih slučaja. Funkcionalnost rešenja potvrđena je opsežnim ispitivanjem na grafičkom alatu za razvoj programske podrške za audio ciljne platforme kompanije Cirrus Logic, a rezultati ispitivanja ulivaju nadu da daljim razvojem možemo dobiti kompletan i stabilan alat koji će biti konkurentan na tržištu.

Ovaj rad je sačinjen od sedam poglavlja. Drugo poglavlje sadrži teorijske osnove koje su potrebne za razumevanje koncepta rešenja i konkretne implementacije. U trećem poglavlju je dat koncept rešenja. Četvrto poglavlje daje prikaz programskog rešenja pojedinačnih alata, kao i primer izvršavanja jednog ispitnog slučaja. Peto poglavlje daje opis ispitivanja i verifikacije rešenja. U šestom poglavlju je u okviru zaključka dat pregled svega urađenog u radu kao i ideje za dalji razvoj. Lista korišćene literature data je u sedmom poglavlju.

2. Teorijske osnove

Ispitivanje je proces tehničke istrage, čiji je cilj da otkrije informacije vezane za kvalitet proizvoda, naravno, uzimajući u obzir kontekst funkcionisanja proizvoda. Ovaj proces uključuje izvršavanje programa ili aplikacije sa namerom pronalaženja grešaka. Ispitivanje obezbeđuje kritiku i komparaciju za stanje i ponašanje proizvoda u odnosu na specifikaciju. Postoje mnogi pristupi ispitivanju softvera, ali je efektivno ispitivanje kompleksnih proizvoda u osnovi proces istrage, a ne samo pitanje kreiranja i praćenja rutinske procedure.

Jedna od definicija ispitivanja je: „Proces ispitivanja softvera u cilju njegove procene“, gde se ispitivanje odnosi na operacije koje ispitivač pokušava da izvrši nad proizvodom, a proizvod odgovara na to ponašanjem izazvanim probama ispitivača [2].

Ispitivanje softvera je proces izvršavanja programa sa ciljem pronalaženja grešaka. Ovaj proces pruža objektivan, nezavisan pogled na softver kako bi se omogućilo klijentu da razume rizike implementacije tog softvera. Softversko ispitivanje se danas vidi kao aktivnost koja obuhvata ceo proces razvoja i održavanja, i predstavlja važan deo kompletne izgradnje softvera. Planiranje ispitivanja treba da počne sa ranom fazom razvojnog procesa, a ispitni planovi i procedure moraju biti sistematski i kontinualno razvijani, i po potrebi menjani[3].

Ciljevi ispitivanja softvera su:

- verifikacija i validacija – proverava se da li je softver saglasan sa specifikacijom zahteva, što ne znači uvek da je softver tehnički ispravan, pouzdan i bezbedan.
- Poboljšanje kvaliteta softvera
- Procena pouzdanosti.

Ispitni slučajevi se mogu klasifikovati na više načina, ali je najpopularnija klasifikacija prema vrsti zahteva na koji se odnose. Prema ovom kriterijumu, ispitni slučajevi se dele na:

1. **Ispitni slučajevi za proveru funkcionalnosti** – ovim ispitnim slučajevima se ispituje da li program pruža funkcionalnost koja je opisana zahtevima. Na ovaj način se ne proverava brzina rada programa ili bezbednost, već samo da li program (ili neki njegov deo) bez greške radi ono što je opisano u funkcionalnim zahtevima. Ovi ispitni slučajevi se dele prema veličini dela programa na koji se odnose na:
   - pojedinačne ispitne slučajeve (*eng. unit test*) – odnose se na pojedinačne klase
   - integracione ispitne slučajeve(*eng. integration test*) – ispituje se više klasa koje se međusobno pozivaju u radu
   - sistemske ispitne slučajeve(*eng. system test*) – ispitivanje celog programa od strane programera
   - ispitne slučajeve prihvatljivosti (*eng. acceptance test*) – ispitivanje celog programa od strane krajnjeg korisnika
   - regresivno ispitivanje (*eng. regression test*) - na osnovu jednom razvijenog ispitnog slučaja više puta se sprovodi ispitivanje programa (tipicno nakon neke izmene u programu da bi se utvrdilo da nisu pokvarene funkcionalnosti programa).
2. **Ispitni slučajevi za proveru nefunkcionalnih karakteristika** – ovim ispitnim slučajevima se ispituje da li su nefunkcionalni zahtevi ispunjeni. Nefunkcionalni zahtevi se najčešće tiču brzine rada ili skalabilnosti programa (mogućnost opsluživanja većeg broja korisnika istovremeno zadovoljavajućom brzinom), ali se mogu ticati i bezbednosti programa, kompatabilnosti i drugih nefunkcionalnih karakteristika. Na osnovu toga i ovi ispitni slučajevi se mogu podeliti na:
   - *Ispitivanje opterećenja* – ispitivanje performansi kojima se procenjuju operativna ograničenja nepromenjivog sistema pod različitim opterećenjima ili različitih konfiguracija sistema pri istom opterećenju. Najčešće se mere protok i vreme odziva.
   - *Ispitivanje bezbednosti* – ispitivanje da li su funkcije dostupne onim i samo onim korisnicima kojima su i namenjene.
   - *Ispitivanje integriteta* – robusnost(otpornost na otkaze)
   - *Ispitivanje u stresnim uslovima* – ispitivanje pouzdanosti sistema pod nenormalnim uslovima (velika opterecenja sistema, nedovoljno memorije ili drugih resursa, neraspoloživi servisi i sl.).
   - *Ispitivanje zagušenja* – provera da li sistem može da zadovolji višestruke zahteve razlicitih aktera za istim resursom.
   - *Ispitivanje instalacije* – ispitivanje instaliranja softvera na razlicitim sistemima i u razlicitim situacijama (npr. prekid napajanja ili nedovoljno prostora na disku).

Pored ove klasifikacije, postoji i klasifikacija ispitnih slučajeva prema kriterijumu poznavanja koda koji se ispituje. Prema ovom kriterijumu ispitni slučajevi se dele na:

1. Ispitivanje metodom bele kutije (White box testing)
2. Ispitivanje metodom crne kutije (Black box testing)

Takođe, ispitni slučajevi se mogu klasifikovati prema načinu izvršavanja ispitnih slučajeva. Prema ovom kriterijumu ispitni slučajevi se dele na:

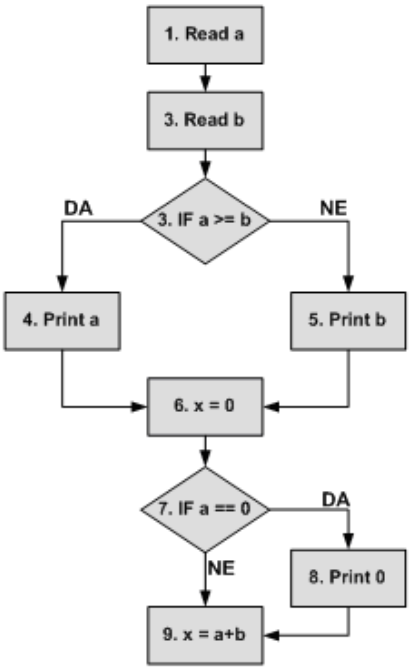
1. Ručno ispitivanje
2. Automatsko ispitivanje

2.1 Ispitivanje metodom bele kutije

Ovo ispitivanje proverava i analizira izvorni kod i zahteva dobro poznavanje programiranja, odgovarajućeg programskog jezika, kao i projekta konkretnog softverskog proizvoda. Plan ispitivanja se određuje na osnovu elemenata implementacije softvera, kao što su programski jezik, logika i stilovi kodiranja. Testovi se izvode na osnovu strukture programa. Kod ove metode postoji mogućnost provere skoro celokupnog koda, na primer proverom da li se svaka linija koda izvršava barem jednom, proverom svih funkcija ili proverom svih mogućih kombinacija različitih programskih naredbi. Specifičnim testovima može se proveravati postojanje beskonačnih petlji ili koda koji se nikada ne izvršava[4].

Jedan od načina da se osmisli dobar skup ispitinih slučajeva bele kutije je na osnovu kontrole toka programa.

Graf kontrole toka programa je prikazan na slici 2-1. Da bi bio osiguran odgovarajući skup ispitnih slučajeva potrebno je uzeti u obzir različite aspekte ovog grafa. Adekvatnost ispitinih slučajeva se meri sa metrikom koja se zove pokrivenost. Na primeru jednostavnog koda (slika 2-2) u daljem tekstu biće prikazani različiti načini pokrivenosti [5].



Slika 2-1 Graf toka – ispitivanje metodom bele kutije

```
1. Read a
2. Read b
3. if (a>=b) then
4.          print a
5. ELSE
6.          print b
7. ENDIF
8. x = 0
9. if (a==0) then
10.         print 0
11. ENDIF
12. x=a+b
```

Slika 2-2 Primer jednostavnog koda za ilustraciju izračunavanja pokrivenosti

2.1.1 Pokrivenost iskaza

Pokrivenost iskaza je procenat iskaza koji su izvršeni od strane ispitinih slučajeva. Cilj je ostvariti 100% pokrivenosti iskaza kroz ispitivanje. Da bismo postigli pokrivenost iskaza, moramo izabrati test primere koji izvršavaju svaku liniju koda u programu. Procenat izvršenih iskaza se može izračunati preko sledeće formule:

Pokrivenost iskaza = (Ukupan broj izvršenih iskaza) / (Ukupan broj izvršnih iskaza u programu) x 100

Jasno je da kako tip iskaza napreduje od običnog sekvencijalnog iskaza do if-then-else petlji itd, tako raste i broj ispitinih slučajeva potrebnih za pokrivenost iskaza.

Pokrivenost iskaza je najneefikasnija tehnika kontrole toka, jer da bi se postigao, ispitivač mora da dovoljnim brojem ispitnih slučajeva primora svaku liniju koda da se izvrši bar jednom.

Primer:

Ispitni slučaj	Vrednost a	Vrednost b	Uslov1, Uslov2
1	0	-1	True, True
2	0	2	False, True

Tabela 2-1 Primer pokrivenosti iskaza

U ispitnom slučaju 1 (a=0, b=-1), izvršeno je 10 iskaza od ukupno 12. Da bi bilo ostvareno 100% pokrivenosti iskaza koristi se dodatni slučaj - ispitini slučaj 2: a=0, b=2 (False, True) sa očekivanom povratnom vrednošću 2, 0.

2.1.2 Pokrivenost odluka

Pokrivenost odluka je procenat tačaka odluke (logičkih izraza) programa koji su ocenjeni i tačnim i netačnim u ispitinim slučajevima. Pokrivenost odluka je bolja tehnika u poređenju sa tehnikom pokrivenosti iskaza jer teži da ide dublje u kod i podrazumeva odgovarajući broj ispitinih slučajeva da osigura izvršavanje odluke ili grane najmanje jednom. U većini proizvoda, pokrivenost odluka se posmatra kao minimum ispitivanja pokrivenosti.

Lako je utvrditi koliko program ima grana jer je grana ishod odluke. Logička odluka kao „IF - iskaz“ ima dva ishoda ili grane (tj. Tačno i Netačno).

Primer:

Mali program prikazan na slici 2-2 ima dve tačke odluke, jednu na liniji 3 i drugu na liniji 9. Za pokrivenost grana/tačaka odluke ocenjuje se ceo logički izraz kao tačan ili netačan čak i ako sadrži više logičkih „i“ ili logičkih „ili“ operatora. Potrebno je obezbediti da svaki od ovih uslova bude ispitan i kao tačan i kao netačan. Pokrivenost grana prikazana je u sledećoj tabeli:

Broj linije	Uslov	TRUE	FALSE
3	(a>=b)	Ispitni slučaj 1 a=0, b=-1	Ispitni slučaj 2 a=1, b=2
9	(a==0)	Ispitni slučaj 1 a=0, b=-1	Ispitni slučaj 2 a=1, b=2

Tabela 2-2 Primer pokrivenosti odluke

Kroz dva ispitina slučaja a=0, b=-1(True, True) i a=1, b=2(False, False) ostvaruje se 100% pokrivenost odluka.

