УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА У НОВОМ САДУ

Дејан Сретеновић

Једно решење алата за аутоматизацију процеса испитивања графичке корисничке спреге

МАСТЕР РАД

Нови Сад, Октобар 2016.

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**:		Дејан Сретеновић	
Ментор, **МН**:		Доц. Др Јелена Ковачевић	
Наслов рада, **НР**:		Једно решење алата за аутоматизацију процеса 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**:		2016	
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)		(7/35/14/1/16/0/14)	
Научна област, **НО**:		Електротехника и рачунарство	
Научна дисциплина, **НД**:		Рачунарска техника	
Предметна одредница/Кqучне речи, **ПО**:		Аутоматизација процеса испитивања, графички кориснички интерфејс	
УДК			
Чува се, **ЧУ**:		У библиотеци Факултета техничких наука, Нови Сад	
Важна напомена, **ВН**:			
Извод, **ИЗ**:		У овом раду је реализован графички алат као део система за аутоматизацију процеса испитивања графичке корисничке спреге. У раду је приказан поступак имплементације са детаљима програмског решења и тестирањем.	
Датум прихватања теме, **ДП**:			
Датум одбране, **ДО**:			
Чланови комисије, **КО**:	Председник:	Доц. Др Илија Башићевић	
	Члан:	Доц. Др Иван Мезеи	Потпис ментора
	Члан, ментор:	Доц. др Јелена Ковачевић	

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**:		Dejan Sretenović	
Mentor, **MN**:		Jelena Kovačević, PhD	
Title, **TI**:		One solution of an graphical application for the process of automatisation of GUI testing	
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**:		2016.	
Publisher, **PB**:		Author's reprint	
Publication place, **PP**:		Novi Sad, Dositeja Obradovica sq. 6	
Physical description, **PD**: (chapters/pages/ref./tables/pictures/graphs/appendixes)		(7/35/14/1/16/0/14)	
Scientific field, **SF**:		Electrical Engineering	
Scientific discipline, **SD**:		Computer Engineering, Engineering of Computer Based Systems	
Subject/Key words, **S/KW**:		GUI testing automatisation	
UC			
Holding data, **HD**:		The Library of Faculty of Technical Sciences, Novi Sad, Serbia	
Note, **N**:			
Abstract, **AB**:		This paper contains implementation of a graphical tool which, as part of a tool chain, provides the automatisation of the GUI testing process. The process of implementation with software solutions details and testing are shown in this paper.	
Accepted by the Scientific Board on, **ASB**:			
Defended on, **DE**:			
Defended Board, **DB**:	President:	Ilija Bašićević, PhD	
	Member:	Ivan Mezei, PhD	Menthor's sign
	Member, Mentor:	Jelena Kovačević, PhD	

Zahvalnost

Zahvalnost dugujem mentoru Doc. Dr Jeleni Kovačević i tehničkom mentoru Đorđu Miljkoviću na strpljenju i tehničkoj podršci. Takođe bih se zahvalio svojoj porodici, kolegama i svima onima koji su doprineli izradi ovog rada.

SADRŽAJ

SPISAK SLIKA

SPISAK TABELA

SKRAĆENICE

API - *Application Programming Interface,* Aplikativna programska sprega

DSP - *Digital signal processors,* Digitalni signal procesori

GUI - *Graphical User Interface,* Grafički korisnički interfejs

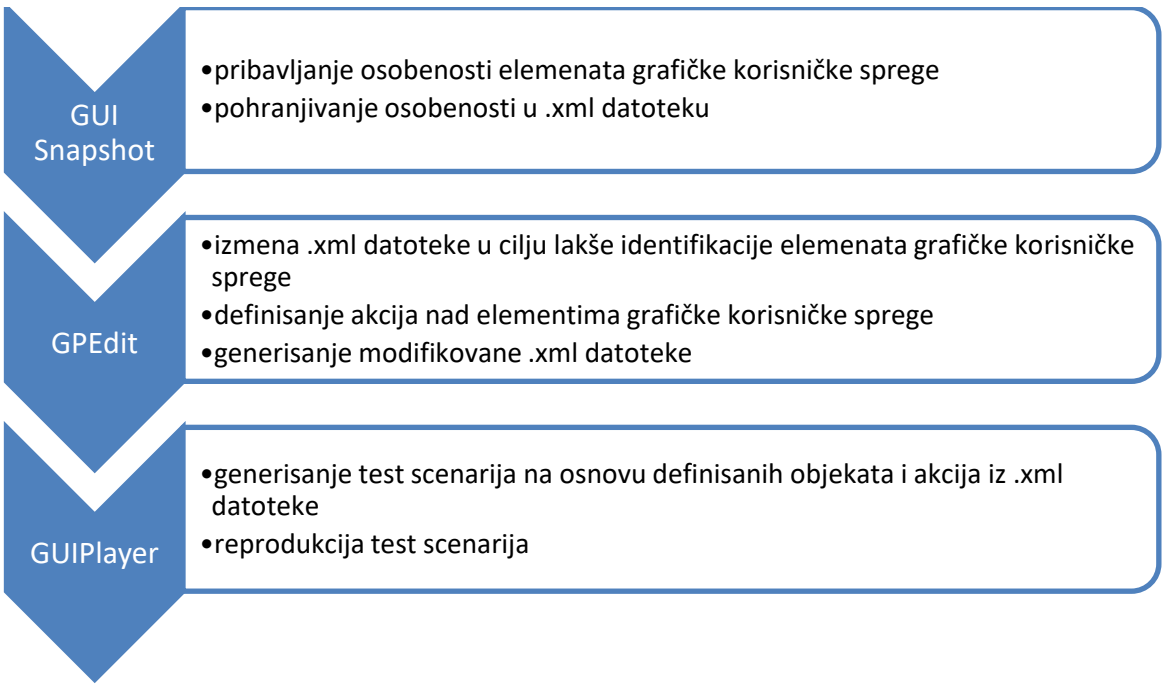
IDE - *Integrated Development Environment*, Integrisano razvojno okruženje

PCR - *Parent childern relations,* Postojanje podelemenata

XML - *Extensible Markup Language,* Proširiv jezik za opisivanje

1. Uvod

U ovom radu objašnjen je koncept i realizacija jednog softverskog rešenja nazvanog *GuiSnapShot,* koji predstavlja alat za automatizaciju procesa ispitivanja grafičke korisničke sprege na *Microsoft Windows* operativnom sistemu. Celinu (eng. *toolchain*) za automatizaciju ispitivanja aplikativnog softvera koji se oslanja na funkcionalnost grafičke korisničke sprege čine tri alata: *GuiSnapShot, GPEdit i GUIPlayer*.



Slika 1-1 Lanac alata koji čine celinu za process automatizacije ispitivanja grafičke korisničke sprege

GuiSnapShot u svojoj izvornoj formi predstavlja konzolnu aplikaciju ograničenog spektra mogućnosti, čiji rezultat izvršavanja nije intuitivan za dalji rad (izlaz GuiSnapShot alata predstavlja ulaz sledećeg alata u lancu, a to je GPEdit) i od korisnika očekuje dodatne napore u pravcu prilagođavanja rezultata. Pred rad su stavljeni zahtevi da finalno rešenje bude u grafičkoj formi, sa intuitivnim interfejsom, proširenog skupa akcija (mogućnost manipulacije (promena imena, brisanje iz strukture) nad pribavljenim osobenostima grafičke korisničke sprege), koji će omogućiti jednoznačnu identifikaciju i definisanje grafičkih elemenata (dugme, polje za izmenu, meni, ...) kao dela celine koju čini aplikacija koju nadgledamo (eng. *monitoring*). Takođe, softversko rešenje treba da omogući i naglašavanje (eng. *highlight*) grafičkih elemenata na ekranu radi lakšeg identifikovanja, kao i finalno pohranjivanje pribavljenih osobenosti elemenata grafičke korisničke sprege u formi XML dokumenta sa predak-potomak naslednim vezama između elemenata strukture. Rešenje koristi spregu *Windows* operativnog sistema sa objektima grafičke korisničke sprege, kojim manipuliše preko takozvanih upravljačkih rukovalaca (eng. *handle*).

Grafička korisnička sprega (eng. *GUI – Graphical User Interface*) predstavlja vid komunikacije korisnika sa računarom posredstvom interakcije sa grafičkim elementima, tekstualnim porukama i obaveštenjima. GUI se bazira na objektima među kojima su prozori (eng. *window*), ikonice (eng. *icon*), meniji (eng. *menu*) i pokazivači (eng. *pointer*) a čiju manipulaciju od strane korisnika omogućava upotreba miša i tastature[1].

Verifikacija i validacija aplikacija kroz proces ispitivanja čini bitan deo toka razvoja softverskih rešenja. Ispitivanje predstavlja osnov u procesu obezbeđivanja kvaliteta. Ispitivanje grafičke korisničke sprege je od vitalnog značaja, zbog toga što se izvršava iz perspektive krajnjeg korisnika datog softverskog rešenja. Grafička korisnička sprega korisnicima otvara širok spektar mogućih akcija u radu sa aplikacijama koje im olakšavaju rad, ali otežava posao razvojnim inženjerima i inženjerima koji vrše proces ispitivanja. Budući da korisnici sa grafičkom korisničkom spregom 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 je da se process ispitivanja automatizuje, što kao krajnji ishod dovodi do smanjenja vremena neophodnog za izvršavanje velikog broja ispitnih slučajeva, smanjenje uticaja ljudskog faktora na pojavu greške prilikom procesa ispitivanja, kao i olakšan proces regresivnog ispitivanja. Rezultati ispitivanja ispravnosti realizovanog rešenja pokazali su da novoformirani prednji deo funkcioniše ispravno.

Rad je sačinjen od sedam poglavlja:

- u drugom poglavlju dat je pregled teorijskih osnova koje stoje iza procesa ispitivanja koje su neophodne radi lakšeg razumevanja koncepta rešenja i same realizacije
- u trećem poglavlju je predstavljen koncept rešenja
- u četvrtom poglavlju je opisana implementacija programskog rešenja
- u petom poglavlju je prikazan proces ispitivanja i verifikacije programskog rešenja
- u šestom poglavlju izneti su zaključci, pregled urađenog, kao i smernice koje se mogu uzeti u cilju daljeg poboljšanja programskog rešenja
- u sedmom poglavlju je prikazana korišćena literatura

2. Teorijske osnove

Ispitivanje je proces tehničke istrage, čiji je cilj da otkrije informacije vezane za kvalitet proizvoda, uzimajući u obzir kontekst funkcionisanja proizvoda. Ovaj proces uključuje izvršavanje programa ili aplikacije sa namerom pronalaženja grešaka. Ispitivanje obezbeđuje kritiku i komparaciju stanja i ponašanja 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 funkcionalne celine

- integracione ispitne slučajeve (eng. *integration test*) – ispituje se više klasa i njihova međusobna interakcija

- 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 (tipično 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 ispitivanje performansi različitih konfiguracija sistema pri istom opterećenju. Najčešće se kao mera kvaliteta uzima protok podataka i vreme odziva.

