Vizualizarea şi Testarea Aplicaţiilor Cu Interfeţe Grafice

Arthur-Jozsef Molnar

Departamentul de Informatică

Universitatea Babeş-Bolyai Cluj-Napoca

- Rezumatul Tezei -

Autorul a fost cofinanţat prin Programul Operaţional Sectorial Dezvoltarea Resurselor Umane, Contract POS DRU 6/1.5/S/3 - Studiile Doctorale: Prin Ştiinţă spre Societate

Februarie 2012

Cuprins

Publicaţii legate de această lucrare

Molnar, A.J.

jSET - Java Software Evolution Tracker

Prezentat în cadrul conferinței *KEPT 2011, Cluj-Napoca*

Lucrarea este disponibilă în volumul Proceedings of KEPT-2011 cu *Presa Universitară Clujeană*, sub *ISSN 2067-1180*, paginile 259–270.

Molnar, A.J.

A Heuristic Process for GUI Widget Matching Across Application Versions

Va fi prezentat în cadrul conferinței *MaCS 2012, Siofok, Ungaria*

Lucararea urmează a fi publicată în *Annales Universitatis. Scientiarum Budapestinensis, Sectio Computatorica*

Molnar, A.J.

A Software Repository and Toolset for Empirical Software Research

Urmează a fi trimisă spre publicare către *Studia Informatica UBB*, *ISSN 1224-869x*, Cluj-Napoca

Molnar, A.J.

An Initial Study on GUI Test Case Replayability

Urmează a fi trimisă spre publicare către *IEEE International Conference on Automation, Quality and Testing, Robotics AQTR 2012, Cluj-Napoca*

Introducere

Această teză este rezultatul cercetărilor mele originale pe tema vizualizării şi testări aplicaţiilor cu interfeţe grafice (abreviere GUI). Cercetările mele au fost demarate in 2008 sub supravegherea Prof. dr. Bazil Pârv.

Munca a fost focalizată în jurul a două direcţii de cercetare: imbunătatirea stării vizualizării şi analizei aplicaţiilor cu interfeţe grafice prin încorporarea ultimelor rezultate din domeniul uneltelor software academice şi furnizarea de noi metodologii în testarea aplicaţiilor ţintă.

Vizualizarea software este reprezentarea informaţiilor obţinute prin studierea sistemelor software. Cercetările noastre constau în utilizarea unor noi unelte dezvoltate în mediul academic capabile de a furniza informaţii valoroase despre dezvoltarea sistemelor ţintă studiate. Utilizăm analizorul static de cod Soot[1] ca punct de intrare al algoritmilor ce permit urmărirea evoluţiei aplicaţiilor cu interfeţe grafice ţintite(6).

Direcţia principală a cercetărilor noastre priveşte automatizarea procesului de testare a aplicaţiilor cu interfeţe grafice. Majoritatea aplicaţiilor software dezvoltate astăzi utilizează paradigma GUI pentru interacţiunea cu utilizatorii, aşadar corectitudinea lor este mai importantă ca niciodată. Munca noastră în testarea acestor aplicaţii se bazează pe modele software şi este strâns legată de eforturile de cercetare ale unei echipe de la Universitatea Maryland condusă de Prof. Atif Memon[2].

Această teză este împărţită în cinci Capitole precum urmează.

Capitolul 1 prezintă cercetările preliminare necesare. Trecem în revistă aspecte relevante legate de automatizarea procesului de testare şi prezentăm eforturi anterioare legate de cercetarea noastră. În **Capitolul 2** introducem un repozitoriu extensibil de software construit pentru a facilita cercetarea software empirică. **Capitolul 3** este dedicat cercetărilor noastre originale privind un framework extensibil de componente software care oferă funcţionalităţi avansate de vizualizare şi testare software. **Capitolul 4** conţine contribuţiile noastre originale la îmbunătăţirea procesului de mentenanţă a cazurilor de testare pentru aplicaţiile GUI. Prezentăm şi implementăm un proces euristic extensibil capabil să atingă acurateţe ridicată în potrivirea elementelor GUI echivalente funcţional şi prezentăm un studiu de caz extins pentru a studia caracteristicile noului proces. Ultima parte a muncii noastre e prezentată în **Capitolul 5**, unde

[1]Soot - http://www.sable.mcgill.ca/soot/
[2]http://www.cs.umd.edu/ atif/

eforturile noastre privind vizualizarea şi testarea software sunt unite către ţelul comun de îmbunătăţire a stării de fapt în administrarea procesului de testare a aplicaţiilor GUI.

Contribuţiile noastre originale sunt relatate in Capitolele 2,3,4 si 5 şi sunt precum urmează:

- Un set de criterii utilizabile în alegerea de aplicaţii ţintă în cercetarea software empirică.
- Un sistem de *Proiecte* utilizat în persistarea artefactelor software necesare pentru a înregistra starea unui sistem soft.
- Un set de unelte software ajutătoare care permit accesul programatic la instanţele de *Proiect*.
- Un repozitoriu de software ce conţine 30 de versiuni ale doua aplicaţii populare, complexe care utilizează interfeţe grafice.
- Fişiere script care automatizează obţinerea artefactelor software ce compun proiectele.
- Un framework extensibil care sprijină crearea şi asamblarea uneltelor software avansate.
- Trei componente software ce oferă funcţionalităţi reutilizabile atât în dezvoltarea precum şi în cercetarea software. Refolosim aceste componente în cadrul Capitolelor 4 şi 5 unde instanţiem noi unelte software bazate pe acest framework.
- Unealta *jSET - Java Software Evolution Tracker* obţinută prin asamblarea componentelor deja dezvoltate într-o unealtă pentru vizualizarea şi analizarea programelor, ce permite examinarea evoluţiei unui sistem soft dintr-o perspectivă de sus in jos, începând cu modificările survenite la nivelul interfeţei grafice până la modificările codului sursă.
- Un proces euristic în trei paşi bazat pe o listă prioritizată de euristici capabilă de a potrivi elementele funcţional echivalente ale unei interfeţe grafice între diferite versiuni ale implementării sale.
- Doi algoritmi pentru controlarea strategiei de execuţie al procesului propus.