U mnogo slučajeva cilj je postići stoprocentnu pokrivenost, dok se u većim sistemima praktikuje samo 75%-85%, a u sistemima od 10 miliona linija koda ili više, praktikuje se samo 50% pokrivenosti odluka.

2.1.3 Pokrivenost putanja

Ispitivanje putanja je sredstvo kojim se obezbeđuje da sve nezavisne putanje modula budu ispitane. Nezavisna putanja je bilo koja putanja u kodu koja uvodi u najmanje jedan novi skup procesnih naredbi ili u novi uslov. Program ili deo programa može početi od početka i krenuti bilo kojom putanjom do njegovog završetka. Pokrivenost putanja se može izračunati korišćenjem sledeće formule:

Pokrivenost putanja = (Broj izvršenih putanja) / (Ukupan broj putanja u programu) x 100

Potrebno je napomenuti da je program koji izvršava pogrešne zahteve i dalje pogrešan program, čak i ako je u potpunosti ispitan sa 100% pokrivenosti putanja.

Primer:

Posmatrajući graf toka, moguće je navesti nekoliko različitih putanja:

a) 1-2-3-5-6-7-9

b) 1-2-3-4-6-7-9

c) 1-2-3-4-6-7-8-9

d) 1-2-3-5-6-7-8-9

Izvršavanjem putanja, ne uzimajući u obzir broj iskaza u njima, većina scenarija bi bila pokrivena (u ovom primeru - svi).

2.1.4 Prednosti i nedostaci ispitivanja metodom bele kutije

Prednosti ispitivanja metodom bele kutije:

- Poznavanjem unutrašnje strukture (tj. samog koda) vrlo lako je uvideti koji tip ulaznih podataka može pomoći u efikasnom ispitivanju aplikacije,
- pomoć u optimizaciji koda,
- uklanjenje suvišnih linija koda koje mogu izazvati greške.

Nedostaci ispitivanja metodom bele kutije:

- Pošto je poznavanje unutrašnje strukture uslov, potreban je vešt ispitivač koji će izvesti ovu vrstu ispitivanja. Ovo zahteva dodatne troškove,
- gotovo je nemoguće pogledati u svaki deo koda da bi se otkrile skrivene greške, to može stvoriti problem koji rezultira neuspehom u primeni aplikacije.

2.2 Ispitivanje metodom crne kutije

Analizira se samo izvršavanje specificiranih funkcija i vrši se provera ulaznih i izlaznih podataka. Tačnost izlaznih podataka proverava se na osnovu specifikacije zahteva za softver [6]. U ovim ispitnim slučajevima se ne vrši analiza izvornog koda. Problem funkcionalnog testiranja može da se pojavi u slučaju dvosmislenih zahteva i nemogućnosti opisivanja svih načina korišćenja softvera. Na osnovu nekih ispitivanja skoro 30% svih grešaka u kodu posledica su problema nepotpunih ili neodređenih specifikacija.[6]

Ispitivanjem metodom crne kutije pokušavaju se pronaći greške u sledećim kategorijama:

- neispravna ili nepotpuna funkcija,
- greška sprege,
- greška u strukturi podataka ili u pristupu spoljnoj bazi podataka,
- greška performanse,
- greška inicijalizacije i greška izvršavanja.

Primenom metode crne kutije izrađuje se skup ispitnih slučajeva koji ispunjavaju zahteve:

- Smanjivanja broja ispitnih slučajeva na mogućnost razumnog testiranja, pri čemu je obuhvaćeno ispitivanje kompletne funkcionalnosti
- Ispitni slučajevi koji prikazuju prisutnost ili odsutnost grešaka.

Klasične tehnike koje se koriste u metodi ispitivanja „crne kutije“ su:

- Podela na klase ekvivalencije
- Analiza graničnih vrednosti

2.2.1 Podela na klase ekvivalencije

Kako bi se smanjili troškovi ispitivanja nema potrebe pisati nekoliko ispitnih slučajeva koji ispituju isti aspekt našeg programa. Dobar ispitni slučaj otkriva drugačije klase grešaka od onih koje su otkrili prethodni. Podela na klase ekvivalencije je strategija koja smanjuje broj ispitnih slučajeva koji moraju biti napisani i deli ulazne domene programa u klase. Za svaku od tih klasa ekvivalencije skup podataka bi trebao biti tretiran isto od strane ispitivanog modula i trebao bi proizvesti isti odgovor. Ispitni slučajevi bi trebali biti dizajnirani tako da se ulazi nalaze unutar tih klasa.

Prilikom određivanja klasa ekvivalencije posmatraju se svi uslovi vezani za ulaze programa koji proizilaze iz specifikacije. Klasa ekvivalencije može biti jedan brojčani podatak ili grupa brojčanih podataka. Klase ekvivalencije koje sadrže samo važeće stanje, tj. dozvoljene situacije zovu se važeće ili validne klase, a klase ekvivalencije koje obuhvataju sve ostale situacije zovu se nevažeće ili nevalidne klase.

Dva glavna koraka koja treba preduzeti u korišćenju podele na klase ekvivalencije su:

- Identifikovati klase ekvivalencije za ulaz i izlaz. Izvesti najmanje dve klase za svaki ulazni i izlazni uslov koji je opisan u specifikaciji:
    - Jednu klasu koja zadovoljava uslov – važeću klasu
    - Jednu klasu koja ne zadovoljava uslov – nevažeću klasu
- Dizajnirati ispitne slučajeve bazirane na izvedenim klasama ekvivalencije.

Neka uputstva za identifikaciju klasa su:

1. Ako ulazni uslov podrazumeva opseg vrednosti, definišu se jedna važeća i dve nevažeće klase ekvivalencije.
2. Ako ulazni uslov zahteva određenu vrednost, definišu se jedna važeća i dve nevažeće klase ekvivalencije.
3. Ako ulazni uslov podrazumeva člana nekog skupa, definišu se jedna važeća i jedna nevažeća klasa ekvivalencije.
4. Ako je ulazni uslov logički uslov, definišu se jedna važeća i jedna nevažeća klasa ekvivalencije.

2.2.2 Analiza graničnih vrednosti

Prilikom pisanja koda programeri često prave greške na granicama ulaznih domena i zbog toga se treba usredsrediti na ispitivanje tih granica. Granična vrednost se definiše kao vrednost podataka koja odgovara minimalnom ili maksimalnom ulazu određenom za sistem ili komponentu.

Iz specifikacije komponente treba izdvojiti ulazne i izlazne vrednosti, a zatim ih grupisati u skupove sa identifikovanim granicama. Svaki skup ili niz sadrži vrednosti koje bi komponenta trebala obraditi na isti način. Podela na opsege podataka za ispitivanje je objašnjena u tehnici podele na klase ekvivalencije.

U analizi graničnih vrednosti, za različite opsege vrednosti, koriste se sledeći tipovi ispitnih slučajeva:

a) Dva važeća (validna) ispitna slučaja koja pokrivaju krajnje granice,

b) Dva nevažeća (nevalidna) ispitna slučaja neposredno preko granica opsega.

Primer:

Uzmimo za primer program koji kao ulaz „Plata“ koristi vrednosti u opsegu od 120 do 350 sa sledećim pretpostavkama:

1. Ukoliko je Plata do 150, taksa = 0,
2. Ukoliko je Plata od 151 do 250 – taksa iznosi 18% od ukupne Plate,

3. Ukoliko je Plata preko 250 – taksa = 20% od Plate.

Važeći ulazni domeni se mogu podeliti u tri važeće klase ekvivalencije:
Klasa K1: Vrednosti u opsegu od 120 do 150
Klasa K2: Vrednosti u opsegu od 151 do 250
Klasa K3: Vrednosti veće od 250

Analiza graničnih vrednosti je efikasan pristup dizajnu ispitnih slučajeva koji može otkriti većinu najčešćih grešaka koje se javljaju prilikom programiranja.

2.2.3 Prednosti i nedostaci ispitivanja metodom crne kutije

Prednosti ispitivanja metodom crne kutije:
- Ispitivanje može biti ne-tehničko (od ispitivača se ne zahtevaju tehnička znanja).
- Ovo ispitivanje će najverovatnije pronaći one greške koje bi i korisnik pronašao.
- Ispitivanje pomaže u identifikovanju nejasnoća i protivrečnosti funkcionalnih specifikacija.
- Ispitni slučajevi mogu biti izrađeni po završetku funkcionalnih specifikacija.

Nedostaci ispitivanja metodom crne kutije:
- Zbog nepoznavanja unutrašnje strukture koda, može se propustiti ispitivanje nekog dela koda
- detaljno ispitivanje svih kombinacija različitih ulaznih parametara za većinu aplikacija praktično ne izvodljivo.

2.3 Ručno ispitivanje

Ručno ispitivanje sadrži listu ispitnih slučajeva koje izvršava čovek i čiji se rezultati unose ručno. Za svaki ispitni slučaj iz liste postoji dodatno polje koje predstavlja rezultat izvršavanja pojedinačnog ispitnog slučaja. Rezultat kod ručnog ispitivanja može imati jednu od sledećih vrednosti:

- *Pass* – prošao.
- *Fail* – pao.
- *Pass with exception* – ako su ispunjene bitne stavke iz Pass/Fail kriterijuma, a neke marginalne reakcije sistema ne odgovaraju opisu očekivanih reakcija.

- *Blocked* – ako ispitni slučaj nije izvršen zbog zaglavljivanja sistema na nekom od prethodnih koraka, dakle ako se u izvršenju akcija nije stiglo do početnog stanja.
- *Dropped* – ispitni slučaj neće biti izvršen u datom ciklusu ispitivanja zbog nekog drugog, spoljašnjeg, razloga (na primer nedostatak vremena).
- *Pending (not executed)* – ispitni slučaj još nije izvršen.
- *None* – ispitni slučaj je izvršen, ali nema jasno definisan rezultat (nije neophodno da ga ima). Na primer, test kojim se određuje maksimalni kapacitet sistema, ukoliko se uspešno izvrši (nije fail), kao rezultat ima brojnu vrednost. Kako se brojna vrednost ne može svrstati ni u jedan od prethodno navedenih tipova rezultata, rezultatu se može dodeliti tip *None*.

Prilikom dodavanja ispitnih slučajeva u ručno izvršenje testa rezultat svakog pojedinačnog ispitnog slučaja dobija vrednost *Pending*.

2.4 Automatsko ispitivanje

Pod automatizacijom ispitnog slučaja se podrazumeva korišćenje softvera za kontrolu izvršenja ispitnih slučajeva, poređenje stvarnog ishoda sa predviđenjim ishodom, postavljanje probnih preduslova, kao i kontrola izveštaja ispitivanih funkcija. Najčešće, automatizacija ispitnih slučajeva podrazumeva automatizaciju ručnog ispitivanja procesa tj. formalizaciju ručnog ispitivanja. Iako ručni ispiti mogu naći mnoge nedostatke u softverskoj aplikaciji, to je naporan i dugotrajan proces. Pored toga ne mogu biti efikasni u pronalaženju određenih klasa grešaka. Automatizacija ispitnog slučaja je proces pisanja računarskog programa koji će raditi ispitivanje, u suprotnom se ispitivanje radi ručno. Kada ispitni slučajevi budu automatizovani, mogu biti pokrenuti brzo. Ovo je često najpovoljnija metoda za softverske proizvode koji imaju dug period održavanja, jer čak i najmanje promene u kodu mogu uzrokovati kvar aplikacije, iako je aplikacija u predhodnim verzijama radila normalno.

Postoje dva opšta pristupa automatizaciji testova:

– Kodom upravljivo (*eng. Code-driven*) ispitivanje. Obična javna sprega (*eng. public interface*) za klase, module ili biblioteke koje se ispituju raznim ulaznim argumentima, da bi se proverili da li su vraćeni rezultati tačni.

- Grafička korisnička sprega (*GUI*) ispitivanje. Ispitivanje u okviru korisničke sprege generiše događaje kao što je pritiskanje tastera miša (klikova), i prati promene koje su rezultat u korisničkoj sprezi, radi potvrde da je ponašanje posmatranog programa ispravno.