- *Ispitivanje bezbednosti* – ispitivanje da li su funkcije dostupne onim i samo onim korisnicima kojima su i namenjene.

- *Ispitivanje robustnosti* – provera otpornosti na otkaze

- *Ispitivanje u stresnim uslovima* – ispitivanje pouzdanosti sistema pod nenormalnim uslovima (velika opterećenja sistema, nedovoljno memorije ili drugih resursa, neraspoloživi servisi i sl.).

- *Ispitivanje zagušenja* – provera da li sistem može da zadovolji višestruke zahteve različitih aktera za istim resursom.

- *Ispitivanje instalacije* – ispitivanje procesa instaliranja softvera na različitim sistemima i u različitim 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** (eng. *White box testing*)
2. Ispitivanje metodom **crne kutije** (eng. *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 (eng. White-box testing)

Ova metoda ispitivanja podrazumeva proveru i analizu izvornog koda i zahteva dobro poznavanje programiranja, odgovarajućeg programskog jezika, kao i izvornog koda konkretnog softverskog proizvoda. Plan ispitivanja se određuje na osnovu elemenata implementacije softvera, kao što su programski jezik, logika i stilovi kodiranja. Ispitni slučajevi 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 ispitnim slučajevima može se proveravati postojanje beskonačnih petlji ili koda koji se nikada ne izvršava[4].

Jedan od načina da se osmisli dobar skup ispitnih slučajeva bele kutije je na osnovu kontrole toka programa. Vizuelna reprezentacija skupa ispitnih slučajeva, radi preglednosti, je u formi *grafa kontrole toka programa*. Da bi bio osiguran odgovarajući skup ispitnih slučajeva potrebno je uzeti u obzir različite aspekte ovog grafa. Adekvatnost ispitinih slučajeva meri se metrikom koja se zove *pokrivenost* [5].

Metoda bele kutije uvodi više nivoa pokrivenosti, i to:

- Pokrivenost iskaza – Pokrivenost iskaza je procenat iskaza koji su izvršeni od strane ispitnih slučajeva. Cilj je ostvariti 100% pokrivenosti iskaza kroz ispitivanje. Da bismo postigli pokrivenost iskaza, moramo izabrati ispitni slučaj 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 povoljnim brojem ispitnih slučajeva primora svaku liniju koda da se izvrši bar jednom.)

- **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*). 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.

- **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. Izvršavanjem putanja, ne uzimajući u obzir broj iskaza u njima, većina scenarija bi bila pokrivena.

2.1.1 Prednosti i mane ispitivanja metodom bele kutije

Neke od prednosti ispitivanja metodom bele kutije uključuju i to da je poznavanjem unutrašnje strukture (tj. samog koda) vrlo lako uvideti koji tip ulaznih podataka može pomoći u efikasnom ispitivanju aplikacije. Primenom ovog metoda ispitivanja dolazi i do uklanjanja suvišnih linija koda koje mogu izazvati greške, tako da ova vrsta ispitivanja ujedno predstavlja i pomoć u optimizaciji koda.

Ispitivanje metodom bele kutije sa sobom donosi i nedostatke kao što je i to da je poznavanje unutrašnje strukture uslov, što implicira potrebu za veštim ispitivačem koji će izvesti ovu vrstu

ispitivanja; ovaj uslov zahteva dodatne troškove. Takođe, 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 (eng. Black-box testing)

Ispitivanje metodom crne kutije predstavlja postupak u kome se analizira 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 ispitivanja 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 uz zadržavanje mogućnosti ispitivanja 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** – Kako bi se smanjili troškovi ispitivanja nema potrebe pisati nekoliko ispitnih slučajeva koji ispituju isti aspekt 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 važeća stanja, 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:

1. 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

2. Dizajnirati ispitne slučajeve bazirane na izvedenim klasama ekvivalencije. Neka uputstva za identifikaciju klasa su:

- Ako ulazni uslov podrazumeva opseg vrednosti, definišu se jedna važeća i dve nevažeće klase ekvivalencije
- Ako ulazni uslov zahteva određenu vrednost, definišu se jedna važeća i dve nevažeće klase ekvivalencije.
- Ako ulazni uslov podrazumeva člana nekog skupa, definišu se jedna važeća i jedna nevažeća klasa ekvivalencije.

Ako je ulazni uslov logički uslov, definišu se jedna važeća i jedna nevažeća klasa ekvivalencije.

- **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.

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

2.2.1 Prednosti i nedostaci ispitivanja metodom crne kutije

Neke od prednosti ispitivanja metodom crne kutije obuhvataju i to da ispitivanje može biti netehničko (od ispitivača se ne zahtevaju tehnička znanja). Ispitivanje metodom crne kutije će najverovatnije pronaći one greške koje bi i korisnik pronašao. Ovaj metod ispitivanja pomaže u identifikovanju nejasnoća i protivrečnosti funkcionalnih specifikacija. Ispitni slučajevi koji se koriste u ispitivanju ovom metodom mogu biti izrađeni po završetku funkcionalnih specifikacija.

Pored nepobitnih prednosti, ispitivanje metodom crne kutije sa sobom donosi i mane koje se manifestuju u vidu ograničenja da se zbog nepoznavanja unutrašnje strukture koda, može propustiti ispitivanje nekog dela koda. Takođe, detaljno ispitivanje svih kombinacija različitih ulaznih parametara za većinu aplikacija praktično je ne izvodljivo.

2.3 Ručno ispitivanje

Na početku ispitivanja definiše se lista ispitnih slučajeva koju kao vodilju koristi čovek koji ručno izvršava sve ispitne slučajeve, ponavljajući proces u svakoj iteraciji ispitivanja. Za svaki ispitni slučaj iz liste postoji dodatno polje u koje ispitivač unosi rezultat izvršavanja pojedinačnog ispitnog slučaja. Rezultat kod ručnog ispitivanja može imati jednu od sledećih vrednosti:

- *Pass* – ispitni slučaj je uspešno izvršen, dobijen je očekivani razultat.

- *Fail* – ispitni slučaj nije uspešno izvršen, nije dobijen očekivani rezultat.

- *Pass with exception* – ispunjene su bitne stavke iz *Pass/Fail* kriterijuma, ali 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, ispitni slučaj 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 ispitnog slučaja 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 predviđenim ishodom, postavljanje probnih preduslova, kao i kontrolu 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 ručni ispitni slučaji mogu biti neefikasni u pronalaženju određenih klasa grešaka. Automatizacija ispitnog slučaja je proces pisanja računarskog programa koji služi za izvršavanje procesa ispitivanja, 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. čak i najmanje promene u kodu mogu uzrokovati kvar aplikacije, iako je aplikacija u predhodnim verzijama radila normalno. Zbog toga je potrebno ponavljati ispitivanje nakon svake izmene. Postoje dva opšta pristupa automatizaciji ispitnih slučajeva:

- Kodom upravljivo (eng. *Code-driven*) ispitivanje. 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.

- Ispitivanje grafičke korisničke sprege (*GUI*). 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 karakteristike proizvoda za automatizaciju, od koje u velikoj meri zavisi uspeh automatizacije. Automatizaciju nestabilnih funkcija ili funkcija 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 različitim okolnostima. Ispitnim slučajevima se opisuju ispiti koji moraju da se pokrenu nad programom da provere da li program radi kao što je očekivano. Ključna primena kodom upravljive (eng. c*ode-driven*) automatizacije je agilni razvoj softvera[8], 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. [9])

2.4.2 Ispitivanje grafičke korisničke sprege (GUI)

U svom ranijem razdoblju, aplikacije su svoje izvršavanje bazirale na komandama. Korisnik bi pamtio i unosio manuelno komande, putem tastature, a kao odgovor na unete komande sistem bi izvršavao određene akcije. Naprednije aplikacije, tog vremena, bi na ekranu izlistavale moguće komande, koje korisnik može da unese, i od korisnika očekivale da odabere neku od ponuđenih opcija. Trenutno je softverska industrija u eri prozora, aplikacije se koriste kroz neki oblik grafičke korisničke sprege (eng. *GUI - Graphical User Interface*). Komponente grafičke korisničke sprege često se oslovljavaju terminima prozor, ikona, meni, pokazivač. Korisnik sa tim elementima intereaguje pomerajući miša, aktivirajući različite grafičke forme pritiskom na levi/desni taster miša, i kombinacijom određenih tastera na tastaturi. Time su grafičke korisničke sprege uticale da programi dobiju na poboljšanju što se tiče aspekta lakoće korišćenja (eng. *user friendly*)

Zbog toga što se ispitivanje i verifikacija kvaliteta aplikacije koja se oslanja na korišćenje grafičke korisničke sprege zasniva na izvršavanju iz perspektive krajnjeg korisnika softverskog rešenja, izdiže proces do nivoa od vitalnog značaja i čini veliki udeo u samom procesu obezbedjivanja kvaliteta. Ovaj vid ispitivanja takođe može pokriti čitavo softversko rešenje zbog činjenice da se kroz grafičku korisničku spregu koristi sama funkcionalnost aplikacije.

Da bi se ispitni slučaj za proveru funkcionalnosti softverskog rešenja automatizovao, u vreme aplikacija koje su bile bazirane na konzolnom izvršavanju, ispitivači su se oslanjali na skripte koje su sadržale kolekciju linijskih komandi. Samo izvršavanje tih komandi bilo je nezavisno od stanja i dešavanja na ekranu. Ispitivanje komponenti grafičke korisničke sprege je drugačije i otežano time što skripte koje se koriste u samom procesu ispitivanja moraju biti sposobne da promene sadržinu bafera ulaznog niza podataka (eng. *reassign*), da pomeraju pokazivač miša, vrše aktiviranje pritiska tastera miša i različite pritiske pojedinačnih, kao i kombinacije tastera na tastaturi. Potom, ispitne skripte moraju posedovati mogućnost da sačuvaju reakciju sistema na te akcije i dinamičke promene koje rezultuju njima. Poredeći dobijene rezultate u toku izvršavanja određenih test scenarija sa unapred pripremljenim polaznim i očekivanim vrednostima (eng. *baseline*) skripte za ispitivanje grafičke korisničke sprege su u mogućnosti da prijavljuju greške (eng. *bugs*).