- Mai multe euristici şi implementări de "fabrici" de euristici capabile de a atinge acurateţe ridicată în potrivirea elementelor echivalente ale interfeţelor grafice.
- O extensie a repozitoriului nostru software care conţine informaţii privind potrivirea corectă a elementelor interfeţelor grafice pentru 28 de versiuni ale aplicaţiilor FreeMind şi jEdit utilizabile în cercetări ulterioare.
- Un set de metrici general utilizabile în evaluarea acurateţii unui proces euristic de potrivire a elementelor interfeţei grafice.
- O configuraţie euristică de acurateţe mare utilizabilă ca baza unui proces de mentenanţă pe termen lung pentru cazurile de testare a unei aplicaţii GUI.
- Un studiu de caz extensiv care examinează acurateţea şi fezabilitatea celei mai bune implementări euristice dezvoltate.
- O analiză a claselor de erori euristice tipic intâlnite.
- O fundaţie teoretică pentru combinarea metricilor de acoperire a codului cu utilizarea uneltelor de analiză statică a codului în contextul aplicaţiilor cu interfeţe grafice.
- Două componente software care implementează funcţionalitatea necesară oferirii de informaţii despre acoperirea paşilor şi cazurilor de testare a aplicaţiilor cu interfeţe grafice.
- O unealtă software care permite o examinare detaliată a execuţiei cazurilor de testare pentru aplicaţii cu interfeţe grafice.
- Un studiu de caz care furnizează răspunsuri la întrebări legate de executarea cazurilor de testare existente pentru noi versiuni ale aplicaţiilor GUI ţintă în contextul evoluţiei acestora şi fezabilitatea utilizării procesului nostru pentru repararea cazurilor de testare ca acestea să funcţioneze pentru versiuni noi ale aplicaţiei ţintă.
- Un proces automatizat pentru testarea regresiilor în cadrul aplicaţiilor cu interfeţe grafice care combină cercetările noastre din Capitolele 2,3,4 şi 5 într-un tot unitar pregătit pentru a fi utilizat în industrie.

1

Introducere

Acest Capitol serveşte ca o introducere a cercetărilor noastre şi este rezultatul unei treceri în revistă a literaturii de cercetare existente în domeniul vizualizării şi testării aplicaţiilor GUI. Secţiunea 1.1.1 detaliază unele aspecte de natură generală a automatizării procesului de testare în timp ce Subcapitolul 1.1.2 este dedicat în mod special testării aplicaţiilor GUI. O scurtă trecere în revistă a stării de fapt în testarea aplicaţiilor GUI se găseşte în Secţiunile 1.1.2.1 şi 1.1.2.2. Subcapitolul 1.2 e rezervat prezentării framework-urilor Soot şi GUITAR deoarece acestea sunt utilizate pe larg în cercetarea noastră.

1.1 Testarea aplicaţiilor GUI

Acest Subcapitol examinează abordări existente în testarea software. Secţiunea 1.1.1 este dedicată trecerii în revistă a celor mai importante aspecte legate de testare şi prezentarea unor contribuţii relevante în munca noastră, în timp ce Secţiunea 1.1.2 prezintă abordări existente în testarea aplicaţiilor cu interfeţe grafice.

1.1.1 Abordări în testarea software

Importanţa în creştere a testării software a fost identificată încă din 1975 în două lucrări seminale: Goodenough a enunţat teorema fundamentală a testării împreună cu o clasificare a erorilor programelor în (14), unde prezintă un număr de programe ca exemple de analiză. El utilizează apoi aceste programe pentru a trage concluzii despre cum trebuie dezvoltate teste eficiente şi cum se derivă acestea folosind specificaţiile

programului. Următoarele secţiuni descriu acele aspecte ale procesului de testare ce sunt decisive în cercetarea întreprinsă de noi. Furnizăm informaţii despre paradigmele şi metodologiile din ziua de azi ce privesc testarea urmând ca să le referim în Capitolele următoare ale muncii noastre, unde ele sunt utilizate sau propuse ca direcţii viitoare de cercetare.

1.1.1.1 Automatizarea procesului de testare

În (1) Bach descrie în detaliu abordările de automatizare a testării specifice anilor '90 şi demontează unele presupuneri greşite privind domeniul, furnizând în locul lor abordări "de bun simţ". Ramler şi Wolfmaier abordează problematica testării din punct de vedere economic în (40), ataşând elemente legate de cost studierii acestui proces. Un raport de dată recentă a Berner et al. oglindeşte rezultatele obţinute în (1) cu privire la dificultăţile întâlnite în păstrarea unui design testabil precum şi în îmbinarea tehnicilor manuale cu cele automate în testare.

Direcţia cercetărilor noastra a fost ghidată de aceste descoperiri. În mod general vom încerca să evităm o abordare simplistă de tipul "automatizăm totul" şi să ne ocupăm de acele aspecte pe care le credem a fi cele mai potrivite pentru a rămâne în sarcina calculatorului. În plus, vom furniza noi funţionalităţi privind vizualizarea software cu rolul de a îmbunătăţi testarea manuală prin furnizarea de vizualizări între versiunile unei aplicaţii ce evoluează.