Alati za automatsko ispitivanje mogu biti skupi (postoje i besplatni (*eng. freeware*) alati), i obično se koriste u kombinaciji sa ručnim ispitivanjem. Automatsko ispitivanje je isplativo na duži rok, naročito kada se koristi u regresionom ispitivanju.

Jedan od načina za generisanje automatskih ispitnih slučajeva je ispitivanje zasnovano na modelima (*eng. model-based*), ispitivanje u kojima se model sistema koristi za generisanje ispitnih slučajeva.

Kod automatizacije ispitivanje izdvajaju se dva ključna pitanja:

- Šta automatizovati?
- Kada automatizovati?

Ispitni (ili razvojni) tim donosi odluku šta i kada automatizovati. Vrlo bitno je izvršiti izbor odgovarajuće karekteristike proizvoda za automatizaciju, od koje velikoj meri zavisi uspeh automatizacije. Automatizaciju nestabilne funkcije ili funkcije koje prolaze kroz promene treba izbegavati.[7]

2.4.1 Kodom upravljivo (*Code – driven*) ispitivanje

Rastući trend u razvoju softvera je korišćenje okvira (*eng. frameworks*) za ispitivanje kao što su xUnit (na primer, JUnit i NUnit) koji omogućavaju izvršenje jediničnih (*eng. unit*) ispitnih slučajeva. Na osnovu ovih ispitnih slučajeva se utvrđuje da li se različiti delovi koda ponašaju kao što je očekivano pod raznim okolnostima. Ispitnim slučajevima se opisuju ispiti koji moraju da se pokrenu nad programom da provere da li program radi kao što je čekivano.

Kod *code – driven* automatizacije ključna karakteristika je agilni razvoj softvera, koji je poznat kao razvoj vođen ispitivanjem (*eng. test - driven*). Karakteristika agilnog razvoja softvera je da se jedinični ispitni slučajevi koji definišu funkcionalnost pišu pre nego što je napisan kod. Tek kada svi ispitni slučajevi prođu, kod se smatra potpunim. Zagovornici tvrde da je softver proizveden na ovaj način pouzdaniji i jeftiniji nego onaj koji je ručno ispitivan. Takođe se smatra pouzdanijim jer je pokrivenost koda bolja, iz razloga što se ispiti stalno pokreću u toku razvoja. Razvojni programeri lako i brzo otkrivaju nedostatke i odmah ih otklanjaju, u ovoj fazi je najjeftinija popravka nedostataka. [8]

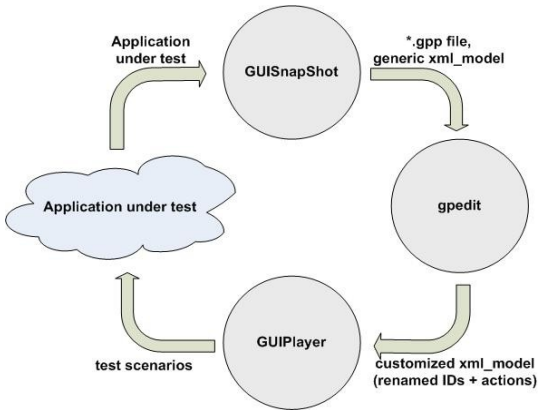
2.4.2 Grafička korisnička sprega (GUI) ispitivanje

Mnogi alati za automatizaciju ispitivanje obezbeđuju snimanje i reprodukciju korisničkih akcija, upoređujući dobijene sa očekivanim rezultatima. Prednost ovog pristupa je to što je potrebno vrlo malo ili nimalo pisanja programskog koda. Ovaj pristup se može primeniti na bilo koju aplikaciju koja ima grafičku korisničku spregu. Međutim, oslanjanje na te funkcije predstavlja veliki problem za pouzdanost i održavanje. Pomeranje dugmeta ili menjanje izgleda

forme mogu da zahtevaju ponovno snimanje ispitnih slučajeva, što zahteva dosta vremena i novca.[9]

3. Koncept rešenja

Rešenje nosi naziv GUIPlayer. GUIPlayer je paket od tri alata koji se koriste za automatizaciju ispitivanja grafičke korisničke sprege. Aplikacija koristi Windows upravljačke ručke (*eng. handle*) za manipulisanje sa objektima GUI-a, tako da ju je moguće pokrenuti samo na Windows XP operativnom sistemu.



Slika 3-1 Koncept rešenja – tok automatizacije

Tok automatizacije može se videti na slici 3-1. Prvi korak je da se svi objekti grafičke korisničke sprege definišu u XML datoteci[10]. Za definisanje objekata razvijen je GUISnapShot alat koji kao rezultat izvršavanja daje XML datoteku sa generičkim imenima objekata. Kako je za dalje stvaranje ispitnih slučajeva pogodnije je da objekat ima intuitivno ime umesto generičkog, za izmenu imena objekata i dodavanje akcija razvijen je Gpedit alat. Izlaz je XML datoteka sa smislenijim imenima objekata i dodatim akcijama. Na kraju se pokreće GUIPlayer aplikacija koja izvršava ispitne scenarije koji se sastoje od skupa akcija implementiranih u Lua skript programskom jeziku.

Lua programski jezik [11] je izabran zato što je jednostavan, brz i prilagodljiv programski jezik. Lua ima jednostavnu i dobro dokumentovanu spregu za programiranje aplikacija (eng. API - Application programming interface) koja omogućava jaku integraciju sa kodom pisanim u

drugim programskim jezicima. Relativno je lako proširiti Lua sa bibliotekama pisanim u drugim programskim jezicima. Takođe je lako da se proširi program pisan u drugim programskim jezicima sa bibliotekama pisanim u Lua programskom jeziku [12].

3.1 GUISnapShot aplikacija

Konzolna aplikacija koja definiše sve objekte koje korisnik odabere u aplikaciji koja se testira (eng. AUT – Application Under Test) kao i njihove pretke (eng. parent) i potomke (eng. child). Pod definisanjem objekata se podrazumeva da se sve informacije o odabranom objektu smestaju u XML datoteku.

Prilikom pokretanja aplikacije otvara se XML datoteka u koju će se upisivati svi definisani objekti. Nakon toga se pokreće aplikacija koja se ispituje (AUT) pomoću parametara prosleđenih prilikom pokretanja aplikacije.

Kompletan sadržaj unutar <gui_model> oznake (*eng. tag*) smatra se delom GUIPlayer-a. Sastoji se od najmanje jedne aplikacije i najmanje jedne globalne funkcije. Svaka aplikacija se sastoji od skupa objekata (<objects> oznaka) i akcija (<actions> oznaka).

Kao izlaz iz GUISnapShot aplikacije dobija se generisana XML datoteka sa generičkim imenima objekata.

3.2 Gpedit aplikacija

Pošto objekti definisani GUISnapShot aplikacijom dobijaju generička imena razvijena je i pomoćna aplikacija koja pomaže korisniku da izmeni jednoznačna imena definisana sa GUISnapShot alatom u imena koji će korisniku davati dovoljno informacija o kom je objektu reč. Ta aplikacija je nazvana Gpedit.

Ovo je GUI aplikacija koja se koristi za prilagođavanje XML GUI modela generisanog GUISnapShot alatom. Uloga ovog alata je da olakša izmene definicije GUI objekta kao što je ObjectID sa smislenijim ObjectID imenom, jer za dalje stvaranje ispitnih slučajeva korisnije je da objekat ima intuitvno ime nego generičko.

Takođe, Gpedit može da se koristi za dodavanje različitih akcija umesto definisanja akcija u XML datoteci pomoću programa za uređivanje teksta.

3.3 GUIPlayer aplikacija

GUIPlayer je aplikacija koja izvršava ispitne slučajeve. Zasnovana je na Lua skriptu što znači da GUIPlayer zahteva prethodno instaliran Lua programski jezik. Takođe, obzirom da ova

aplikacija koristi Windows upravljačke ručke za interakciju sa specifičnim GUI objektima, GUIPlayer je moguće pokrenuti samo na Microsoft Windows operativnim sistemima.

Osnovna datoteka koja se prosleđuje GUIPlayer-u kao parameter je *"gui player project"* (*.gpp) koji sadrži sve definisane *.xml datoteke.

4. Implementacija

U ovom poglavlju izneti su detalji implementacije datog rešenja, odnosno svih alata koji su uključeni u automatizaciju ispitivanja.

4.1 Detalji implementacije

Implementacija paketa je zasnovana na tri alata koji ravnopravno učestvuju u konfiguraciji sistema. Do krajnjeg cilja, a to je potpuna automatizacija ispitnih slučajeva, potrebno je ispratiti određeni redosled pokretanja razvijenih alata na način objašnjen u 3. poglavlju. Prvobitno su razvijeni alat za definisanje svih objekata ispitivane aplikacije (GUISnapshot alat) i alat za izvršavanje ispitinh slučajeva (GUIPlayer alat), a preimenovanje svih definisanih objekata sa generičkim imenima je vršeno ručno u tekst editoru. Kako je bilo teško pronalaziti objekte sa generički dodeljenim imenima, a samim tim i pisati ispitne slučajeve uočena je potreba za razvojem pomoćne aplikacije koja bi omogućila lakše preimenovanje imena definisanih objekata. U tu svrhu razvijena je pomoćna aplikacija Gpedit koja omogućava dodeljivanje intuitivnijih imena objektima na lakši način.

Ovom implementacijom rešenje nije konačno, jer je cilj da se pomoćne aplikacije GUISnapshot i Gpedit spoje u jedinstvenu aplikaciju i da se omogući automatsko snimanje ispitnih slučajeva.

4.2 GUISnapshot

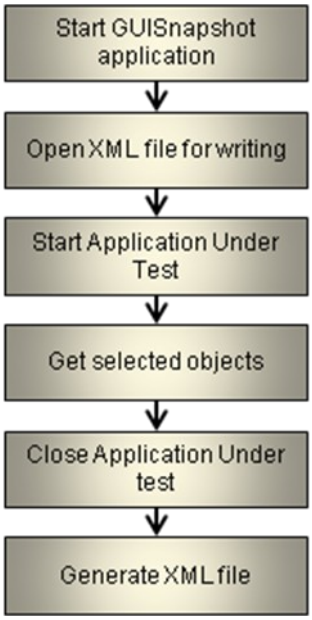
GUISnapshot je konzolna aplikacija koja definiše sve objekte koje korisnik odabere u aplikaciji koja se testira (*eng. AUT – Application Under Test*) kao i njihove pretke (*eng. parent*) i potomke (*eng. child*). Pod definisanjem objekata se podrazumeva da se sve informacije o odabranom objektu smestaju u XML datoteku.

Aplikacija je pisana u C++ programskom jeziku. Pokretanje aplikacije se vrši iz konzole na sledeći način:

GUISnapshot.exe –wWorking_folder –oOutput.xml -- Application_path parameters

- Working_folder – radni direktorijum za navedenu aplikaciju
- Output.xml – ime datoteke u kojoj će biti snimljeni svi definisani objekti
- Application_path – putanja do izvršne datoteke aplikacije koja se ispituje
- Parameters – karakteristični parametri potrebni za pokretanje aplikacije koja se ispituje

Proces definisanja objekata može se videti na slici 4-1.



Slika 4-1 Proces definisanja objekata ispitivane aplikacije

Prilikom pokretanja aplikacije otvara se XML datoteka u koju će se upisivati svi definisani objekti. Nakon toga se pokreće aplikacija koja se ispituje (AUT) pomoću parametara prosleđenih prilikom pokretanja aplikacije. Kada se pokrene AUT svaki selektovani objekat ili novootvoreni dijalog će biti definisan i smešten u privremenu listu. Na primer, za svako dugme (*eng. button*), polje za unos (*eng. edit box*) ili novootvoreni dijalog, upravljačke ručke će biti uzete i smeštene u privremenu listu sa sledećim podacima:

```
<gui_object id="paw::frame_2" class="paw::frame">
<windowclass value="paw::frame"/>
<windowname value="DSP Composer - CS4953X [untitled0]" use="1" />
<style value="0x16cf0000" use="1" />
<exstyle value="0x00000100" use="1" />
<height value="742" use="0" />
<width value="1440" use="0" />
</gui_object>
```

Nakon definisanja svih objekata korisnik zatvara AUT i samim tim se raskida veza između AUT i GUISnapshot aplikacije. Poslednji korak je da se svi objekti iz privremene liste smeste u XML datoteku, koja se nakon toga zatvara i zatvara se GUISnapshot aplikacija.