Široko rasprostranjen i prihvaćen pristup generisanju skript datoteka za ispitivanje grafičke korisničke sprege oslanja se na metod akvizicije i reprodukcije (eng. *capture/playback*)[10]. Ovom metodom se od osobe koja se bavi ispitivanjem grafičke korisničke sprege traži da obavlja zahtevne/naporne interakcije sa elementima grafičke korisničke sprege, bilo putem miša ili tastature. Ovi koraci u interakciji se zatim čuvaju snimanjem u formi skript fajla, čijom se naknadnom reprodukcijom dolazi do testa kojim se automatizuje neka akcija. Ispitne skripte koje se ovim putem kreiraju su često fiksne (eng. *hard-coded*). Svaka promena metoda unosa zahteva da se ispitna skripta ponovo generiše. Takođe, regresivno ispitivanje elementa grafičke korisničke sprege, koji su još uvek u razvoju, nije moguće obaviti ovim putem. Skripte su vrlo rigidne u tom ispitnom scenariju. U većini slučajeva, jako je teško generisati temeljan test kojim bi se pokrili svi elementi grafičke korisničke sprege. Ljudska interakcija sa elementima grafičke korisničke sprege, tokom procedure akvizicije i reprodukcije, često u sebi sadrži i greške (aktiviranje levog tastera miša van zone grafičkog elementa koji smo hteli da koristimo, pritisak pogrešnog tastera na tastaturi isl.) što kao rezultat proizvodi pojavu redundantnih informacija kojim se popunjavaju ispitne-skripte.

2.4.2.1 Aspekt automatizacije ispitivanja grafičke korisničke sprege

Bazirajući se na trenutnim tehnologijama ispitivanja grafičke korisničke sprege automatizacija procesa ispitivanja sa sigurnošću zahteva ručnu izradu, promenu i traženje grešaka u skriptama. Poredeći proces ispitivanja grafičke korisničke sprege sa drugim vidovima ispitivanja, dobija se činjenica da ovaj oblik ispitivanja iziskuje veća novčana ulaganja u pribavljanju alata, kao i za napore koji moraju biti uloženi u procesu održavanja ispitnih slučajeva. Ne retko se samo delovi ispitivanja grafičke korisničke sprege mogu automatizovati, dok se ostale celine i dalje moraju ispitivati ručno. Veliku ulogu u automatizaciji izvršavanja ispitnih slučajeva nosi i samo iskustvo u ručnom izvršavanju ispitnih slučajeva, i znanje pribavljeno u tim scenarijima može biti iskorišćeno za kreiranje ispitnih skripti za različite ispitne slučajeve. U praksi se pokazalo da tim veštih ispitivača ručnom metodom, može napraviti skript sa višim nivoom automatizacije od onih koji to nisu.

Komponenta grafičke korisničke sprege može biti identifikovana po njenom imenu koje joj sistem dodeljuje, njenoj poziciji u hijerarhiji, klasi komponenata kojoj pripada, naslovu prozora njenog nadređenog elementa (eng. *parent*), kao i na osnovu oznake (eng. *tag*) i identifikatora (eng. *ID*) koji joj je dodeljen od strane programera. U toku svake od sesija izvršavanja aplikacije, komponente grafičke korisničke sprege locirane su na osnovu jedinstvenog para koordinata na ekranu. Na žalost, položaj elementa grafičke korisničke sprege često se menja od strane programera koji je razvija, platforme, rezolucije ekrana... Skripta za ispitivanje elementa grafičke korisničke sprege treba da izbegne fiksno kodiranje (eng. *hard-coded*) pozicije elementa na ekranu. Ime komponente, takođe, ne mora biti od velike koristi, zato što se programeri ne retko odlučuju da isto ime dodeljuju različitim komponentama grafičke korisničke sprege. Uobičajeno je da se programerski dodeljene osobine grafičkog korisničkog interfejsa, kao što su tekst labele (eng. *label text*), nazivi dugmeta (eng. *button captions*), nazivi prozora (eng. *window titles*) itd. ne menjaju i da su jedinstveni, tako da su pogodni za jednoznačnu identifikaciju elemenata grafičke korisničke sprege.

Razvoj ispitne skripte treba da kombinuje, i po mogućnosti dodeljuje kvalifikacione razrede/težinu za svaki od ovih obeležja; i njihovim kombinovanjem da se dođe do jednoznačne identifikacije elementa grafičke korisničke sprege.

2.4.2.2 Z-order

Operativni sistem *Microsoft Windows* koristi Z-order hijerarhiju nad više (eng. *stack*) preklapajućih prozorskih formi. Ako uzmemo u obzir da koristimo X i Y koordinate za određivanje širine i visine ekrana, tada se više grafičkih elementa ili prozora orijentiše po imaginarnoj Z-osi koja je orijentisana od "površine ekrana" ka napolje. Prozorski elementi grafičke korisničke sprege koji su na vrhu hijerarhije Z-ordera preklapaju sve ostale prozore. Ovaj prozorski element se naziva “vršni prozor” (eng. *top-most*). Sistem održava ovu hijerarhiju tako što na vršni prozor dodaje nove prozore. Vršni prozor ima stil identifikovan kao "*WS_EX_TOPMOST*", preklapa sve ostale, nevršne prozore, time ne uzimajući u obzir podatak da li su isti aktivni (eng. *foreground*) ili su neaktivni, odnosno, u pozadini (eng. *background*). Prozori koji su "nevršni" nazivaju se *top-level* i prozori deca (eng. *child windows*). Prozor-dete je grupisan sa svojim roditeljom (eng. *parent window*) u formi Z-order hijerarhije. Prozor može postati vršni-prozor pozivanjem metode *BringWindowToTop()*, takođe se može menjati i hijerarhija prozora po Z-osi pozivanjem metoda *SetWindowPos()* i *DeferWindowsPos()*. Opcija pretrage svih prozora-potomaka (eng. *child windows*) realizovana je pozivom metode *GetTopWindow()* koja još i vraća rukovalac (eng. *handle*) najviše prozorske forme u hijerarhiji. *GetNextWindow()* metoda vraća rukovalac (eng. *handle*) na susednu prozorsku formu u hijerarhiji.

3. Koncept rešenja

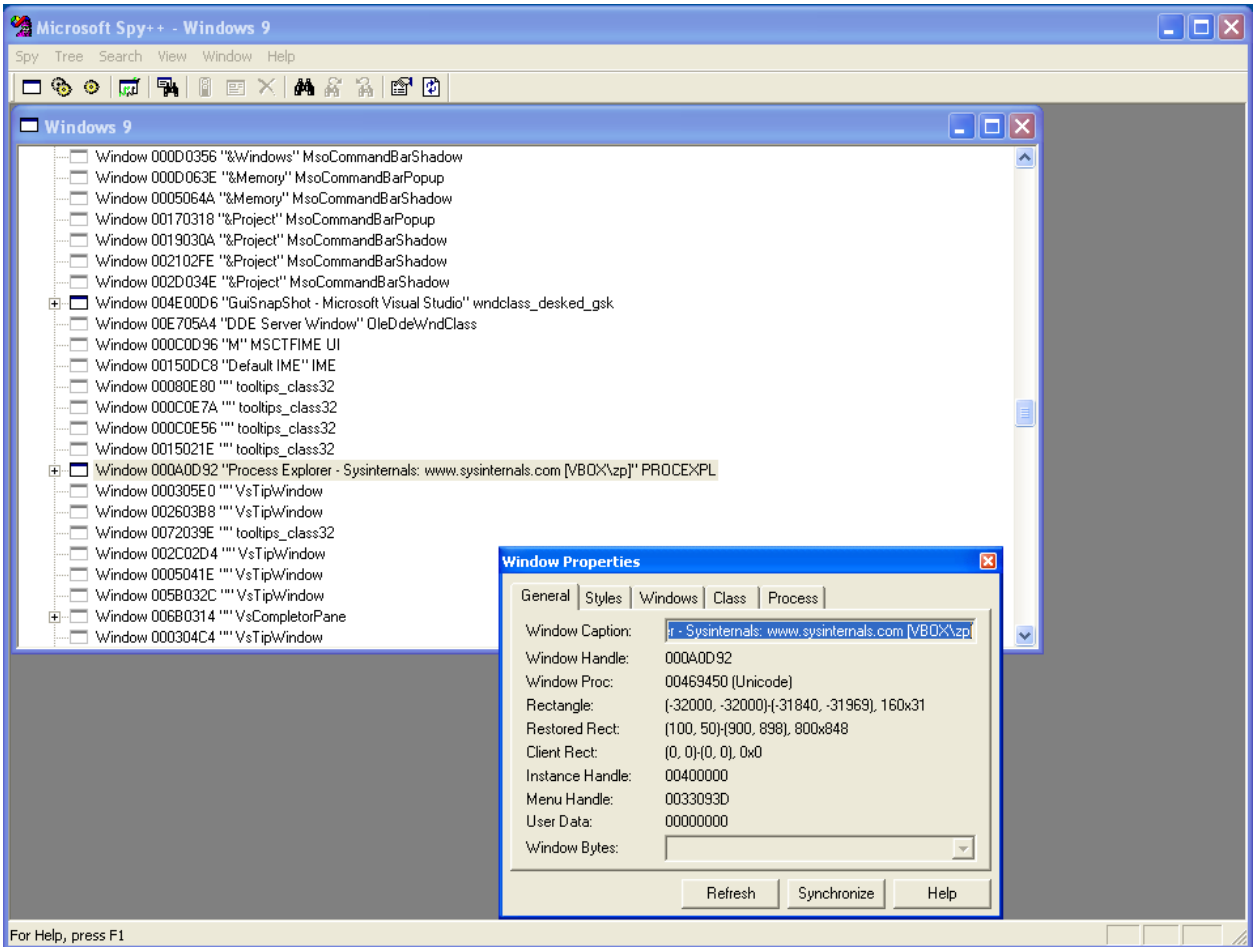
Proces identifikacije elemenata grafičke korisničke sprege je vrlo vremenski zahtevan. Od korisnika koji radi na procesu ispitivanja se zahteva višestruko ponavljanje radnji koje usled svoje monotone prirode neretko dovode do grešaka u samom procesu. Čak i sama činjenica da operativni sistem za elemente grafičke korisničke sprege ne daje intuitivna imena, ne olakšava process jedinstvene identifikacije.