1.1.1.2 Problema oracolului de testare

Nici o discuţie pe tema automatizării testării nu poate fi completă fără a atinge problematica oracolului (42). În mod similar cu echivalentul său istoric, oracolul furnizează informaţii legate de corectitudinea executării unui caz de testare (2) prin furnizarea rezultatului corect al executării sale.

Subliniem că nici un proces de testare automată nu este viabil fără a furniza o soluţie la problema oracolului. Un oracol automatizat este un sistem software specializat care atunci când primeşte datele de intrare şi ieşire ale unui caz de testare va furniza un răspuns de tipul *succes/eroare* privind datele de ieşire ale cazului de testare fără a fi nevoie de intervenţie umană. Staats et al. discută proprietăţile importante ale unui oracol automat (42) precum *completitudinea* (este oracolul conform cu specificaţiile

programului), *relevanţa* (este rezultatul dat de oracol cel corect) şi *perfecţiunea* (este oracolul atât complet cât şi relevant).

Un efort timpuriu pentru furnizarea de oracole automate în testarea aplicaţiilor GUI a fost întreprins de Memon, Pollack şi Soffa (33). Ei au modelat formal starea interfeţei grafice utilizând aspectele teoretice descrise în (29), astfel că starea corectă a interfeţei putea fi dedusă oricând folosind starea iniţială şi secvenţa de paşi efectuată. Regăsim o variantă a acestei abordări în (26), unde un "standard de aur" e folosit pentru extragerea stării interfeţei grafice după fiecare pas pentru obţinerea informaţiei de oracol. Un studiu aprofundat privind oracolele automate a fost întreprins de Xie şi Memon (54), unde sunt studiate mai multe nivele de informaţie şi procedură a oracolului pentru a determina echilibrul optim între acurateţea obţinută şi costul asociat testării.

1.1.1.3 Evaluarea suitelor de testare

Unele din soluţiile propuse pentru evaluarea suitelor de testare folosite sunt metodele de inserare a defectelor precum testarea mutaţiilor. Tehnicile de inserţie a defectelor constau din introducerea cu bună ştiinţă a erorilor în sistemul software studiat. Pentru a aplica tehnica testării mutaţiilor, se porneşte de la programul original din care se construiesc mutanţi prin introducerea de defecte. "Puterea" unei suite de testare e evaluată prin numărul de mutanţi pe care o detectează sau "omoară", după termenul folosit în literatura de specialitate. Această tehnică permite aproximarea numărului de erori nedescoperite din program prin extrapolarea numărului de mutanţi omorâţi cu numărul total de mutanţi introduşi în aplicaţie.

Testarea prin tehnica mutaţiilor e utilizată în multe studii de caz din testarea aplicaţiilor GUI. În (54), Xie şi Memon folosesc inserarea de defecte pentru a evalua acurateţea oracolelor GUI construite. Strecker şi Memon constuiesc un număr mare de mutanţi în (43) pentru evaluarea suitelor de testa asupra detectării erorilor. Tehnica inserării de erori este iarăşi utilizată în (57), unde autorii o folosesc pentru a studia eficacitatea cazurilor de testare construite prin utilizarea de modele.

1.1.2 Starea de fapt în testarea aplicaţiilor GUI

Deoarece interacţiunea utilizatorilor cu aplicaţiile GUI se desfăşoară prin interfaţa grafică, funcţionarea corectă a acestora este considerată crucială (10). Un studiu de caz privind erorile din interfaţa grafică raportate de utilizatori în aplicaţii complexe a fost

întreprins de Robinson şi Brooks în (41), care au studiat "două sistemele industriale dezvoltate de ABB". Concluziile lor au arătat următoarele:

1. "65% din defecte au ca rezultat pierderea de funcţionalităţi ale aplicaţiei .
2. 60% din defectele depistate au fost în interfaţa grafică şi nu în codul aplicaţiei.
3. defectele din interfaţa grafică au necesitat mai mult timp pentru a fi reparate decât cele din aplicaţia ăn sine (Din (41)).

Precum au stabilit şi alte studii (10, 41), testarea aplicaţiilor GUI nu este trivială deoarece codul asociat interfeţei grafice poate cuprinde până la jumătate din codul aplicaţiei (29). Aşa cum a fost evidenţiat şi în (10), găsim multe situaţii în care erorile dintr-o aplicaţie se reflectă în interfaţa sa grafică. Acest fapt e sprijinit de studiul de caz al lui Xie şi Memon (53) unde interfaţa grafică a mai multor versiuni ale unor aplicaţii open-source populare sunt testate şi erori nedescoperite până în acel moment sunt găsite. Din acest motiv considerăm că îmbunătăţirile aduse testării aplicaţiilor cu interfeţe grafice aduc valoare adăugată produselor software prin scurtarea perioadei de descoperire a erorilor şi a măsurilor de reparare a acestora.

1.1.2.1 Unelte bazate pe captură şi reluare

Uneltele bazate pe captură şi reluare funcţionează în cele două faze care le-au şi dat numele (16). În timpul fazei de captură aplicaţia înregistrează interacţiunea utilizatorului cu aplicaţia testată şi o persistă pentru utilizare ulterioară. Datele înregistrate sunt refolosite în a doua fază, unde cazul de testare ce conţine interacţiunile înregistrate este reluat pe aplicaţia ţintă. Această abordare prezintă însă anumite dezavantaje:

- Comportamenul aplicaţiei testate este supus schimbării, ceea ce ar putea cauza marcarea unor rezultate de testare ca fiind incorecte în mod eronat.
- Schimbările din structura interfeţei grafice ce apar odată cu evoluţia acesteia cauzează ca multe interacţiuni înregistrate să nu poată fi reluate pe noile versiuni modificate ale aplicaţiei.
- În cel mai bun caz această abordare rezolvă doar o jumătate a problemei, căci crearea şi înregistrarea testalor rămâne o întreprindere complet manuală.