Struktura generisane XML datoteke data je na slici 4-2.

```
<gui_model>
        <application>
                <objects>
                        <gui_object>
                        </gui_object>
                        <graphic_object>
                        </graphic_object>
                </objects>
                <actions>
                        <function>
                        </function>
                </actions>
        </application>
        <global_functions>
        </global_functions>
</gui_model>
```

Slika 4-2 Struktura generisane XML datoteke

Kompletan sadržaj unutar <gui_model> oznake smatra se delom GUIPlayer-a. Sastoji se od najmanje jedne aplikacije i najmanje jedne globalne funkcije. Svaka aplikacija se sastoji od skupa objekata (<objects> oznaka) i akcija (<actions> oznaka).

Pod akcijama se podrazumevaju proste funkcije koje se često ponavljaju i pozivaju se u ispitnim slučajevima. Na primer, otvaranje dijaloga, zatvaranje dijaloga, prevlačenje objekta ...

Pod globalnim funkcijama se podrazumevaju ispitni slučajevi i u njima se pozivaju osnovne funkcije (akcije).

GUISnapshot ne generise akcije ni globalne funkcije, nego ih korisnik piše ručno.

Pošto objekti definisani ovom aplikacijom dobijaju generička imena razvijena je i pomoćna aplikacija koja pomaže korisniku da izmeni jednoznačna imena definisana sa GUISnapShot alatom u imena koji će korisniku davati dovoljno informacija o kom je objektu reč. Ta aplikacija je nazvana Gpedit.

4.3 Gpedit

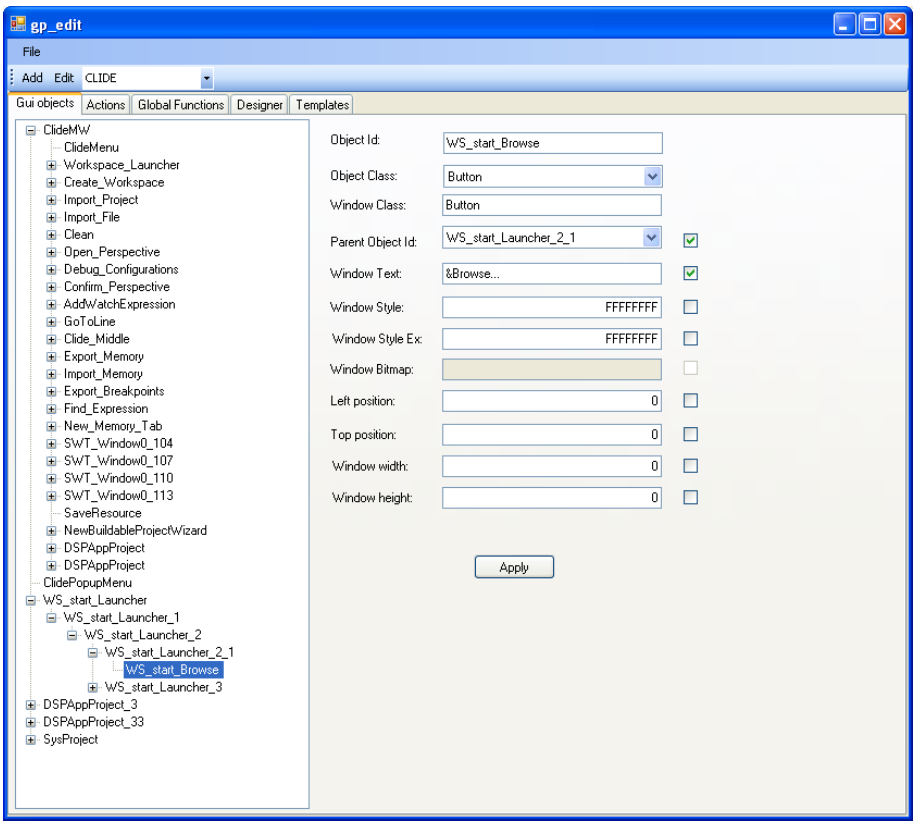
Gpedit aplikacija je razvijena naknadno, nakon što je uočena potreba za lakšim preimenovanjem imena objekata. Ova GUI aplikacija je realizovana u C# programskom jeziku.

Ulazni podatak koji se prosleđuje aplikaciji je gui player project (.gpp) datoteka koja se kreira ručno i sadrži sve definisane XML datoteke. Izgled gpp datoteke je dat na slici 4-3.

```
<?xml version="1.0" encoding="ASCII" standalone="no"?>
<gui_player_project>
        <include_file value="my_project.xml" sourcetype="actions"/>
        <include_file value="my_project_constants.xml" sourcetype="globals"/>
        <include_file value="my_project_tests.xml"/>
        <include_dir value="."/>
        <define name="load" value="value"/>
</gui_player_project>
```

Slika 4-3 Primer gui player project datoteke

Izgled aplikacije je prikazan na slici ispod:



Slika 4-4 Izgled Gpedit aplikacije – GUI objects kartica

Kao što je prikazano na slici, u aplikaciji postoji više kartica koje će biti objašnjene u daljem tekstu.

GUI objects kartica – svi objekti definisani GUISnapshot aplikacijom nalaze se sa leve strane u strukturi stabla. Odabirom željenog objekta iz liste, sa desne strane se ispisuju njegovi parametri koji se mogu menjati. Polja za potvrdu pored određenih obeležja označavaju da ta obeležja mogu i ne moraju da se koriste u definiciji objekta.

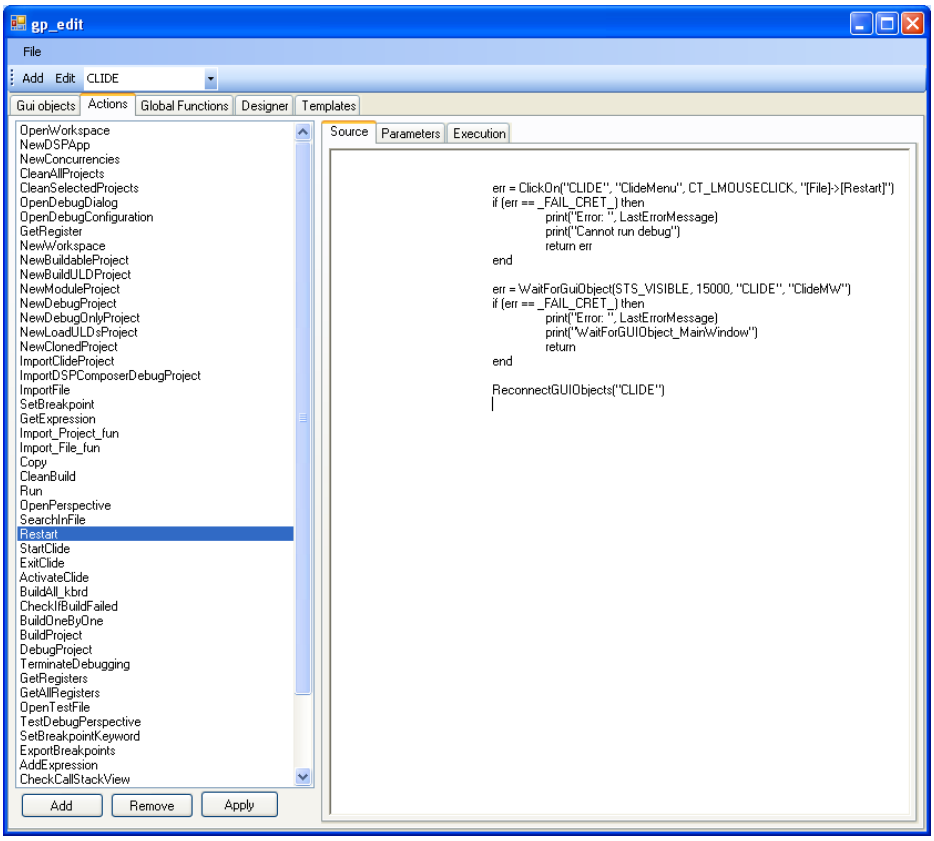
Actions kartica – sa leve strane su u listi navedene sve akcije, kada se odabere akcija koja se želi izmeniti sa desne strane se ispisuje kod koji se menja. Po unetim izmenama, potrebno je kliknuti na *Apply* dugme kako bi izmene bile

snimljene. Takođe moguće je pokrenuti i izvršavanje neke akcije iz Gpedit aplikacije.

Global functions kartica – na isti način kao i kod akcija (*Actions* kartica) menjaju se ispitni slučajevi. Ispitne slučajeve nije moguće izvršiti iz Gpedit aplikacije nego ih je moguće samo menjati.

Designer kartica – ovaj deo aplikacije je zamišljen da vizuelno pomogne korisniku prilikom preimenovanja generički definisanih objekata. Naime svaki selektovani objekat u listi definisanih objekata (*GUI objects* kartica) se na osnovu svojih obeležja grafički predstavlja kako bi se lakše zaključilo o kom objektu je reč.

Templates kartica – uključuje sve šablone koji se mogu iskoristiti prilikom pisanja neke akcije ili ispitnog slučaja.



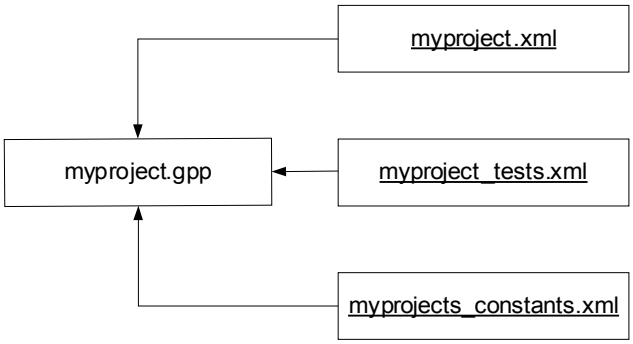
Slika 4-5 Izgled Gpedit aplikacije – Actions kartica

4.4 GUIPlayer

Ovo je konzolna aplikacija realizovana u C++ programskom jeziku. Zasnovana je na Lua skriptu što znači da GUIPlayer zahteva prethodno instaliran Lua programski jezik. Takođe, obzirom da ova aplikacija koristi Windows upravljačke ručke za interakciju sa specifičnim GUI objektima, GUIPlayer je moguće pokrenuti samo na Microsoft Windows operativnim sistemima.

Lua programski jezik je izabran zato što je jednostavan, brz i prilagodljiv programski jezik. Lua ima jednostavnu i dobro dokumentovanu spregu za programiranje aplikacija (*eng. API - Application programming interface*) koja omogućava jaku integraciju sa kodom pisanim u drugim programskim jezicima. Relativno je lako proširiti Lua sa bibliotekama pisanim u drugim programskim jezicima. Takođe je lako da se proširi program pisan u drugim programskim jezicima sa bibliotekama pisanim u Lua programskom jeziku.

Osnovna datoteka koja se prosleđuje GUIPlayer-u kao parameter je *"gui player project"* (*.gpp) koji sadrži sve definisane *.xml datoteke. Organizaciona struktura svih dokumenata uključenih u projekat data je na slici 4-6:



Slika 4-6 Organizaciona struktura svih dokumenata uključenih u GUIPlayer projekat

U *gui player project* datoteci (myproject.gpp) su uključene sve XML datoteke koje treba da budu uključene u projekat.

U *myproject.xml* datoteci su definisane aplikacije, objekti i osnovne funkcije (proste funkcije koje se često ponavljaju i pozivaju se u ispitnim slučajevima).

U *myproject_test.xml* datoteci su definisani ispitni slučajevi. U ispitnim slučajevima se pozivaju osnovne funkcije definisane u myproject.xml datoteci.

Datoteka *myproject_constants.xml* sadrži konstante koje se prosleđuju funkcijama kao parametri.

Primer pokretanja GUIPlayer aplikacije:

echo test_function | GUIPlayer.exe [options] myproject.gpp

[options] - dodatne opcije omogućavaju korisniku da personalizuje aplikaciju prilikom izvršavanja automatskih testova. Pokrivene su sledeće opcije:

- Brzina kursora. Vreme u milisekundama potrebno da kursor pređe sa jedne na drugu stranu ekrana.
- Vreme pritiska tastera u milisekundama. Vreme potrebno da se pritisne i pusti taster.
- Kašnjenje u milisekundama. Vreme koje prođe između pritiska dva tastera na tastaturi.