Operativni sistem *Windows*, komunicira sa elementima grafičke korisničke sprege putem takozvanih rukovalaca (eng. *handle*). Jedinstvenoj identifikaciji elementa grafičke korisničke sprege doprinosi i niz parametara, koji im se dodeljuju od strane operativnog sistema. Ideja ovog zadatka je bila da se realizuje alat koji bi sličio već razvijenom alatu "*Spy++*" (alat se nalazi u okviru paketa alata koji dolaze uz *Microsoft Visual Studio* integrisanog razvojnog okruženja). Naime, i sam *Spy++* alat eksploatiše mehanizam rukovalaca kojim se služi operativni sistem kako bi identifikovao elemente grafičke korisničke sprege. Pored identifikatora rukovaoca (eng. *handle ID*), to su još i osobenosti (eng. *Properties*) grafičkog elementa razvrstane po kategorijama u: opšte (eng. *General*), stilovi (eng. *Styles*), prozorske (eng. *Windows*), klasne (eng. *Class*) i procesne (eng. *Process*). Primer odrednica u okviru grupe "opšte"

General

- *Window Caption*
- *Window Handle*
- *Window Proc*
- *Rectangle*
- *Restored Rectangle*
- *Client Rectangle*
- *Interface Handle*

- *Menu Handle*
- *User Data*
- *Window Bytes*



Slika 3-1 Odrednice elemenata grafičke korisničke sprege na primeru Microsoft Spy++ alata

Iz priloženog se može videti da sistem svakom od objekata grafičke korisničke sprege dodeljuje veliki broj odrednica, broj koji korisnik u procesu identifikacije ne bi mogao da isprati. Zato je od svih tih odrednica izabran podskup osobina koje bi imale najveći benefit po lakoći identifikacije i uparivanja sa konkretnim objektom, a pomoglo bi programskom rešenju da pouzdano odredi objekat kao jedinstven.

Podskup je sveden na sledeće karakterne odrednice:

- *Object ID*
- *WindowPosition*
- *WindowSize*
- *WindowText*
- *ParentWindow*

- *ParentId*
- *WindowStyle*
- *WindowExStyle*

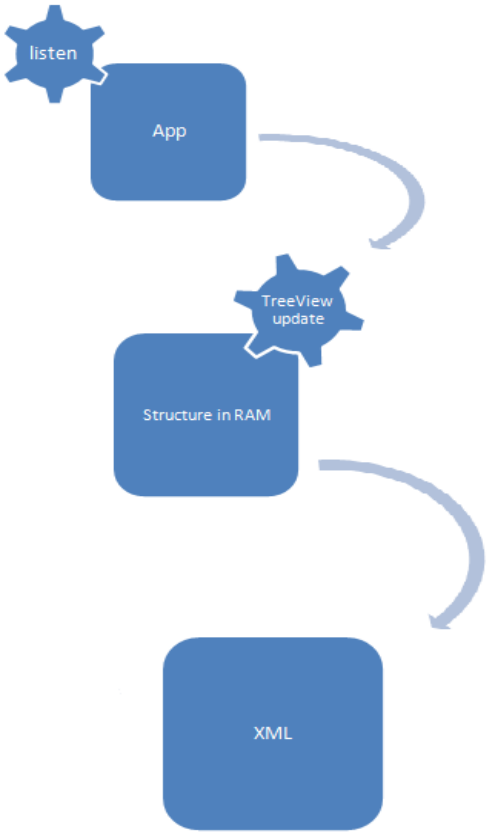
Metoda za definisanje objekta grafičke korisničke sprege kombinuje te osobine, i za svaku od njih dodeljuje različit težinski faktor koji ulazi u formulu za identifikaciju jedinstvenog elementa. Do dodeljivanja različitih težinskih faktora različitim osobinama (eng. *property*) elementa grafičke korisničke sprege došlo se iskustveno što je još jedan slučaj koji potvrđuje pravilo koliko su za izradu ispitnih slučajeva bitni inženjeri sa iskustvom u oblasti ispitivanja. One osobine na koje manje utiču sitne izmene, koje su u ispitni sistem unete bilo od strane operativnog sistema, bilo programerskim izmenama na aplikaciji koja se ispituje, ili podešavanjima grafičke korisničke sprege (npr. promena rezolucija ekrana, promena teme...) njima će se dodeliti veći težinski faktor u identifikaciji.

Jedan od zahteva koji je stavljen pred izradu realizovanog programskog rešenja je da se od strane korisnika omogući izmena vrednosti za polja "*Object ID*" kao i "*WindowText*". Razlozi su sledeći: programerski, ne retko se za različite elemente grafičke korisničke sprege dodeljuju ista ili slična imena, *ID* ili *Window Name*. Uz to, sama nomenklatura imenovanja je uglavnom neintuitivna za krajnjeg korisnika; što u daljoj interpretaciji i pokušaju jedinstvene identifikacije elementa dovodi do problema. Ovim rešenjem bi se ostavila korisniku mogućnost imenovanja, samim tim i lakšeg snalaženja u hijerarhijskoj strukturi, i same identifikacije elementa grafičke korisničke sprege. Imenovanje, takođe, prati i konvenciju kojom se korišćenjem specijalnih karaktera sledećem alatu u lancu alata za automatizaciju procesa ispitivanja grafičke korisničke sprege kaže da ime potpada pod takozvanu "*wild-card* notaciju". *Wild-card* notacija je mehanizam kojim se pokušava izbeći problem sa identifikacijom elemenata grafičkog korisničkog interfejsa koji je potpao pod dinamičko menjanje imena objekta; time što će se u daljem procesu identifikacije elementa koristiti string-nadskup kao relevantno imenovanje, koje po predpostavci ne potpada pod uticaj dinamičkog menjanja imena; a biva terminiran karakterima "*" ili "&".

Opcija "*Finder*" alata *Spy++*, koja je predstavljena u formi "mete" i koja se operativno ne razlikuje od korišćenja pokazivača miša, čijim se prevlačenjem na određenu prozorsku formu automatski dovodi korisnika do njegovog hijerarhijskog pandana u stablu međuzavisnosti. Po uzoru na tu alatku razvijena je "*highlight*" funkcionalnost realizovanog programskog rešenja. To predstavlja još jedan vid identifikacije elementa grafičke korisničke sprege.

Sumirajući, koncept rešenja predstavlja radnu nit (eng. *Worker Thread*) koja je aktivna u pozadini tokom čitavog vremena izvršavanja programskog rešenja. Njen zadatak je da nadgleda i

prikuplja svaku novu pojavu upravljačkih rukovalaca koji se generišu pri osvežavanju grafičke forme aplikacije koju osluškujemo. Radna nit potom prosleđuje osobenosti (eng. *properties*) elemenata grafičke korisničke sprege niti koja izvlači podskup osobenosti grafičkog elementa, prema unapred određenom algoritmu težinskih faktora. Odabrane osobenosti se zatim uvezuju u stablo međuzavisnosti sa predak-potomak odnosima, i ostaju u radnoj memoriji računara, zauzetoj od strane programskog rešenja. Ta struktura u radnoj memoriji podložna je daljoj manipulaciji tokom trajanja izvršavanja realizovanog programskog rešenja. Da bi se, nakon sekvenci radnji kontrolisanog prekida izvršavanja aplikacije koju nadgledamo a zatim i samog realizovanog programskog rešenja, njen sadržaj upisao u XML datoteku na perifernu memoriju računara, koja predstavlja krajnji izlaz aplikacije. Grafički prikaz koncepta rešenja sledi na Slika.3-2



.

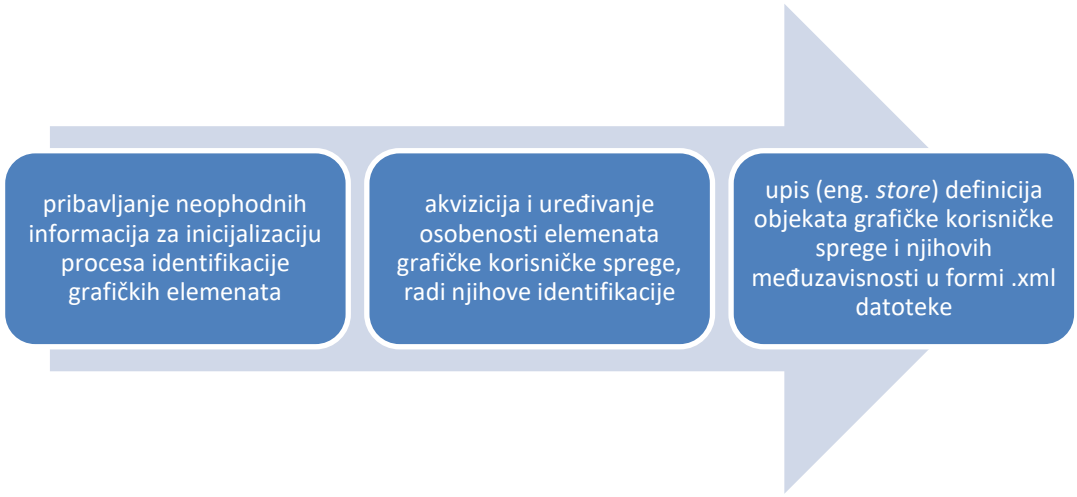
Slika 3-2 Grafički prikaz koncepta rešenja

4. Implementacija

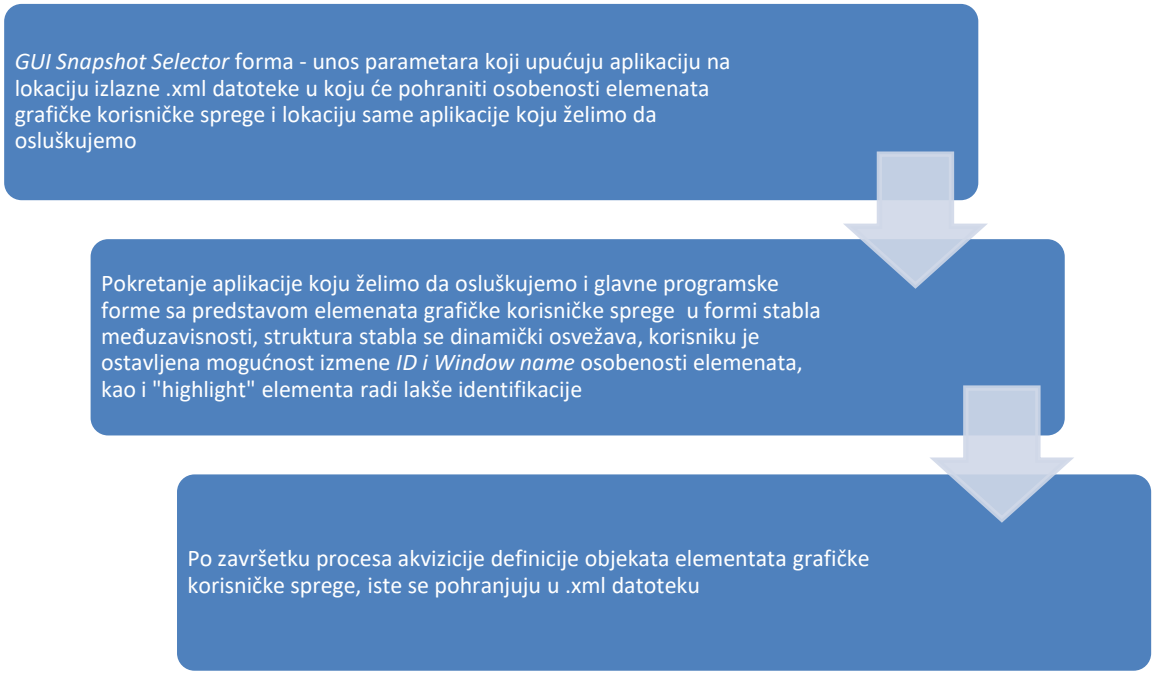
Realizovano programsko rešenje je, usled nastojanja da se očuva vertikalna kompatibilnost sa ostatkom alata iz lanca alata (eng. toolchain), implementirano kroz *Win32 API, koristeći Microsoft Visual Studio 2005 IDE*.