Din cauza acestor limitări inerente, abordările moderne combină adesea aceste unele cu tehnici bazate pe modele, precum sistemul GUI Test Generator descris de Nguyen et al. în (38), sau abordarea detaliată în (3) de Bertolini şi Mota unde autorii propun utilizarea unui framework pentru designul cazurilor de testare ce utilizează un limbaj natural controlat descris în (12).

1.1.2.2 Unelte bazate pe modele

Cele mai relevante cercetări din domeniu, atât din punct de vedere teoretic cât şi practic sunt rezultatul muncii unei echipe de la Universitatea din Maryland, condusă de Prof. Atif Memon[1].

Prima lucrare ce trebuie menţionată e chiar teza de doctorat a lui Memon (29), prezentată în 2001, în care furnizează conceptele teoretice care stau la baza eforturilor ulterioare, precum definirea clasei de interfeţe grafice la care se referă cercetarea, modelarea formală a stării interfeţei grafice precum şi definirea grafului evenimentelor, care modelează fluxul valid de evenimente din cadrul unei interfeţe grafice.

Munca echipei din Maryland este importantă deoarece ne folosim de aspectele teoretice dezvoltate, precum şi de uneltele software rezultante (descrise în detaliu în Secţiunea 1.2.2) în cercetările noastre originale detaliate în această lucrare. Prima contribuţie importantă priveşte definirea clasei de interfeţe grafice ţintite în cercetarea ulterioară, definită de Memon astfel:

Definition 1.1 *O Interfaţă Grafică cu Utilizatorul (abreviere din limba engleză GUI) este o faţadă ierarhizată a unui sistem software ce acceptă ca intrare evenimente generate de sistem şi de utilizator dintr-o mulţime fixă de evenimente şi produce ieşire grafică deterministă. O interfaţă grafică cu utilizatorul conţine obiecte grafice. Fiecare obiect are o mulţime fixă de proprietăţi. În orice moment din timp, aceste proprietăţi iau valori discrete, mulţimea acestor valori constituind starea interfeţei grafice. (Din (29))*

Bazat pe definiţia de mai sus, Memon defineşte (29) starea interfeţei grafice în funcţie de:

- *obiectele* $\mathrm{O} = o_1, o_2, \ldots, o_m$, *şi*

[1] University of Maryland - Event Driven Software Lab - http://www.cs.umd.edu/~atif/edsl

- *proprietăţile* P $= p_1, p_2, \ldots, p_n$ *acestor obiecte. Fiecare proprietate* p_i *este o relaţie booleană 1 la* n_i *pentru* $n_i \geq 1$*, unde primul argument este un obiect* o_1 *inclus în* O*. Dacă* $n_i > 1$ *ultimul argument poate fi un obiect sau valoarea unei proprietăţi şi toate valorile intermediare trebuie să fie obiecte. Astfel Memon defineşte starea interfeţei grafice la un moment dat ca mulţimea* P *a tuturor proprietăţilor tuturor obiectelor* O *conţinute de interfaţă (Din (29)).*

Pe aceste baze Memon introduce (29) graful evenimentelor, care dându-se o componentă C este definit astfel:

Definition 1.2 *Graful evenimentelor este o tuplă* $< V, E, B, I >$ *unde:*

1. V *este mulţimea vârfurilor grafului şi reprezintă toate evenimentele componentei. Fiecare* v *inclus în* V *reprezintă un eveniment al* C*.*

2. E *inclus în* $V \times V$ *este mulţimea arcelor între vârfuri. Evenimentul* e_i *urmează după* e_j *dacă şi numai dacă* e_j *poate urma imediat după* e_i*. Un arc* (v_x, v_y) *este inclus în* E *dacă şi numai dacă evenimentul reprezentat de* v_y *urmează evenimentul reprezentat de* v_x*.*

3. B *inclus în* V *este mulţimea vârfurilor ce reprezintă acele evenimente ale* C *care sunt disponibile utilizatorului la invocarea componentei.*

4. I *includ în* V *e mulţimea evenimentelor de focalizare restrânsă a componentei (Din (29)).*

Importanţa acestei lucrări, în lumina eforturilor ulterioare este că furnizează o specificare formală pentru o clasă de interfeţe grafice şi modelează structurile necesare în reprezentarea şi testarea acestora. Aspectele teoretice detaliate mai sus sunt utilizate în implementarea framework-ului de testare a aplicaţiilor cu interfeţe grafice GUITAR (25, 46) detaliat în cadrul Secţiunii 1.2.2 şi folosit pe larg în cercetările noastre prezentate în Capitolele 2,3,4 şi 5.

Aspectele teoretice detaliate în (29) au fost utilizate în multe cercetări ulterioare pe care le regăsim publicate în multe lucrări ce prezintă îmbunătăţiri aduse procesului de testare a aplicaţiilor cu interfeţe grafice (20, 26, 27, 28, 31, 33, 52, 53, 54, 55).