- Nivo upisivanja log informacija u datoteku:

0 – iskljuceno (podrazumevano)

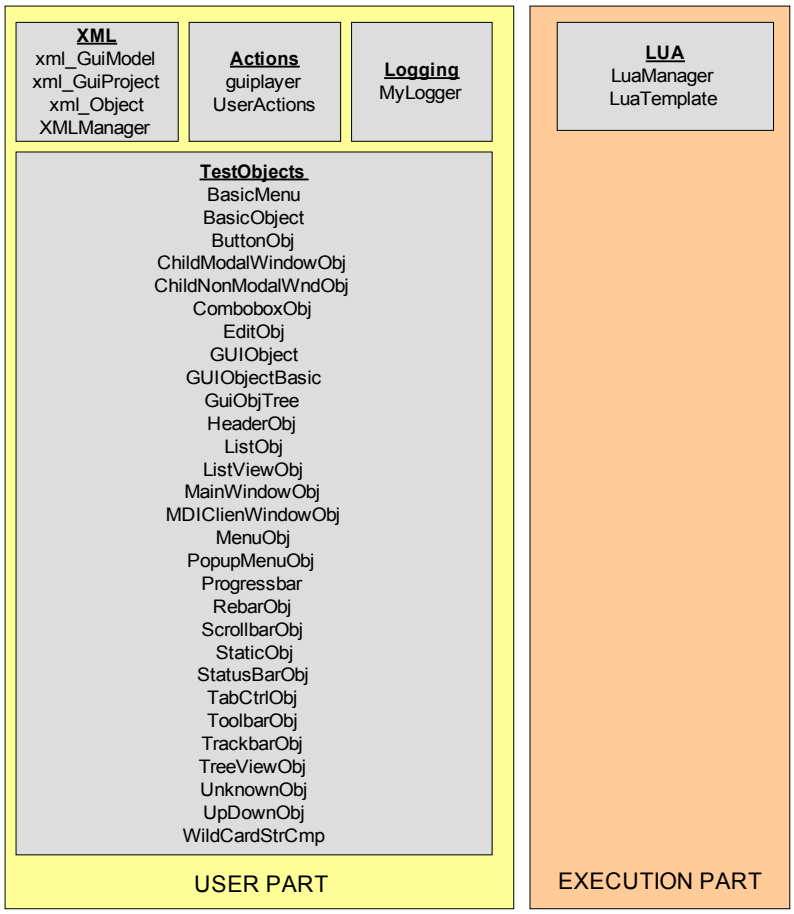
1 - Window konekcija

Generalno GUIPlayer aplikaciju možemo podeliti na dva dela korisnički i izvršni deo. Korisnički deo se sastoji sledećih klasa:

- klase za raščlanjivanje XML datoteka
- klase za definisanje korisničkih akcija
- klase za evidentiranje informacija

Izvršni deo se sastoji od sledećih klasa:

- klase za izvršavanje akcija



Slika 4-7 Podela GUIPlayer aplikacije na korisnički i izvršni deo

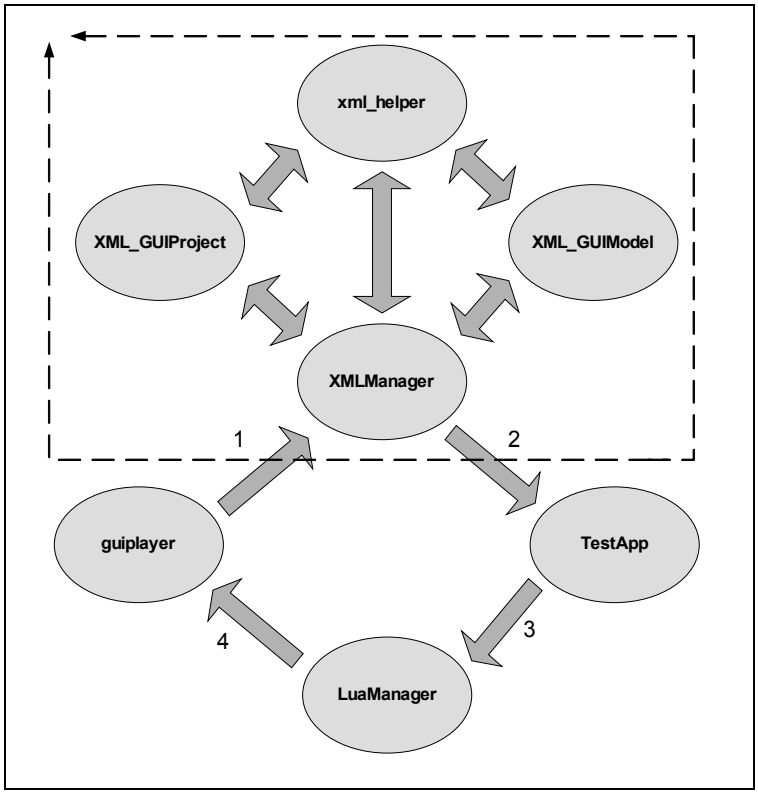
Na osnovu ove podele može se zaključiti da je izvršni deo zasnovan na Lua script programskom jeziku. Drugim rečima, za izvršavanje ispitnih slučajeva koriste se gotove Lua funkcije koje su prilagođene C jeziku.

Navedene su neke od funkcija:

- ClickOn
- TypeText
- PressKeysComb

- MouseMoveToXY
- MouseMove
- ActivateWindow
- ReconnectGUIObjects
- WaitForGuiObject
- WaitForProcess
- MouseClick
- MouseDown
- MouseUp

Na slici 4-8 prikazan je blok dijagram redosleda akcija prilikom izvršavanja jednog ispitnog slučaja. Na slici su navedene klase koje se pozivaju. Ovo je znatno pojednostavljen prikaz kako bi opis implementacije bio razumljiviji.



Slika 4-8 Izvršavanje jednog ispitnog slučaja

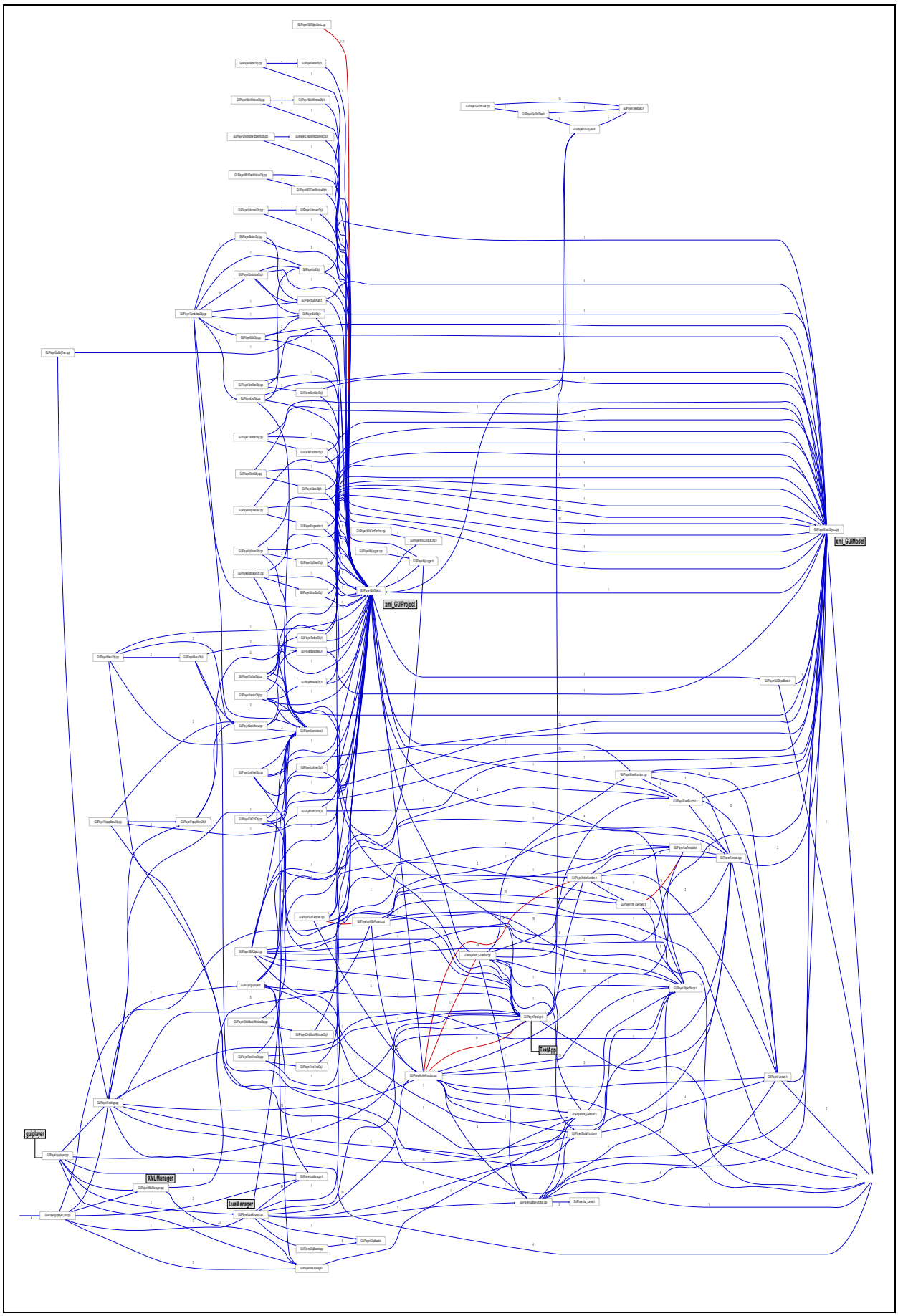
Radi lakšeg razumevanja funkcionalnosti GUIPlayer aplikacije biće objašnjena pojedinačna funkcionalnost navedenih klasa sa slike 4-8 kao i tok izvršavanja jedog ispitnog slučaja.

Funkcionalnost klasa:

- **guiplayer** – ovo je osnovna klasa cele aplikacije. U njoj se vrši raščlanjivanje ulaznih parametara, kreiranje niti za čitanje sa standardnog ulaza, otvaranje datoteke za upis povratnih informacija i pozivanje klasa za raščlanjivanje XML datoteka. Naravno na kraju se vrši oslobađanje zauzete memorije, kao i zatvaranje niti.

- **XMLManager** – upravlja procesom raščlanjivanja XML datoteka uključenih u gpp projekat. Sve aplikacije, objekti, akcije i ispitni slučajevi se smeštaju u strukturu na tačno definisan način, na osnovu svojih obeležja.
- **xml_helper** – pomoćna klasa koja se koristi za čitanje XML datoteka i smeštanje sadržaja datoteka u niz koji se koristi za raščlanjivanje.
- **xml_GUIModel –** vrši raščlanjivanje XML dokumenta i ulančavanje akcija i ispitnih slučajeva u listu.
- **xml_GUIProject –** ulančava objekte u strukturu na osnovu prosleđenih informacija o pretku i potomku određenog objekta.
- **TestApp –** Ovo je jedna od najvažnijih klasa GUIPlayer aplikacije. U ovoj klasi su definisane sve interakcije između aplikacije koja se ispituje (AUT) i GUIPlayer-a. Pokreće nit za povezivanje sa ispitivanom aplikacijom, pokreće aplikaciju, prekida aplikaciju, zatvara aplikaciju, itd.
- **LuaManager –** u ovoj klasi je urađeno prilagođavanje postojećih Lua funkcija C programskom jeziku. Pozivajući te funkcije izvršava se specificirani ispitni slučaj.

Ovo su samo neke osnovne klase GUIPlayer aplikacije, na osnovu kojih se može predstaviti funkcionalnost cele aplikacije. Da bi problem bio slikovito prezentovan i da bi se shvatila složenost celog sistema, dovoljno je pogledati sliku 4-9 gde je dat dijagram izvršenih akcija prilikom izvršenja ispitnih slučajeva.