Temporalna organizacija rada realizovanog programskog rešenja, može se podeliti u tri faze:

- pribavljanje neophodnih informacija za inicijalizaciju procesa identifikacije grafičkih elemenata i
- proces akvizicije i uređivanja osobenosti elemenata grafičke korisničke sprege, radi njihove identifikacije,
- upis (eng. *store*) definicija objekata grafičke korisničke sprege i njihovih međuzavisnosti u formi .xml datoteke.

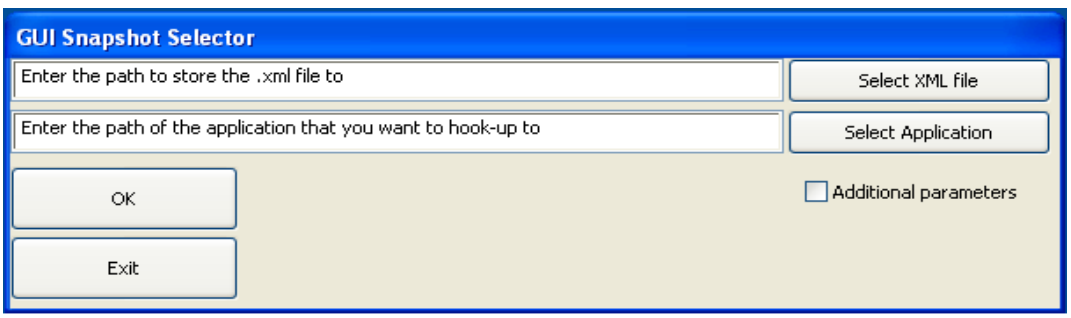


Slika 4-1 Temporalna organizacija realizovanog programskog rešenja



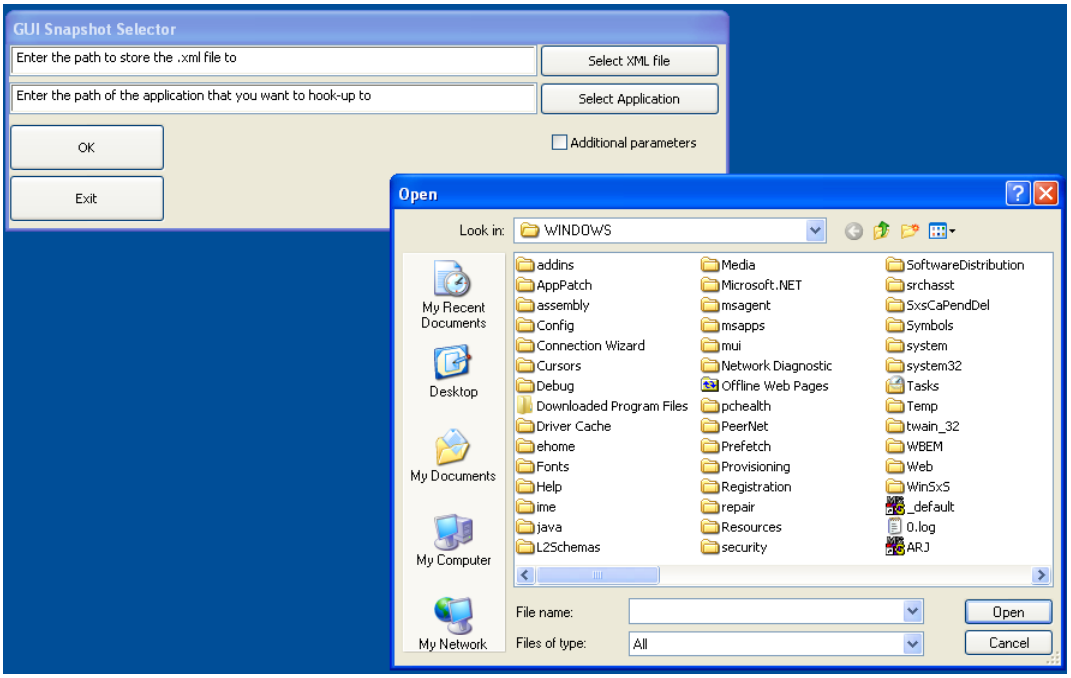
Slika 4-2 Vizuelna fazna organizacija realizovanog programskog rešenja

Realizacija projekta vizuelno je prezentovana kroz dve prozorske forme. Početna prozorska forma, koja nosi formalno-deklarativnu notaciju "*GUI Snapshot Selector*", omogućava da se od strane korisnika unesu parametri neophodni za dalje funkcionisanje realizovanog programskog rešenja. Podrška funkcionalnosti omogućena je koristeći sledeće elemente grafičke korisničke podrške: polja za izmenu (eng. *Edit field*) kojima se pohranjuju putanje na lokalnom sistemu za organizovanje podataka (eng. *Local file system*) na kojima korisnik želi da sačuva mapu objekata i njihovih hijerarhijskih odnosa u formi .xml datoteke objekata. Sledeća *edit field* forma omogućava korisniku da unese putanju ka aplikaciji sa kojom želimo da se "povezemo" (eng. *Hook-up*) od strane realizovanog programskog rešenja u cilju akvizicije parametara elemenata grafičkog korisničkog interfejsa.



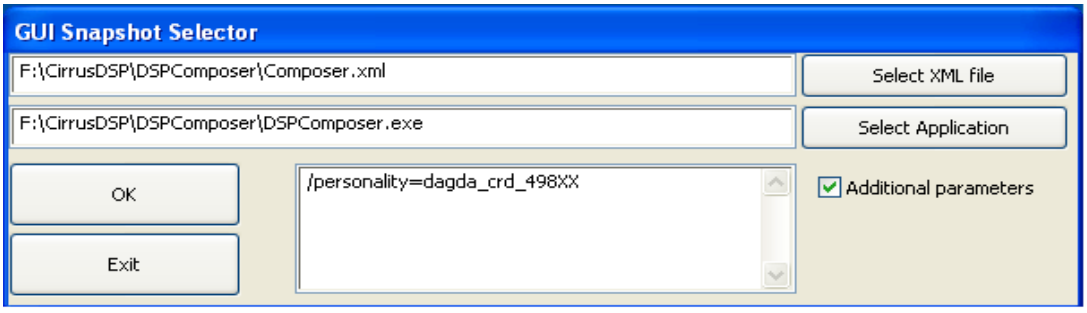
Slika 4-3 Izgled početne forme realizovanog programskog rešenja

Dati elementi grafičke korisničke sprege mogu biti popunjeni uz pomoć ručnog unosa putanja koristeći tastaturu, ili aktivacijom, levim klikom miša, na dodeljene im elemente grafičke korisničke sprege u formi tastera (eng. *button*). Rezultat te akcije je inicijalizacija nove prozorske forme, koja reprezentuje podrazumevanu (eng. *default*) formu za pretragu i selekciju datoteke na lokalnom sistemu za organizaciju datoteka (eng. *Open dialog*).



Slika 4-4 Integracija sa formom operativnog sistema

Ako se od korisnika zahteva da unese dodatne parametre pri pokretanju aplikacije čije parametre želimo da prikupimo, ta mogućnost se inicira aktiviranjem elementa grafičke korisničke sprege radio dugme (eng. *Radio button*) sa nomenklaturom "*Additional parameters*", koja kao rezultat korisniku predočava, do tada skrivenu, *edit-field* formu, u koju pohranjuje dodatne parametre neophodne za izvršavanje aplikacije.

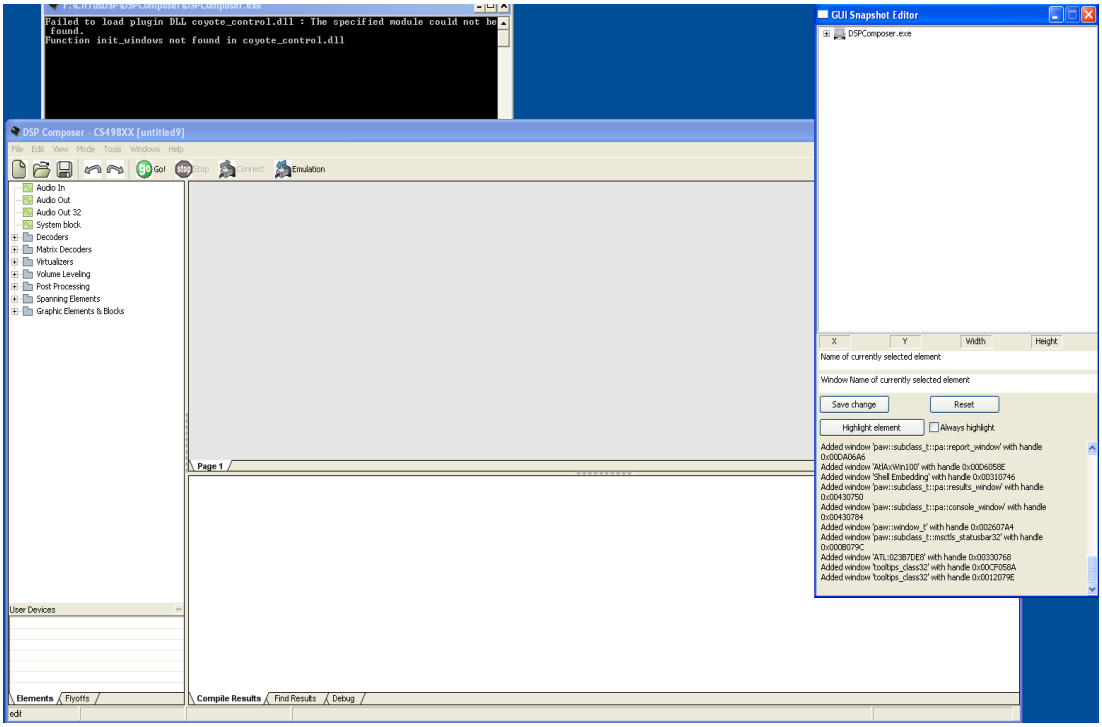


Slika 4-5 Unos dodatnih parametara aplikacije na koju želimo da se "povežemo"

Ako se neko od polja za izmenu (eng. *Edit field*) ostavi prazno, pokušaj prelaska na sledeću fazu u izvršavanju realizovane programske podrške biće onemogućen i ispraćen odgovarajućom vizuelnom notifikacijom u formi prozora sa odgovarajućom porukom "*Error: Please check if all parameters are entered*", potvrdom na koju se u fokus korisniku vraća inicijalna forma za unos podataka.