1.2 Framework-uri avansate de cercetare

Secţiunile următoare prezintă două framework-uri ce furnizează coloana vertebrală a cercetărilor noastre. Framework-ul Soot furnizează capabilităţi pentru analizarea statică a codului pe platforma Java, ceea ce face din el o unealtă deosebit de importantă în cadrul testării de tip white-box. Pe de altă parte, framework-ul de testate GUITAR utilizează doar informaţii disponibile din interfaţa aplicaţiei aşa că poate fi considerată o unealtă de tip black-box.

1.2.1 Frameworkul de analiză statică Soot

Această Secţiune prezintă Soot (47), un framework de cercetare a analizei statice[1] a programelor Java dezvoltat la Universitatea McGill. Aflat la versiunea 2.4.0[2] Soot are o istorie lungă marcată de multe implementări de algoritmi adăugaţi de-a lungul timpului. Website-ul său (48) menţionează o listă lungă de utilizatori din mediul academic care îl folosesc atât ca material de curs cât şi în activităţi de cercetare.

Framework-ul Soot furnizează implementări pentru analiza ierarhică a claselor - CHA, descris ca o optimizare a compilării în (13). Analiza CHA furnizează informaţii privind tipurile de instanţe ce primesc mesaje în program şi permite analiza şi optimizarea la nivelul întregului program prin determinarea grafului static de apeluri aşa cum e descris în (45).

Graful de apeluri al programului este un graf calculat static ce are ca vârfuri metodele programului şi în care fiecare arc reprezintă o relaţie de apel între metode. Deoarece este generat în mod static, nu există o ordonare între arcele de apel deoarece nu avem de unde cunoaşte ordinea în care metodele vor fi apelate la lansarea în execuţie a programului. Deoarece analiza CHA este cea mai ieftină din punct de vedere computaţional, ea cauzează o supra-aproximare semnificativă a grafului de apeluri, ceea ce a dus la eforturi în dezvoltarea de algoritmi avansaţi pentru a elimina elementele în plus (45). Asemenea algoritmi avansaţi au fost descrişi în teza de masterat a lui Sundaresan (44), care propune analiza rapidă a tipurilor. Acest algoritm ţine cont de faptul că instanţele ce primesc mesajele trebuie să fi fost instanţiate (44). O altă

[1] Static în sensul că programul nu este rulat, aşa cum se explică în (17), secţiunea 4.5.
[2] În Decembrie 2011

contribuţie importantă la dezvoltarea Soot a avut-o şi Ondrej Lhotak, dezvoltatorul framework-ului de analiză SPARK integrat în Soot (22).

Interesul nostru în Soot se datorează capacităţilor sale de construcţie a grafului de apeluri care furnizează o reprezentare precisă în cazul aplicaţiilor software complexe, aşa cum o demonstrează mai multe studii de caz (22, 23, 44). Credem că prin utilizarea acestor funcţionalităţi putem oferi unelte avansate pentru analiza şi vizualizarea software.

1.2.2 Frameworkul pentru testare GUITAR

Implementarea framework-ului de testare GUITAR a fost un mare pas în avansarea testării aplicaţiilor cu interfeţe grafice. Scopul implementării sale a fost automatizarea proceselor de testare prin includerea de unelte capabile de a genera, executa şi evalua cazurile de testare. GUITAR (46) este compus din patru componente semnificative, prezentate în ordinea în care ele sunt de regulă folosite:

- *GUIRipper* este de regulă prima unealtă utilizată. Descrisă în detaliu de Memon el al. în (25), GUIRipper porneşte aplicaţia ţintă şi salvează modelul interfeţei sale grafice într-un fişier XML. Noi utilizăm implementarea Java a acestei unealte în cercetările noastre legate de obţinerea de modele a interfeţelor grafice pentru aplicaţiile din repozitoriul descris în Capitolul 2.

- *GUI2EFG*. Această unealtă folosesşte modelul obţinut la pasul anterior şi creează graful evenimentelor aplicaţiei. Graful evenimentelor este important deoarece detaliază secvenţele valide de evenimente pentru interfaţa grafică ţintă, permiţând crearea de cazuri de testare executabile (30).

- *TestCaseGenerator*. Intrarea generatorului de cazuri de testare este fişierul XML ce reprezintă graful evenimentelor obţinut în pasul anterior. Datele de ieşire sunt cazurile de testare generate prin implementări de plugin-uri (15).

- *TestReplayer*. Această componentă poate executa cazurile de testare generate pe aplicaţia ţintă executând toate evenimentele în ordinea dată (11, 27).

2

Un repozitoriu de aplicaţii pentru cercetare software empirică

Acest capitol prezintă cercetare originală în construirea unui repozitoriu extensibil de software ce constând din mai multe versiuni ale unor aplicaţii software open-source populare care au fost alese ca ţinte pentru experimentele şi studiile de caz prezentate în cercetarea noastră. Subcapitolul 2.1 detaliază criteriile folosite în căutarea aplicaţiilor potrivite pentru a fi utilizate în cercetarea empirică iar Subcapitolul 2.2 constă din cercetări originale ce descriu modelul de date al repozitoriului şi uneltele software asociate. În Subcapitolul 2.3 descriem două aplicaţii software populare, open-source care furnizează datele din repozitoriului nostru. În final, Subcapitolul 2.4 propune noi direcţii in extinderea repozitoriului.

Contribuţiile originale sunt trecute în cadrul primului Capitol al lucrării şi au fost trimise spre publicare (9).