Slika 4-9 Dijagram akcija izvršavanja jednog ispitnog slučaja

4.4.1 Izvršavanje jednog ispitnog slučaja

U ovom odeljku biće pojašnjen redosled izvršavanja potrebnih akcija prilikom izvršavanja jednog ispitnog slučaja. Pozivamo sledeći ispitni slučaj:

echo „OpenProject(„project“)“ | c:\GUIPlayer\GUIPlayer.exe -console -single c:\GUIPlayer\myproject.gpp

Prilikom pokretanja GUIPlayer aplikacije prvo se vrši raščlanjivanje i analiza parametara (- *console*, da čita podatke sa standardnog ulaza, - *single*, da se izvršavanje vrši samo jednom...). Proverava da li već postoji pokrenuta GUIPlayer aplikacija (ne mogu dve GUIPlayer aplikacije biti pokrenute u isto vreme) i kreira datoteku za upis korisničkih informacija (ukoliko ime datoteke nije specificirano prilikom poziva aplikacije, dodeljuje mu se ime *guiplayer.log*).

Nakon otvaranja datoteke za upis korisničkih informacija vrši se raščlanjivanje myproject.gpp datoteke a zatim i XML datoteka koji su njen sastavni deo. Svi definisani objekti se upisuju u strukturu *GuiStruct* na osnovu informacija o pretku i potomku (izgled strukture moze se videti na slici 4-4). Sve definisane akcije se upisuju u strukturu *ActionStruct* a svi ispitni slučajevi u strukturu *GlobalCodeStruct.* Nakon iščitavanja svih navedenih XML datoteka iz gpp datoteke kao i njihovog smeštanja u strukture pokreće se nit za povezivanje GUIPlayer aplikacije sa AUT i izvršava se ispitni scenario specificiran prilikom pozivanja GUIPlayer-a („OpenProject(„project“)).

```
function OpenProject(project_name)
                    err = ClickOn("APPLICATION", "APPMenu", CT_LMOUSECLICK, "[File]->[Open...\tCtrl+O]")
                    if (err == _FAIL_CRET_) then
                              print("Error: ", LastErrorMessage)
                              print("Cannot ClickOn menu")
                              return err
                    end
                    err = WaitForGuiObject(STS_VISIBLE, 10000, "APPLICATION", "OpenButton")
                    if (err == _FAIL_CRET_) then
                              print("Error: ", LastErrorMessage)
                              print("There is no button...")
                              return err
                    end
                    TypeText(project_name)
                    ReconnectGUIObjects("APPLICATION")
                    err = ClickOn("APPLICATION", "OpenButton", CT_LMOUSECLICK)
                    if (err == _FAIL_CRET_) then
                              print("Error: ", LastErrorMessage)
                              print("Cannot ClickOn Open button")
                              return err
                    end
end
```

Slika 4-10 Ispitni scenario OpenProject(„project“)

Svi ispitni scenariji su pisani u Lua skript programskom jeziku. Jedan primer je dat na slici 4-10. Ovaj primer otvara postojeći projekat. Kao što se vidi na primeru, posle svake akcije vrši se provera da li je rezultat očekivan ili nije.

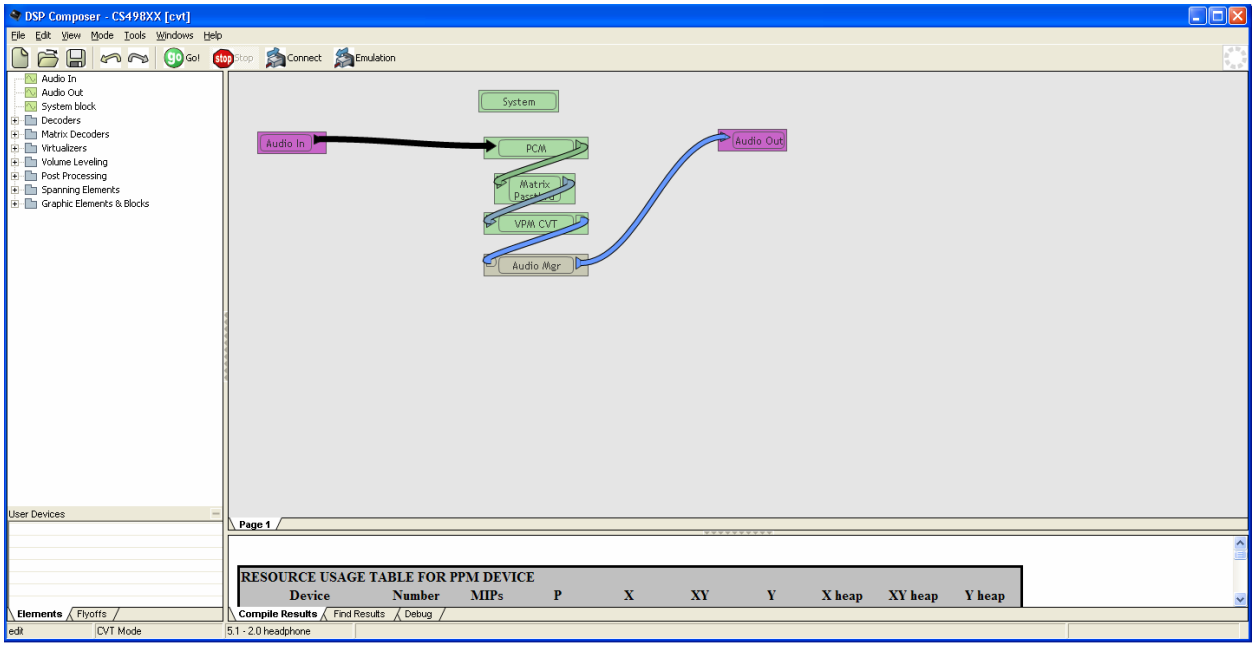
Opis ispitnog slučaja na slici 4-10:

1. Klikni na meni File -> Open
2. Sačekaj da se pojavi novi dijalog (OpenButton objekat se nalazi u novom dijalogu)
3. Upiši ime projekta koji se otvara (project)
4. Klikni na OpenButton i otvori specificirani projekat.

Nakon izvršenja navedenog ispitnog slučaja zatvara se nit za povezivanje GUIPlayer-a i AUT i oslobađa zauzeta memorija

5. Ispitivanje i verifikacija

Nakon razvoja alata, pristupilo se ispitivanju rešenja. Ispitivanje je vršeno na grafičkom alatu za razvoj programske podrške za audio ciljne platforme kompanije Cirrus Logic. Alat se zove DSP Composer [13] i namenjen je Crystal DSP familiji procesora firme Cirrus Logic. DSP Composer je grafički alat koji podržava „prevuci i spusti“ (*eng. drag-and-drop*) dizajn tokova audio signala. Korisnici ovog alata mogu kreirati upravljački program (*eng. firmware*) za audio obradu za bilo koji procesor iz DSP familije bez pisanja prilagođenog DSP koda kombinovanjem elemenata iz velikog izbora postojećih primitivnih elemenata za audio obradu. DSP Composer takođe obezbeđuje run-time kontrolu firmware parametara, kako za obradu blokova definisanih od strane korisnika tako i za obradu blokova isporučenih od strane Cirrus Logic-a. Izgled DSP Composer-a dat je na slici 5-1.



Slika 5-1 DSP Composer

Ispitivanje je vršeno na nekoliko načina koji će biti objašnjeni:

- Vizuelno praćenje akcija sa očekivanim rezultatom
- Vizuelno praćenje akcija na različitim rezolucijama ekrana
- Bit-identično ispitivanje

5.1 Vizuelno praćenje akcija sa očekivanim rezultatom

Pod vizuelnim praćenjem akcija sa očekivanim rezultatom podrazumeva se vizuelna verifikacija da je izabran objekat koji je definisan ispitnim slučajem. Odabrano je 500 ispitnih slučajeva koji pokrivaju sve objekte definisane aplikacije. Svaki ispitni slučaj je napisan na osnovu unapred definisanog ispitnog scenarija koji se izvršavao ručno, tako da su rezultati, odnosno redosled otvaranja dijaloga, bili unapred poznati. Rezolucija ekrana je bila 1920x1080 (širina x visina). Ispitnim slučajevima je obuhvaćena osnovna funkcionalnost aplikacije (startovanje aplikacije, zatvaranje aplikacije, otvaranje projekta, snimanje izmena u projektu...) a akcenat je stavljen na pokrivenost svih definisanih objekata. Urađeno je vizuelno praćenje izvršavanja zadatih akcija za sve ispitne slučajeve kao i upređivanje vremena izvršavanja sa vremenom potrebnim da se ručno izvrše svi ispitni slučajevi. Svaki ispitni slučaj je uspešno izvršen a vreme izvršavanja se smanjilo 3 puta, što se moze videti u Tabeli 5-1. Vreme izvršavanja ispitnih slučajeva je u danima, pri čemu je reč o radnim danima (radni dan je 8 časova).

Način ispitivanja ***Broj ispitinih slučajeva***	***ručno***	***automatsko***
500	**3 dana**	**1 dan**

Tabela 5-1 Zavisnost vremena ispitivanja od načina izvršavanja ispitnih slučajeva

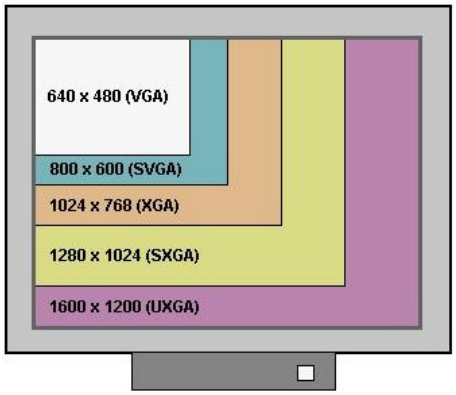
U predhodnoj tabeli prilikom upoređivanja vremena izvršavanja za automatsko izvršavanje ispitnih slučajeva nije uzeto u obzir potrebno vreme za konfiguraciju sistema. Da bi se dobila realna slika o uštedi vremena, u Tabeli 5-2 je dodato vreme koje je potrebno da se definišu svi objekti i napišu ispitni slučajevi. Potrebno vreme za pisanje ispitnih slučajeva nije uzeto u obzir jer su ručni testovi bili već napisani. Ono što je bio cilj da se prikaže je da i u slučaju kad već postoje napisani ručni testovi, pisanje novih automatskih testova je isplativo, gledajući sa strane uštede vremena, već nakon tri izvršavanja skupa od 500 testova, uključujući i konfiguraciju sistema.

Način ispitivanja *Broj ispitinih slučajeva*	*ručno*	*automatsko*
500	**3 dana**	**8 dana**

Tabela 5-2 Zavisnost vremena ispitivanja od načina izvršavanja ispitnih slučajeva sa dodatim vremenom konfiguracije sistema za automasko ispitivanje

Kao što se vidi iz tabele ako se u izvršavanje ispitnih slučajeva uračuna i konfigurisanje sistema, ručno izvršavanje će se pre izvršiti. Prava ušteda vremena se ogleda u ponavljanju ispitnih slučajeva, odnosno sistem se jednom konfiguriše i nakon toga se dobijaju izvršni ispitni slučajevi koji se izvršavaju automatski i višestruko brže. Iz dobijenih rezultata vidi se da već nakon trećeg izvršavanja se dobija ušteda u vremenu.

5.2 Vizuelno praćenje akcija na različitim rezolucijama ekrana



Ispitivanje koje je izvršeno i opisano u prethodnom poglavlju ponovljeno je ali ovaj put je menjana rezolucija ekrana, kako bi bila potvrđena funkcionalnost i robusnost alata za automatizaciju ispitivanja u različitim uslovima. Svi ispitni slučajevi su automatski izvšavani na različitim rezolucijama ekrana.