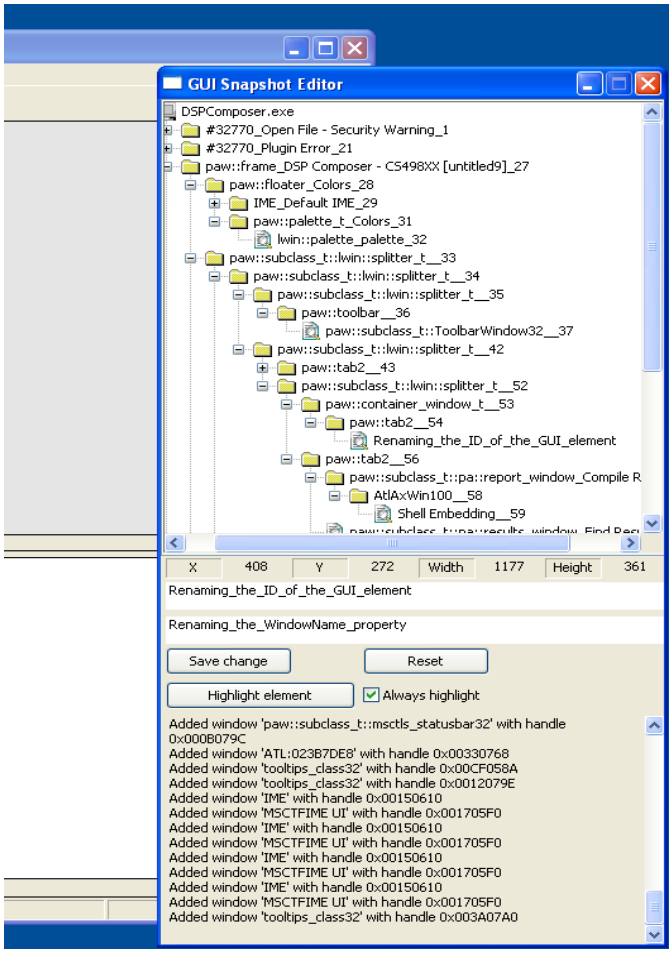
Kao rezultat uspešnog unosa parametara neophodnih za dalje izvršavanje realizovane programske podrške, prelazi se na sledeću fazu a to je samo pribavljanje (eng. *hijacking*) osobenosti (eng. *GUI properties*) aplikacije na koju smo se povezali.

Realizovano programsko rešenje, u svojoj glavnoj prozorskoj formi, akcenat stavlja na polje za izmenu koje predstavlja dominanatan elemenat u interakciji sa korisnikom. Isti predstavlja prikaz hijerarhijskog odnosa predak-potomak (eng. *parent-child*) koji je, radi preglednosti, organizovan u stablo nasleđivanja (eng. *treeview*). Stablo međusobnih hijerarhijskih odnosa, zarad zadržavanja preglednosti, izlistava samo ID parametar elementa grafičke korisničke sprege. Pozadinska nit dinamički osvežava stanje modela stabla nasleđivanja koji se u toku procesa aktivne akvizicije osobenosti aplikacije koja se osluškuje nalazi u radnoj memoriji programa. Aktivna akvizicija predstavlja process tokom koga je realizovano programsko rešenje povezano preko rukovaoca programom i da radi periodično očitavanje osobenosti aplikacije i pohranjuje je u objektnu strukturu, koja će u svojoj finalnoj formi predstavljati .xml datoteku objekata međuzavisnosti.



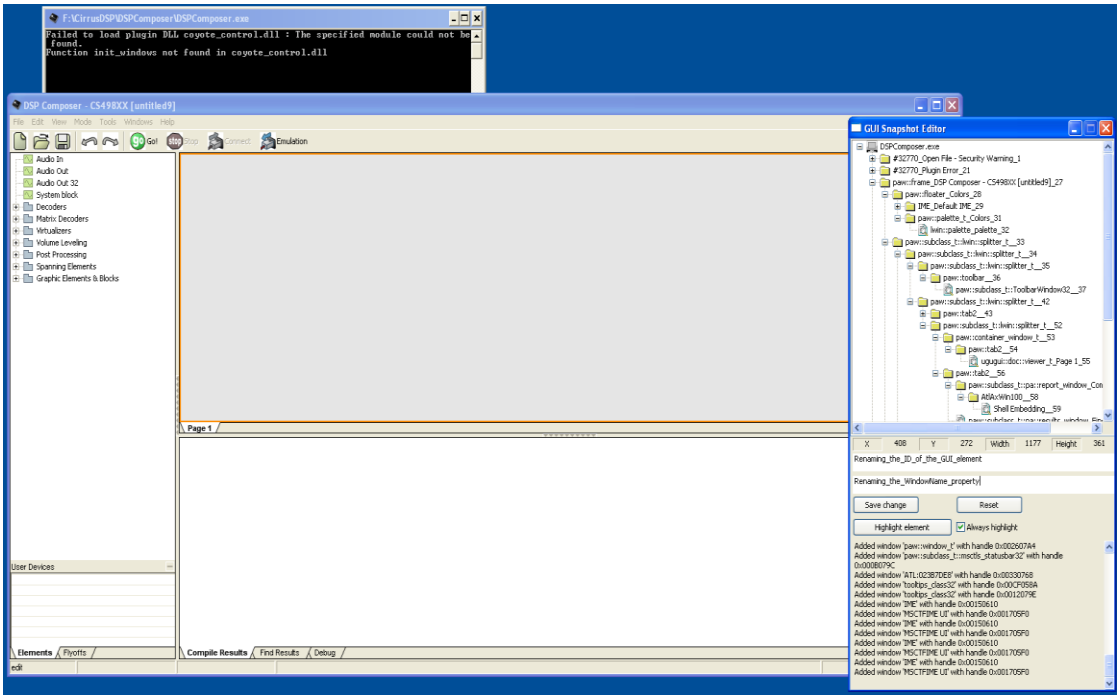
Slika 4-6 Izgled realizovane programske podrške prilikom pohranjivanja osobenosti grafičkih elemenata aplikacije koju “osluškujemo” u stablo međuzavisnosti

Ispod polja za izmenu koje je populisano stablom međuzavisnosti, nalaze se pomoćni elementi grafičke korisničke sprege, takođe u formi polja za izmenu, koji izlistavaju neke, za korisnika korisne parametre selektovanog elementa grafičke korisničke sprege, kao što su dimenzije i relativne koordinate položaja elementa u odnosu na gornji levi ugao ekrana. Korisniku je ostavljena mogućnost da se kreće kroz reprezentaciju stabla međusobnih odnosa elemenata grafičke korisničke sprege. Pritiskom levog tastera miša, korisnik obeležava (eng. *select*) određeni element iz stabla, za koji se izlistavaju parametri *ID* i *WindowName* u za to određenim elementima grafičke korisničke sprege u formi polja za izmenu, koji se nalaze niže. U skladu sa zahtevima koji su stavljeni pred realizovano programsko rešenje, vrednosti u ovim poljima su podložni izmenama od strane korisnika, zarad intuitivnije nomenklature elemenata i lakše identifikacije. Izmene koje je korisnik uneo postaju vidljive nakon aktiviranja elemenata grafičke korisničke sprege u formi tastera "*Save change*"; dok je za vraćanje na prethodni korak, pre izmene, usled moguće greške, ostavljena mogućnost aktiviranjem tastera "*Reset*".



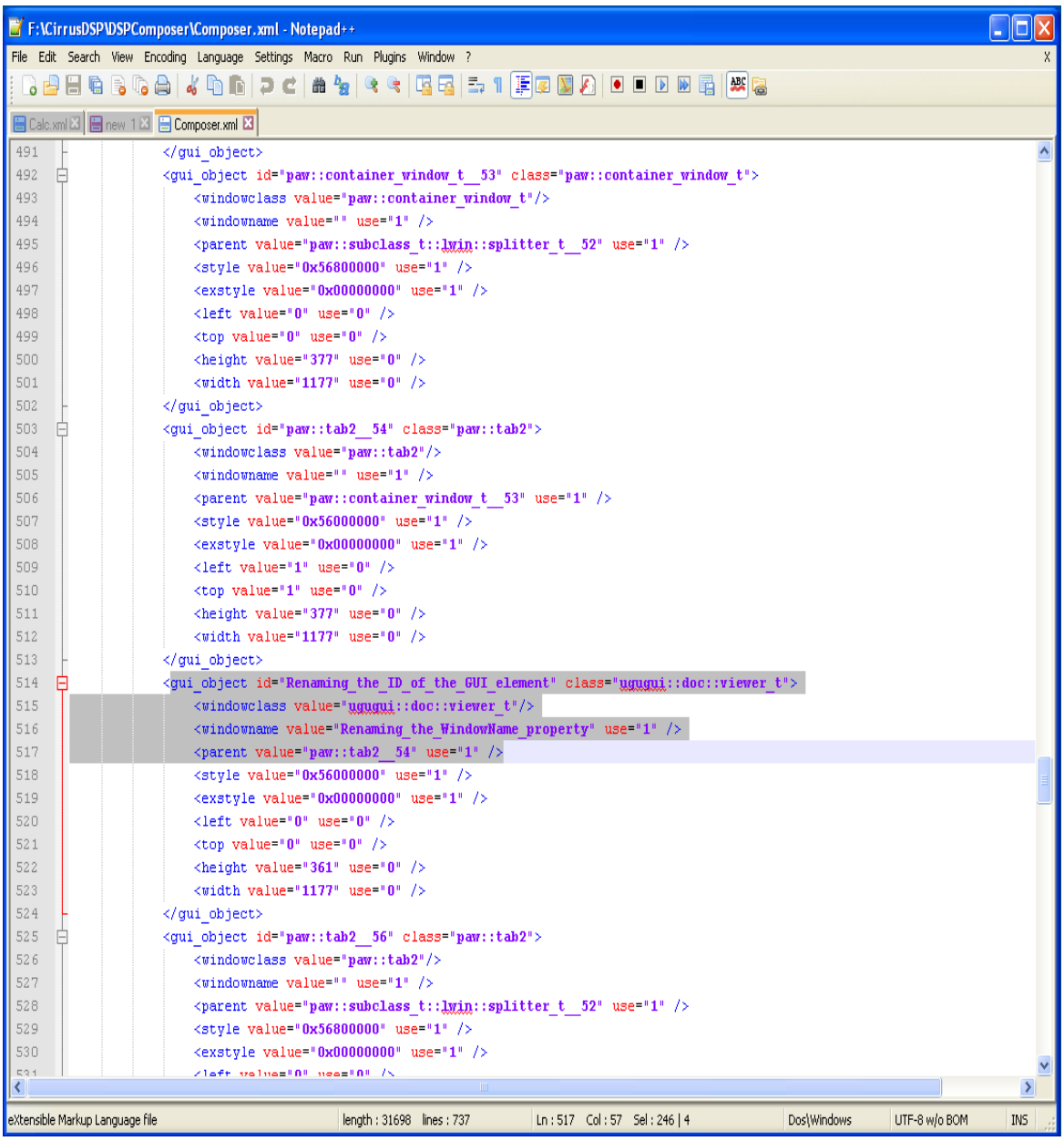
Slika 4-7 Detaljniji pogled na realizovanu aplikaciju sa populisanim stablom međuzavisnosti između elemenata grafičke korisničke sprege aplikacije koju "osluškujemo"