2.1 Criterii pentru alegerea aplicaţiilor

Secţiunea curentă descrie câteva din criteriile importante care califică aplicaţiile software ca si candidate ale cercetării empirice. Dacă noi am ajuns la criteriile prezentate din considerente specifice cercetărilor ţintite, credem că acestea sunt general valabile şi furnizează o bună fundaţie pentru alegerea aplicaţiilor ce urmează a fi incluse în experimente sau studii de caz:

- *Gradul de utilizare*: Credem că cercetările de impact trebuie să ţintească aplicaţiile software utile. În acest sens considerăm că cel mai bun semn în reprezintă o comunitate activă a utilizatorilor şi dezvoltatorilor care ghidează evoluţia aplicaţiei.

- *Complexitatea*: Software-ul este complex şi există multe metodologii şi metrici pentru măsurarea complexităţii sale. Cercetările relevante trebuie bazate pe mai multe metrici de complexitate şi trebuie legate de criteriul legat de utilizare prezentat mai sus.

- *Aspecte legate de autor*: Credem că prin alegerea de aplicaţii complet independente de efortul de cercetare întreprins putem valida generalitatea abordării alese. Cea mai bună metodă este de a alege aplicaţii ţintă produse de terţi neinteresaţi şi neimplicaţi în efortul curent.

- *Disponibilitate*: Cercetările noastre ţintesc îmbunătăţirea stării de fapt în procese legate de dezvoltarea de software. Pentru a ne valida ideile avem nevoie de acces la aplicaţii software complexe. Aceasta aduce implicaţii atât de ordin legal, cât şi de ordin tehnic, căci aplicaţiile ţintă trebuie să fie disponibile în mod gratuit şi uşor de instalat, configurat şi apoi demontat.

- *Simplitate*: Nevoie de a a acesa mai multe versiuni ale aceleaşi aplicaţii software creşte importanţa acestui criteriu. Pentru a putea avea mai multe programe instalate, configurate şi pregătite pentru executare am căutat aplicaţii care nu necesită terţe componente dificil de configurat.

2.2 Un model propus al repozitoriului

Repozitoriul nostru este modelat ca o mulţime de *proiecte*. Fiecare proiect surprinde aplicaţia ţintă la un moment dat. Având disponibile mai multe proiecte ce surprind aceeaşi aplicaţie devine posibilă studierea testării de regresie şi urmărirea şi analizarea evoluţiei programelor în timp. Fiecare proiect are asociate următoarele artefacte:

- *Fişierul de proiect*. Acest fişier XML conţine numele şi locaţia tuturor artefactelor ce compun proiectul.

- *Fişierele binare.* Fiecare proiect conţine două directoare: unul pentru aplicaţia compilată şi unul pentru bibliotecile necesare. Fiecare proiect conţine şi un fişier script ce poate fi utilizat pentru a porni aplicaţia.
- *Fişierele sursă.* Fiecare proiect are asociat un director sursă ce conţine sursele programului.
- *Modelul GUI.* Acest fişier XML conţine modelul interfeţei grafice obţinute prin rularea uneltei GUIRipper pe aplicaţia ţintă.
- *Capturi de ecran.* Acesta este un director ce conţine capturi de ecran cu toate elementele interfeţei grafice ale aplicaţiei. Aceasta permite studierea interfeţei grafice fără a porni aplicaţia.
- *Graful de apeluri.* Acesta e graful de apeluri al programului obţinut prin executarea algoritmului SPARK din cadrul uneltei Soot pe aplicaţia ţintă.

Cea mai apăsătoare limitare a modelului nostru priveşte graful de apeluri. Deoarece framework-ul Soot funcţionează doar pentru aplicaţii Java, pentru programe implementate folosind alte platforme nu se poate înregistra acest artefact.

Repozitoriul propus este structurat astfel încât fiecare proiect să aibă propriul director SVN, ceea ce uşurează căutarea şi descărcarea versiunilor individuale. Mai mult, setul nostru de unelte permite accesul programatic la datele proiectului. Fiecare proiect este reprezentat programatic printr-o instanţă a clasei *jset.project.Project* care furnizează accesul la datele ţintă. Aceste proiecte pot fi încărcate prin utilizarea clasei *jset.project.ProjectService* care furnizează metodele necesare.

2.3 Conţinutul repozitoriului

Căutarea bazată pe criteriile descrise în primul Subcapitol ne-au condus la două aplicaţii: softul FreeMind (49) de diagrame şi editorul de text jEdit (50). Ambele sunt disponibile în mod gratuit pe site-ul SourceForge şi sunt furnizate cu licenţe care permit efectuarea de modificări şi redistribuirea lor. Următoarele Secţiuni discută aceste aplicaţii.

2.3.1 FreeMind

Software-ul de diagrame FreeMind a fost dezvoltat pe platforma Java şi este disponibil prin licenţa GPL a GNU. Repozitoriul nostru conţine 13 versiuni ale aplicaţiei datate între Noiembrie 2000 şi Septembrie 2007. Tabelul 2.1 conţine detalii privitoare la versiunile descărcate precum data acestora, versiunea aproximativă corespunzătoare, numărul de clase, linii de cod şi ferestre ale aplicaţiei.