Korišćene su sledeće rezolucije:

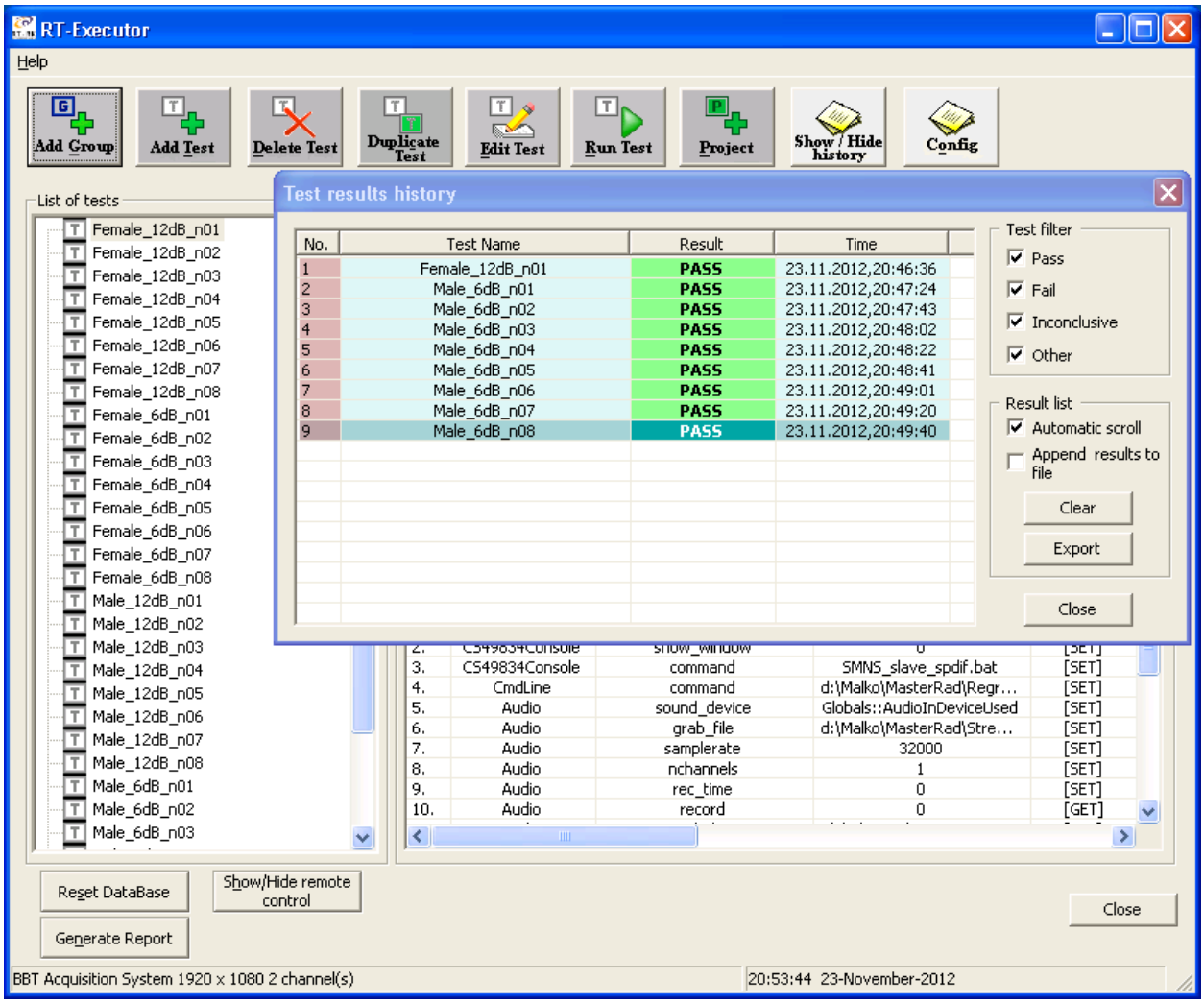
- 800 x 600
- 1024 x 768
- 1280 x 1024
- 1600 x 1200

Naravno, treba napomenuti da za izvršavanje ispitnih slučajeva nije rađeno ponovno definisanje objekata i pisanje istih, nego su iskorišćeni definisani objekti i akcije iz prethodnog testiranja (rezolucija 1920 x 1080).

Za sve izvršene ispitne slučajeve dobijeni su očekivani rezultati. Ovim je potvrđeno da su svi objekti dobro definisani i da izvršavanje ne zavisi od rezolucije ekrana. Naravno ovde treba voditi računa da se u ispitnim slučajevima ne koristi pozicioniranje kursora na ekranu, jer će u tom slučaju ispitni slučaj biti zavisan od veličine ekrana.

5.3 Bit-identična ispitivanja

Iako bit-identična ispitivanja ne nose informaciju o kvalitetu alata, vrlo su korisna jer se ispituje ponašanje alata prilikom izvršavanja velikog broja ispitnih slučajeva. Takođe, prilikom izvršavanja velikog broja ispitnih slučajeva najizraženija je ušteda vremena prilikom automatskog izvršavanja ispitnih slučajeva. Ovakva ispitivanja su dakle jako korisna, jer mogu vrlo lako da ukažu na grešku, ukoliko je bitska razlika van predviđenih okvira. Za bit-identično ispitivanje korišćen je poseban alat, BBT [14] (*eng. Black Box Testing* – ispitivanje metodom crne kutije). Izgled ovog alata prikazan je na slici 5-2.

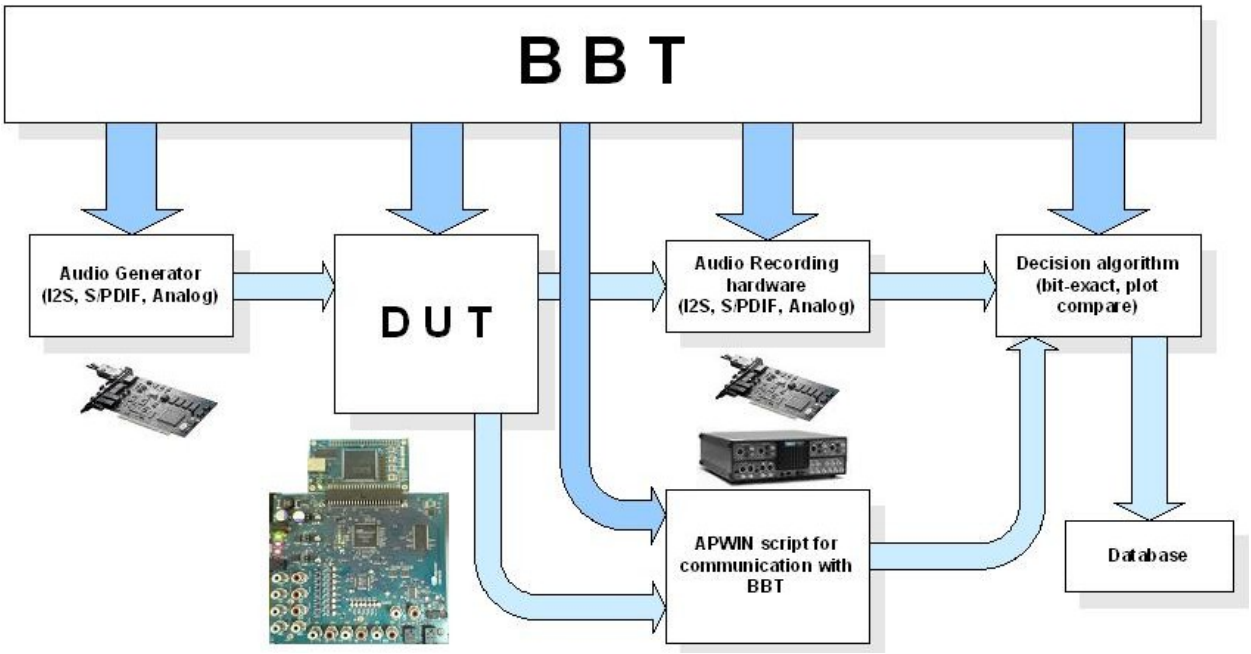


Slika 5-2 Prikaz BBT alata

BBT alat je razvijen na katedri za računarsku tehniku i računarske komunikacije Fakulteta tehničkih nauka u Novom Sadu i predstavlja moćan alat za ispitivanje i verifikaciju. BBT alat je organizovan je tako da se sastoji od jezgra (*eng. core*) aplikacije i zasebnih modula u kojima su implementirane sve potrebne funckionalnosti. To se postiže dodavanjem dinamičke biblioteke (*eng. dynamic link library – dll*) sa definicijom novog modula u instalacioni direktorijum. BBT sistem dinamički učitava sve uređaje smeštene u instalacioni direktorijum prilikom pokretanja

BBT aplikacije. Ovim se dobija na modularnosti, odnosno dodavanje novih funkcionalnosti ne zahteva izmene u samom jezgru aplikacije.

Prednosti BBT alata su brojne, od relativno jednostavne sintakse za pisanje ispitnih slučajeva (a samim tim i razvoj istih je relativno brz), do lakog korišćenja, prenosivosti, mnoštva ugrađenih alata koji pojednostavljuju proces ispitivanja, sve do preglednih izveštaja o rezultatima. Ovaj alat, dakle, omogućava brz razvoj ispitnih slučajeva, a ugrađeni alati omogućavaju izvršavanje širokog spektra operacija. Standardna procedura za ispitivanje izlaza sa ciljne platforme (konfiguracija, snimanje izlaza, vremensko poravnanje i bit-identično poređenje), korišćenjem BBT alata, prilično je jednostavna i jednom razvijeni ispitni slučajevi mogu se ponoviti neograničeni broj puta, kao i na proizvoljnom računaru.



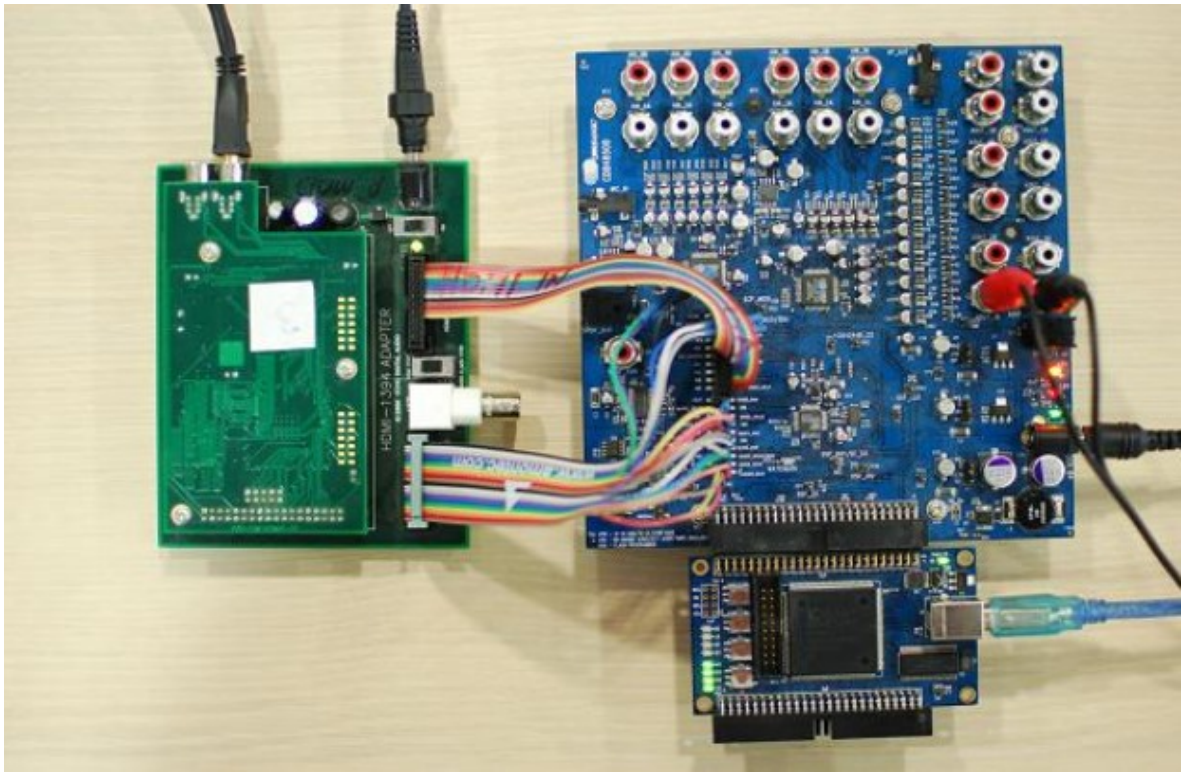
Slika 5-3 Primer ispitnog BBT okruženja

U okviru ovog rada urađeno je ispitivanje korišćenjem BBT alata, na način da je konfiguracija odnosno spuštanje firmware-a u ispitnim slučajevima rađeno korišćenjem GUIPlayer alata (kreiranje projekata u DSPComposer-u). Kao ciljni processor korišćen je CS48560 [15] digitalni signal procesor iz Crystal DSP familije procesora, firme Cirrus Logic, a za reprodukciju i snimanje signala korišćena je zvučna kartica kompanije Echo Audio[16]. Specifikacija ispitnih vektora data je u tabeli 5-3.