Još jedna od olakšica u procesu identifikacije i povezivanja selektovanog elementa grafičke korisničke sprege iz stabla međuzavisnosti sa trenutnim stanjem na ekranu, omogućena je aktiviranjem opcije "*Highlight*". Kao rezultat te akcije, iscrtava se kvadrat oko identifikovanog elementa grafičke korisničke sprege, čime postaje uočljiviji za korisnika. Elemenat grafičke sprege radio-dugme (eng. *Radio-button*) "*Always highlight*" automatski primenjuje iscrtavanje kvadrata oko selektovanog elementa grafičke korisničke sprege iz stabla međuzavisnosti.



Slika 4-8 Izgled realizovane programske podrške prilikom izbora opcije "highlight", kojom se uokviri selektovani grafički element iz stabla međuzavisnosti

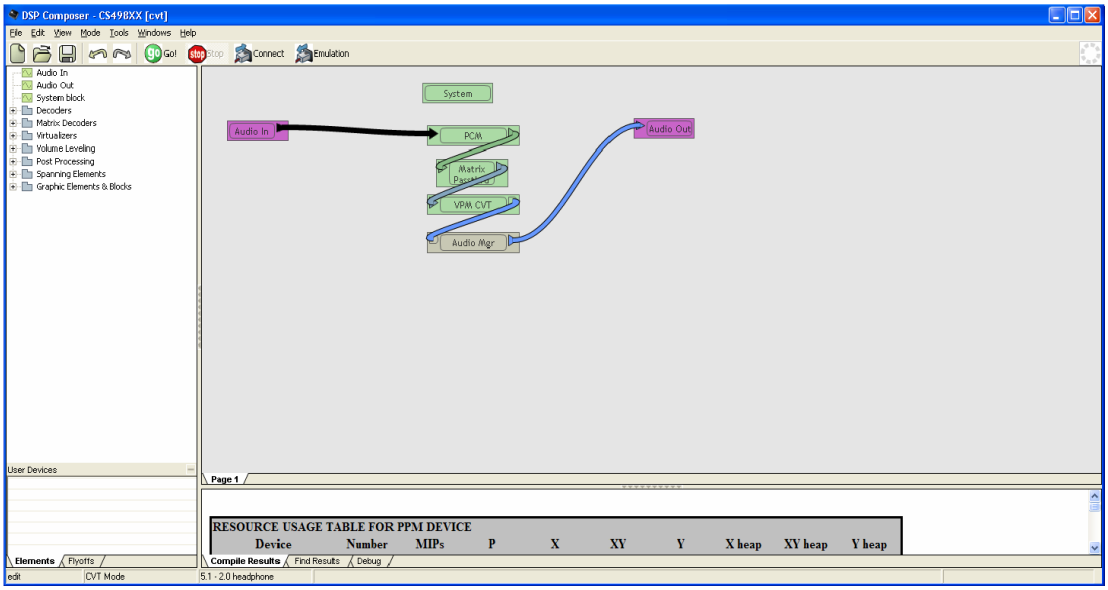
Sve vreme tokom koga korisnik vrši aktivnu akviziciju različitih elemenata grafičke korisničke sprege, osobenosti objekta se nalaze u radnoj memoriji i na njih se može uticati (promena vrednosti *ID* parametra, odnosno "*WindowName*" parametra). Kada se proces akvizicije završi, aplikacija nad kojom smo radili *hook-up* se dovede u stanje kontrolisanog prekida rada, potom se realizovano programsko rešenje takođe dovede u isto stanje, pritiskom na taster "X (rezultira da se prikupljeni parametri elemenata grafičke korisničke sprege sačuvaju u formi .xml datoteke hijerarhijskih objekata). Primer te datoteke prikazan na slici 4-9.



Slika 4-10 Izgled XML datoteke u koju smo pohranili podatke u formi stabla međuzavisnosti grafičkih elementa

5. Ispitivanje ispravnosti rešenja

Nakon razvoja alata, pristupilo se procesu ispitivanja realizovanog programskog rešenja. U tu svrhu 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*. Reč je o grafičkom alatu kojim se može dizajnirati tok audio signala u potpunosti korišćenjem „prevuci i spusti“ (eng. *drag-and-drop*) mogućnosti koju nudi grafička korisnička sprega. Kombinovanjem velikog broja, već realizovanih primitivnih elemenata za obradu audio signala, korisnici mogu, bez pisanja specijalno prilagođenog koda za ciljnu platformu, kreirati upravljački program za bilo koji processor *Crystal DSP* familije procesora firme *Cirrus Logic*. *DSP Composer* alat, takođe obezbeđuje kontrolu srednjeg sloja programske podrške (eng. *firmware*) parametara tokom procesa izvršavanja istog (eng. *run-time*). Izgled DSP Composer grafičkog alata prikazan je na slici 5-1.



Slika 5-1 Izgled DSP Composer aplikacije

Ispitivanje je vršeno na nekoliko načina, i to:

- Vizuelnim praćenjem izvršavanja akcija definisanih realizovanom programskom podrškom i poređenjem rezultata izvršavanja ispitnog slučaja sa očekivanim rezultatom
- Bit identično ispitivanje

5.1 Vizuelno praćenje izvršavanja akcija

Pod vizuelnim praćenjem izvršavanja akcija definisanih realizovanom programskom podrškom podrazumeva se verifikacija kojom se potvrđuje da je izabran objekat grafičke korisničke sprege 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 akcija bili unapred poznati (redosled otvaranja dijaloga isl.)

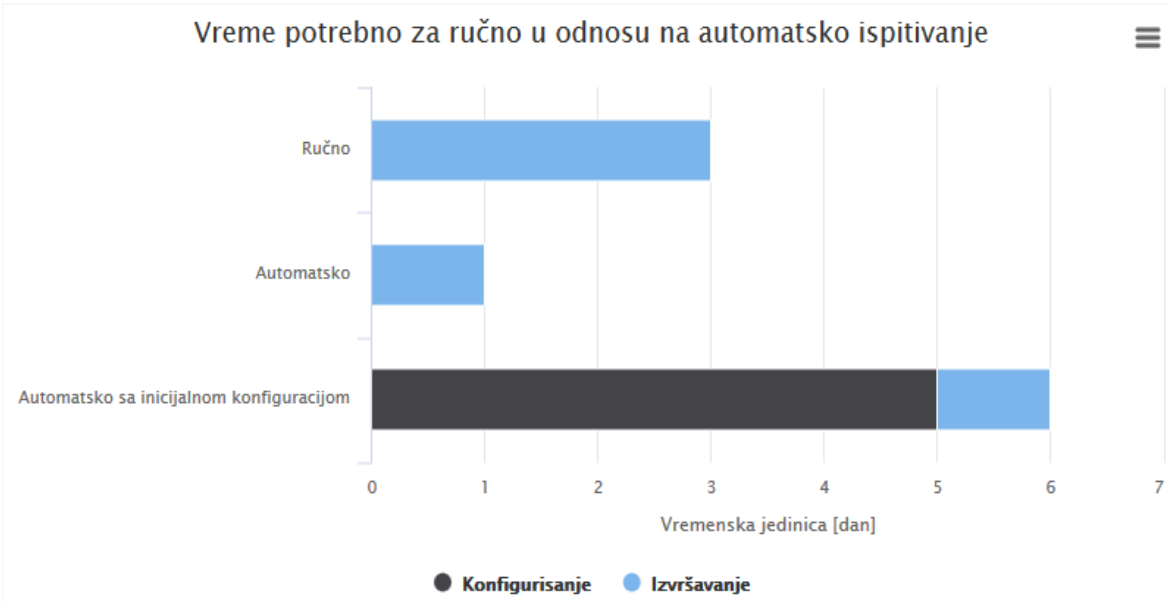
Ispitnim slučajevima je obuhvaćena osnovna funkcionalnost Cirrus Logic DSP Composer aplikacije (startovanje aplikacije, zatvaranje aplikacije, otvaranje projekta, snimanje izmena u projektu...) a akcenat je stavljen na pokrivenost svih definisanih objekata. Urađeno je vizuelno praćenje izvršavanja zadatih akcija za sve ispitne slučajeve kao i upoređivanje vremena izvršavanja sa vremenom potrebnim da se ručno izvrše svi ispitni slučajevi. Svaki ispitni slučaj je uspešno izvršen a vreme izvršavanja se smanjilo tri puta, što se moze videti u Tabeli 5-1.

Broj ispitnih slučajeva (500)		
Metod ispitivanja	Ručno	Automatski
Vreme potrebno za ispitivanje	3 dana	1dan(+5dana za inicijalnu konfiguraciju)

Tabela 5-1 Međuzavisnost metoda ispitivanja sa vremenom ispitivanja

Treba napomenuti, da se 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, treba dodati i vreme koje je potrebno da se definišu svi objekti i napišu ispitni slučajevi, koje je na postojeće vreme dodalo još pet dana za inicijalnu konfiguraciju ispitnog okruženja. I pored toga, čak i u slučaju kad već postoje napisani ručni ispitni slučajevi, pisanje novih automatskih ispitnih slučajeva je isplativo, gledajući sa strane uštede vremena, jer već nakon tri izvršavanja skupa od 500 ispitnih slučajeva, uključujući i

konfiguraciju sistema, automatski ispitni slučajevi donose prednost. Još jedna napomena, prethodna poređenja su vršena uzimajući u obzir još jedno ograničenje, radni dan od osam sati. Time se opstruira još jedna prednost automatskog ispitivanja, a to je da je njeno izvršavanje moguće tokom 24 sata, čime se razlika još više produbljuje u korist automatskog ispitivanja.