Versiunea	Datarea CVS	Clase	Linii cod	Elemente GUI	Ferestre
0.1.0	01.11.2000	77	3597	101	1
0.2.0	01.12.2000	90	4101	106	1
0.2.0	01.01.2001	106	4453	132	1
0.3.1	01.04.2001	117	6608	127	1
0.3.1	01.05.2001	121	7255	134	1
0.3.1	01.06.2001	126	7502	136	1
0.3.1	01.07.2001	127	7698	137	1
0.4.0	01.08.2001	127	7708	137	1
0.6.7	01.12.2003	175	11981	244	1
0.6.7	01.01.2004	180	12302	251	1
0.6.7	01.02.2004	182	12619	251	1
0.6.7	01.03.2004	182	12651	251	1
0.8.0	01.09.2007	544	65616	280	1

Table 2.1: Versiuni FreeMind utilizate

2.3.2 jEdit

Aplcaţia de editare text jEdit este dezvoltată pe platformă Java şi în mod asemănător cu FreeMind, disponibilă prin licenţa GPL a GNU. Pentru construirea repozitoriului nostru am utilizat un număr de 17 versiuni ale acestei aplicaţii. În mod similar cu abordarea aplicaţiei FreeMind, am ales doar versiuni distincte având cel puţin o lună de dezvoltare între ele. Prima versiune considerată este 2.3pre2 disponibilă din 29 Ianuarie 2000, iar ultima este versiunea 4.3.2final, disponibilă începând cu 20 Mai 2010. Tabelul 2.1 prezintă versiunile din repozitoriul nostru împreună cu informaţii cheie legate de fiecare versiune, similar cu tabelul echivalent pentru aplicaţia FreeMind.

Versiune	Datarea CVS	Clase	Linii cod	Elemente GUI	Ferestre
2.3pre2	29.01.2000	332	23709	482	12
2.3final	11.03.2000	347	25260	533	14
2.4final	23.04.2000	357	25951	559	14
2.5pre5	05.06.2000	416	30949	699	16
2.5final	08.07.2000	418	31085	701	16
2.6pre7	23.09.2000	456	35020	591	12
2.6final	04.11.2000	458	35544	600	12
3.0final	25.12.2000	352	44712	584	13
3.1pre1	10.02.2001	361	45958	590	13
3.1pre3	11.03.2001	361	46165	596	13
3.1final	22.04.2001	373	47136	648	13
3.2final	29.08.2001	430	53735	666	12
4.0final	12.04.2002	504	61918	736	13
4.2pre2	30.05.2003	612	72759	772	13
4.2final	01.12.2004	650	81755	860	14
4.3.0final	23.12.2009	872	106398	992	16
4.3.2final	10.05.2010	872	106510	992	16

Table 2.2: Versiuni jEdit utilizate

2.4 Concluzii şi paşii următori

Acest capitol a descris eforturile noastre în implementarea unui repozitoriu extensibil de aplicaţii software ce pot servi ca ţinte ale cercetării empirice în multe direcţii de cercetare precum vezualizarea aplicaţiilor, analiza de cod şi testarea software. Credem că acest repozitoriu este în mod special util cercetărilor legate de testarea software, testarea interfeţelor grafice şi a regresiilor.

3

Un framework extensibil pentru vizualizare şi testare GUI

Acest capitol prezintă cercetarea noastră în dezvoltarea unui framework extensibil de componente software ce furnizează capabilităţi avansate de analiză şi vizualizare. Cu excepţia Subcapitolului 3.1 care detaliază preliminariile necesare acest capitol este în întregime original.

Subcapitolul 3.1 descrie motivele dezvoltării acestui set de unelte software, în timp ce Subcapitolul 3.2 prezintă o trecere în revistă a implementării alese. Subcapitolul 3.3 prezintă trei componente reutilizabile iar Subcapitolul 3.4 introduce unealta software jSET obţinută prin asamblarea componentelor implementate. Limitările acestei unelte sunt prezentate în 3.4.3, în timp ce concluziile încheie prezentul capitol.

Contribuţiile originale ale acestui capitol au fost prezentate în cadrul Conferinţei KEPT 2011 (6) şi sunt disponibile în formă extinsă în (5).

3.1 Introducere

Prin studiere evoluţiei unor medii de dezvoltare precum Eclipse (18, 19) putem observa că fiecare versiune nouă propune unelte mai complexe pentru a ajute profesionişii în construirea rapidă a software-ului de înaltă calitate. Aruncând însă o privire atentă observăm însă că cele mai multe abordări nu includ unelte care utilizează algoritmi complecşi pentru a furniza vizualizare şi analiză unificată asupra diverselor straturi ale aplicaţiilor GUI. Noi adresăm această problemă prin dezvoltarea acestui framework de

componente pe plaftorma Java utilizând o abordare modulară ce permite asamblarea componentelor disponibile în noi unelte software ce furnizează funcţionalităţi avansate.

3.2 Design extensibil

Celula de bază a framework-ului nostru este *Componenta*, care este o unitate software ce include comportament şi prezentare grafică pentru a furniza funcţionalităţi avansate. Noile implementări trebuie să extindă clasa *AbstractView* care furnizează funcţionalităţi de bază şi un dicţionar al proprietăţilor ce reprezintă contextul componentei. Instanţierile framework-ului nostru constau din multiple asemenea componente care atunci când sunt combinate furnizează capacităţi noi de analiză şi vizualizare. Următoarele Secţiuni ale acestui Subcapitol detaliază câteva din componentele existente, iar implementări adiţionale se găsesc în Subcapitolul 5.1.