	Broj kanala	*Trajanje(sec)*	*Frekvencija uzorkovanja(Hz)*	*Veličina uzorka (bit)*
input_2ch.wav	**2**	**10**	**48000**	**16**
input_4ch.wav	**4**	**10**	**48000**	**16**
input_6ch.wav	**6**	**10**	**48000**	**16**
input_8ch.wav	**8**	**10**	**48000**	**16**

Tabela 5-3 Specifikacija ispitnih vektora

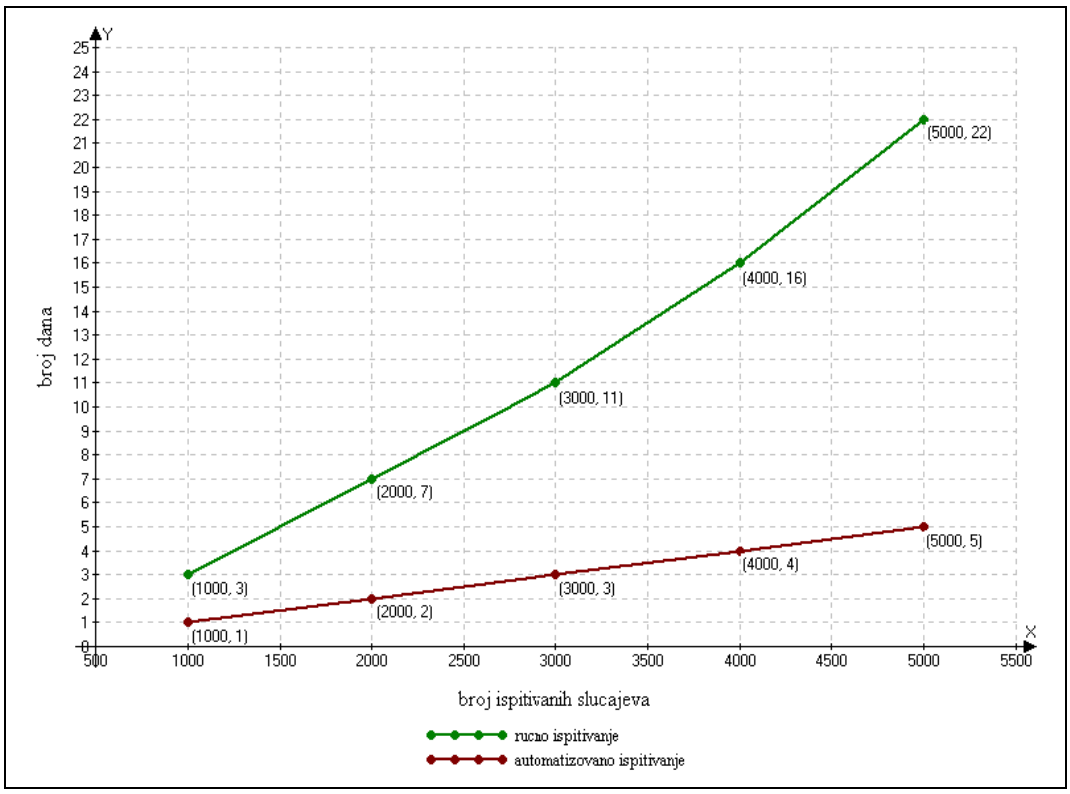
Ispitno okruženje prikazano je na slici 5-4.



Slika 5-4 Ispitno okruženje

Snimljeni izlazi su automatski poređeni sa unapred poznatim referentnim signalima. Potvrda koncepta se ogledala u tome da se potvrdi bit-identičnost snimljenih rezultata i referentnih, odnosno da je GUIPlayer pravilno izgenerisao kod grafičkim alatom i spustio ga na ploču.

Takođe, istovremeno sa izvršavanjem automatskih ispitnih scenarija rađeno je ručno ispitivanje, odnosno jedna osoba je ručno izvršavala sve ispitne slučajeve kako bi bila potvrđena ušteda vremena prilikom izvršavanja velikog broja ispitnih slučajeva. Svi testovi su uspešno izvršeni, a vreme testiranja se značajno smanjilo. Na slici 5-5 data je zavisnost vremena ispitivanja od broja izvršenih ispitnih slučajeva ukoliko se ispitivanje izvršava ručno ili automatski. Broj dana na grafiku odnosi se na broj radnih dana (radni dan je 8 časova).



Slika 5-5 Grafik zavisnosti vremena ispitivanja od broja izvršenih ispitnih slučajeva pri ručnom i automatizovanom testiranju.

Na osnovu datog grafika vidi se da se vreme izvršavanja testova smanjilo za više od 4 puta. Potrebno je napomenuti da je i za automatsko ispitivanje uzet radni dan, odnosno 8 časova. Ovde namerno nije iskorišćena prednost automatskog ispitivanja, a to je kontinualno izvršavanje ispitnih slučajeva 24 časa dnevno, iz razloga da se vidi da čovekova koncentracija i efikasnost opada tokom vremena a samim tim verovatnoća greške se povećava.

Naravno, prostom računicom može se zaključiti da ukoliko bi se automatsko ispitivanje izvršavalo 24 časa, izvršavanje svih 5000 ispitnih slučajeva bi trajalo manje od 2 dana (40 časova) što znači da bi ušteda u vremenu bila 20 dana.

Ova ušteda je višestuko značajna ako se uzme u obzir da se ispitivanje ponavlja nakon izmena u kodu ispitivane aplikacije ili kodu modula koji se koristi kroz ispitivanu aplikaciju kao u ovom slučaju.

6. Zaključak

U ovom radu realizovan je alat za automatizaciju ispitivanja grafičke korisničke sprege na Microsoft Windows XP operativnom sistemu. Alat nosi naziv GUIPlayer a sastoji se od tri nezavisna alata razvijena kako bi se lakše izvršila potpuna automatizacija ispitivanja. Razvijeni su alati:

- **GUISnapShot** - alat koja definiše sve objekte koje korisnik odabere u aplikaciji koja se testira kao i njihove pretke i potomke. Sve informacije o odabranom objektu smesta u XML datoteku.
- **Gpedit** - alat koji pomaže korisniku da izmeni jednoznačna imena definisana sa GUISnapShot alatom u imena koji će korisniku davati dovoljno informacija o kom je objektu reč.
- **GUIPlayer** – alat koji izvršava ispitne slučajeve.

Ovim rešenjem je ostvarena značajna ušteda vremena tokom izvršavanja testova, kao i smanjene aktivne angažovanosti inženjera u ispitivanju. Na ovaj način se smanjuje ljudski faktor, odnosno mogući previdi prilikom izvršavanja velikog broja testova. Verifikacija rešenja urađena je na grafičkom alatu za razvoj programske podrške za audio ciljne platforme kompanije Cirrus Logic na tri načina. Prvi način je bio vizuelna verifikacija da su prilikom izvršavanja ispitnih slučajeva izabrani očekivani objekti i da su sve akcije uspešno izvršene. Kako su svi ispitni slučajevi uspešno izvršeni, potvrđena je osnovna funkcionalnost alata, pravilno definisanje objekata aplikacije koja se ispituje i izvršavanje ispitnih slučajeva. Drugi način verifikacije podrazumevao je ponavljanje ispitnih scenarija iz prvog slučaja ali na različitim rezolucijama ekrana. Uspešnim izvršavanjem svih ispitnih slučajeva potvrđena je funkcionalnost i robusnost alata za automatizaciju ispitivanja u različitim uslovima. Treći način verifikacije je bio izvršavanjem bit-identičnih ispitivanja gde je potvrđena integracija predloženog rešenja u sistem za ispitivanje na principu crne kutije. Istovremeno sa izvršavanjem automatskih ispitnih scenarija rađeno je ručno ispitivanje kako bi bila potvrđena ušteda vremena prilikom izvršavanja velikog broja ispitnih slučajeva. Svi testovi su uspešno izvršeni, a vreme testiranja se smanjilo za više od

4 puta. Nakon višemesečnog ispitivanja potvrđena je stabilnost, robusnost i pouzdanost alata za automatsko ispitivanja grafičke korisničke sprege. Iz ugla korisnika, aplikacija obezbedjuje visok komfor i mogućnost upravljanja objektima grafičkog okruženja ispitivane aplikacije, kao i pouzdanost prilikom izvršavanja velikog broja testova.

Tokom ispitivanja rešenja primećeni su nedostaci čije ispravljanje treba da bude smernica ka daljem razvoju ovog alata. Kao najveći nedostaci nameću se generičko imenovanje objekata prilikom generisanja XML datoteke i nemogućnost automatskog snimanja ispitnih slučajeva, nego se oni ručno pišu. O ovom je već bilo reči u radu, ali ideja je da se dva pomoćna alata GUISnapshot i Gpedit sjedine u alat koji će automatski snimati ispitne slučajeve i definisati objekte sa dovoljno intuitivnim imenima da preimenovanje nije potrebno.

Takođe, kako bi ovo rešenje bilo konkurentno na tržištu neophodno je da se razvije grafička korisnička sprega za GUIPlayer kako bi korisniku bilo omogućeno lakše upravljanje programom.

7. Literatura

[1] Hou Wenjun, *"The Three-Dimensional User Interface, Advances in Human Computer Interaction",* Shane Pinder (Ed.), InTech, 2008, ISBN: 978-953-7619-15-2

[2] Myers G.J., Badgett T., Thomas T.M., Sandler C.: *„The Art of Software Testing",* Second Edition - John Wiley & Sons, Inc. 2005

[3] Daniel Galin, *Software Quality Assurance, From Theory to implementation*, Pearson Education Limited 2004.

[4] Ron Patton, *Software Testing,* Que/Sams, November, 2000, ISBN 9780768657258

[5] Kelly J. Hayhurst, Dan S. Veerhusen, John J. Chilenski, and Leanna K. Rierson, "A Practical Tutorial on Modified Condition/Decision Coverage," published jointly be NASA and FAA in 2001.

[6] Boris Beizer, *Black-Box Testing : Techniques for Functional Testing of Software and Systems*, John Wiley & Sons, Inc. 1995, ISBN 0471120944

[7] Gerald D. Everett, Raymond McLeod, Jr., Software Testing Testing Across the Entire Software Development Life Cycle, by John Wiley & Sons, Inc, 2007.

[8] Mark Fewster & Dorothy Graham (1999). *Software Test Automation*. ACM Press/Addison-Wesley. ISBN 978-0-201-33140-0.

[9] Herb Isenberg, *"Automated Testing"*, June, 2012

[10] Elliotte Rusty Harold, W. Scott Means, XML in a Nutshell, 3rd edition September 2004. O'Reilly. ISBN: 86-7555-294-7

[11] Lua 5.1 Reference Manual, by R. Ierusalimschz, L. H. de Figueiredo, W. Celes, Lua.org, August 2006, ISBN 85-903798-3-3, http://www.lua.org/manual/5.1/, January, 2013

[12] Programming in Lua, by R. Ierusalimschz, Lua.org, March 2006, ISBN 85-903798-2-5

[13] *„DSP Composer Users Manual”*, Cirrus Logic, Inc., February 2012

[14] Dušica Marijan, Vladimir Zlokolica, Nikola Teslić, Vukota Peković, Tarkan Teckan, “Automatic Functional TV Set Failure Detection System”, *IEEE Transactions on Consumer Electronics*, Vol. 56, No. 1, February 2010.

[15] *„CS485xx Famiy Data Sheet”*, Cirrus Logic, Inc., October 2011

[16] *"Owner’s Manual Version 2.2 for Windows",* Echo Digital Audio Corporation, September, 2009