Slika 5-2 Uporedni prikaz vremena izvršavanja ručnog i automatskog ispitivanja

Ispitivanje koje je izvršeno ponovljeno je, ali uz izmene rezolucije ekrana, kako bi bila potvrđena funkcionalnost i robusnost alata za automatizaciju ispitivanja na različitim platformama (desktop, prenosni računar) koje sa sobom donose ograničenja u vidu najčešće korišćenih rezolucija ekrana. Svi ispitni slučajevi su automatski izvšavani na različitim rezolucijama ekrana. Korišćene su sledeće rezolucije:

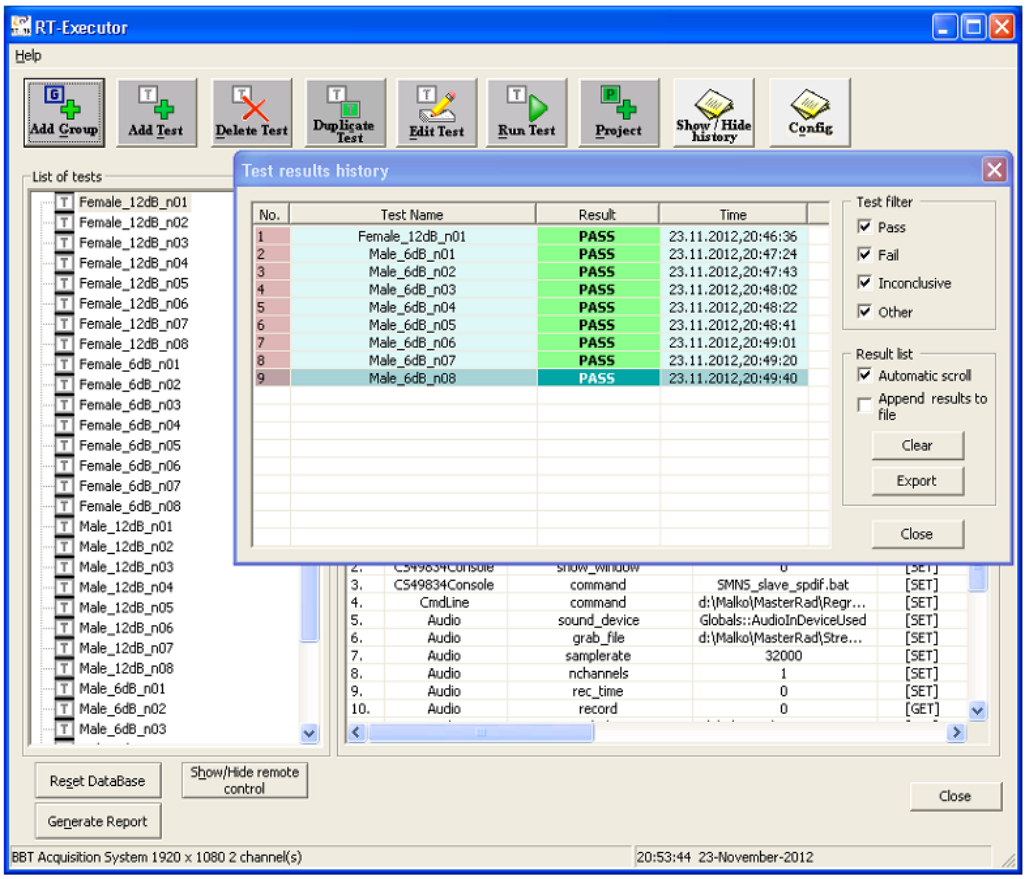
- 1680 x 1050
- 1900 x 1200

Treba napomenuti da za izvršavanje ispitnih slučajeva nije rađeno ponovno definisanje objekata i pisanje istih, nego su iskorišćeni već postojeći i definisani objekti i akcije iz prethodnog ispitivanja (koje je vršeno na rezoluciji 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.

5.2 Bit identično ispitivanje

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 (eng. *Black Box Testing* – ispitivanje metodom crne kutije). Izgled ovog alata prikazan je na slici 5-2.

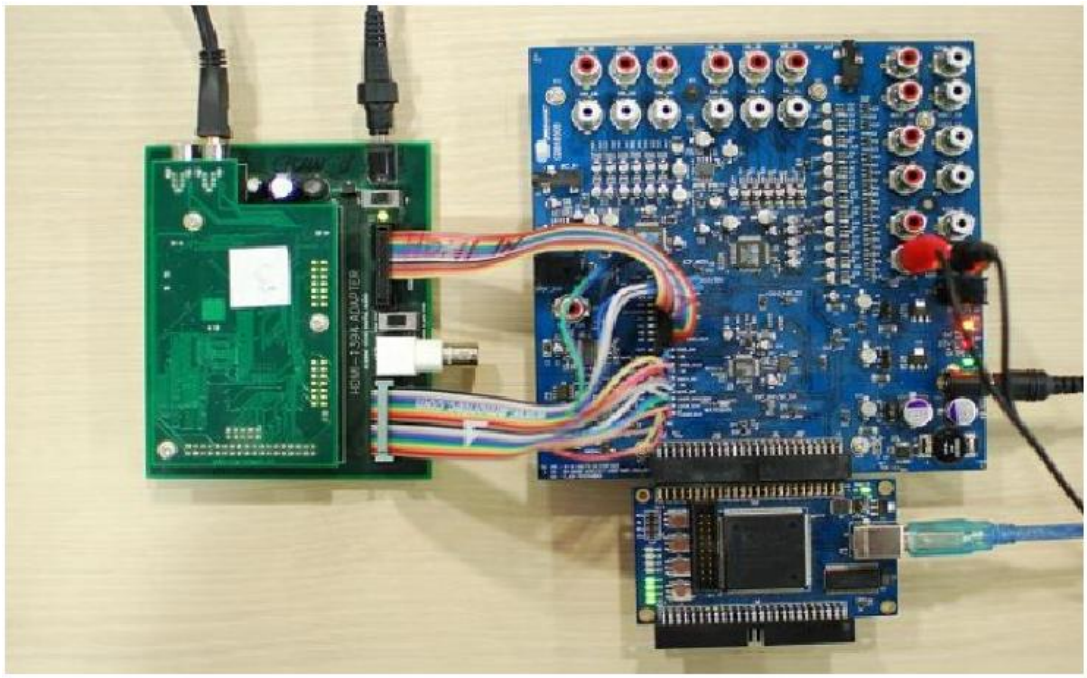


Slika 5-3 Izgled BBT aplikacije

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 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.

U okviru ovog rada urađeno je ispitivanje korišćenjem BBT alata, na takav način da je konfiguracija odnosno spuštanje *firmware*-a u ispitnim slučajevima rađeno korišćenjem GUIPlayer alata (kreiranje projekata u *DSPComposer*-u). Kao ciljni procesor korišćen je CS498XX digitalni signal procesor iz *Crystal DSP* familije procesora, firme *Cirrus Logic*, a za reprodukciju i snimanje signala korišćena je zvučna kartica kompanije *Echo Audio*[14]. Ispitno okruženje prikazano je na slici 5-4.



Slika 5-4 Izgled hardverskog dela ispitnog okruženja

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 ispitni slučajevi su uspešno izvršeni, a vreme ispitivanja se značajno smanjilo. 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 *GUISnapShot*. Osnovni zadatak ovog rada bio je realizacija vizuelnog alata koji će omogućiti jednoznačnu identifikaciju i definisanje grafičkih elemenata (dugme, polje za izmenu, meni, ...) kao dela celine koja reprezentuje aplikaciju koju nadgledamo (eng. *monitoring*). Pohranjivanje pribavljenih osobenosti elemenata grafičke korisničke sprege u formi XML dokumenta sa predak-potomak naslednim vezama između elemenata strukture. Mogućnost manipulacije (promena imena, brisanje iz strukture) pribavljenim osobenostima grafičke korisničke sprege, kao i naglašavanje na ekranu, radi lakšeg identifikovanja. Svi gore navedeni zahtevi vode ka cilju olakšanog pravljenja ispitnih slučajeva za automatizaciju procesa ispitivanja GUI aplikacija. Ovim rešenjem takođe je ostvarena značajna ušteda vremena tokom pravljenja ispitnih slučajeva, uz smanjenje aktivne angažovanosti inženjera u procesu, time smanjujući ljudski faktor koji može da rezultuje previdom. Verifikacija rešenja urađena je na grafičkom alatu za razvoj programske podrške za digitalne signal procesore kompanije *Cirrus Logic*, i to: vizuelnom verifikacijom, gde je vršena provera da li su prilikom izvršavanja ispitnih slučajeva izabrani očekivani objekti i 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. Ponavljanjem ispitnih scenarija iz prvog slučaja ali na različitim rezolucijama ekrana potvrdjena je robustnost realizovanog alata za automatizaciju ispitivanja grafičke korisničke sprege. Sledeći 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 ispitni slučajevi su uspešno izvršeni, a vreme ispitivanja se smanjilo za više od četiri puta.

Pravci daljeg unapređenja postojećeg rešenja nameću se samim time što je rešenje, zarad vertikalne kompatibilnosti sa ostatkom alata iz lanca alata realizovano za rad u okviru *Windows XP* operativnog sistema, i vode ka prilagođavanju i finalnoj tranziciji softverskog rešenja na aktuelne verzije *Windows* operativnog sistema. Ovaj pravac daljeg razvoja sa sobom donosi brojna unapređenja na nivou operativnog sistema, interakcije sa korisnikom realizovane kroz napredniji API, a sve u cilju poštovanja novih trendova u interakciji sa krajnjim korisnikom. Sledeće u nizu poboljšanja je krajnja integracija sva tri alata u jedan celoviti alat za razvoj automatskih ispitnih slučajeva grafičke korisničke sprege.

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.Eclipse Platform Technical Overview, International Business Machines Corp., 2006.

[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] Erickson, John, Kalle Lyytinen, and Keng Siau. *"Agile Modeling, Agile Software Development, and Extreme Programming: The State of Research." Journal of Database Management* 16.4 (2005): 88-100. Print.

[9] Mark Fewster & Dorothy Graham (1999). "*Software Test Automation"* ACM Press/Addison-Wesley. ISBN 978-0-201-33140-0.Harary, F. Graph Theory. Reading, MA: Addison-Wesley, 1994.

[10] Atif M. Memon, Martha E. Pollack and Mary Lou Soffa. "*Using a Goal-driven Approach to Generate Test Cases for GUIs*" ICSE '99 Proceedings of the 21st international conference on Software engineering.

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

[12] Froglogic "*Squish GUI Testing"*, http://www.froglogic.com/squish/gui-testing/index.php

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

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