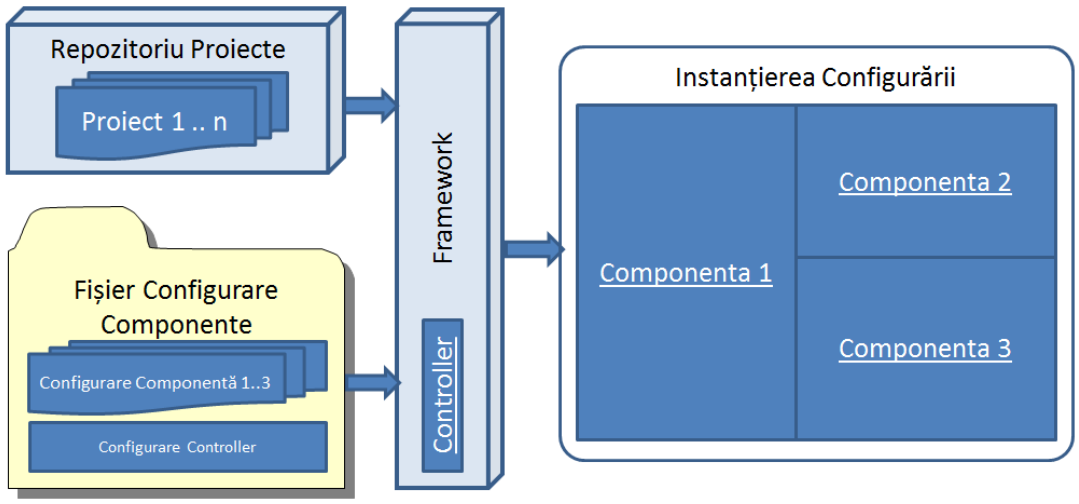

Figure 3.1: Instanțierea Frameworkului

Uneltele software implementate folosind framework-ul nostru sunt instanţiate utilizând un fişier de intrare XML reprezentat prin dosarul galben. Cea mai importantă informaţie e reprezentată de *Controllerul* utilizat, deoarece acesta e responsabil cu legarea componentelor şi furnizarea de comportament unitar al aplicaţiei. Când unealte este executată, framework-ul afişat în Figura 3.1 instanţiază controller-ul dat împreună cu toate componentele necesare. Datele de intrare sunt în forma *Proiectelor* descrise în cadrul Subcapitolului 2.2 din Capitolul 2.

Prima unealtă implementată este jSET, detailată în Subcapitolul următor în timp ce o altă implementare e prezentată în Secţiunea 5.1.4.

3.3 Componente Implementate

Această secţiune descrie componentele ce formează aplicaţia jSET. Toate aceste componente vor fi reutilizate în cadrul altor unelte folosite în cercetările descrise în această lucrare.

3.3.1 Componenta GUI Compare View

Această componentă a fost implementată pentru a furniza o reprezentare vizuală a interfeţelor grafice ţintă. Ea utilizează o structură de date ierarhică unde ferestrele aplicaţiei sunt reprezentate pe al doilea nivel, iar elementele fiecărei ferestre sunt descendenţi ai acesteia.

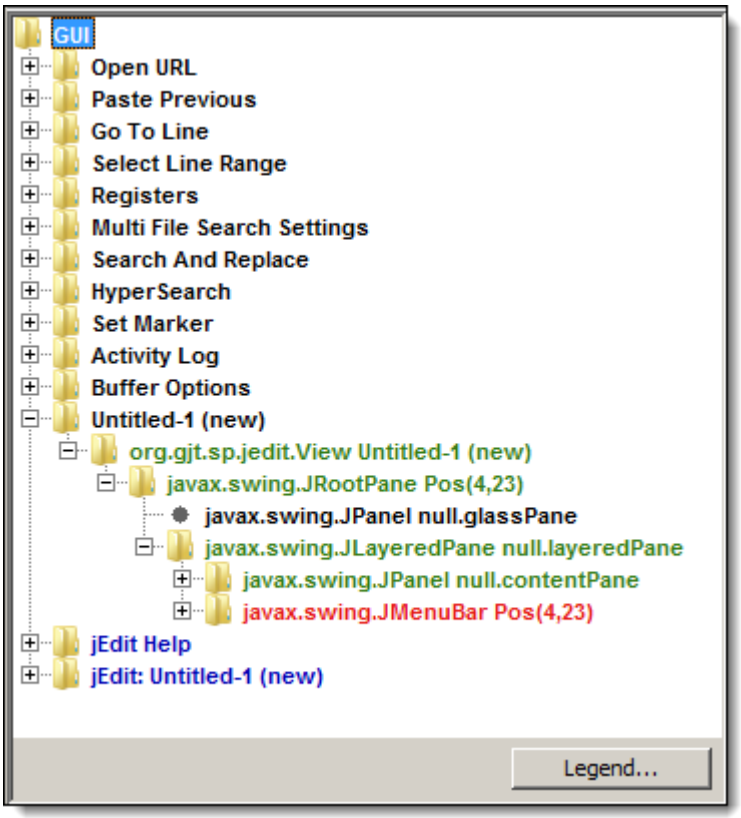

Figure 3.2: GUI Compare View

Principala diferenţă care diferenţiază implementarea noastră de altele, precum aplicaţia SwingExplorer este capacitatea de a integra două versiuni ale interfeţei grafice ale aceleaşi aplicaţii ţintă într-o singură ierarhie. De exemplu, în Figura 3.2 sunt afişate versiunile interfeţei grafice pentru aplicaţiile jEdit versiunile 2.3pre2 şi 2.3final. Ierarhia prezentată este calculată comparând ierarhiile celor două interfeţe grafice, elementele fiind potrivite utilizând proprietăţile existente în model. Trebuie să notăm că elementele interfeţei sunt codificate utilizând culori. Roşul reprezintă elemente care nu se regăsesc pe versiunea nouă, în timp ce elementele noi sunt afişate cu albastru. Culoarea verde e rezervată elementelor grafice prezente în ambele versiuni dar care au fost afectate de schimbări în codul sursă.

3.3.2 Componenta Widget Information View

Această componentă este cea mai importantă din cele implementate. Ea e folosită pentru a furniza informaţii despre elementele interfeţei grafice pentru proiectele încărcate. Implementarea curentă utilizează trei taburi: primul afişează elementul grafic în mod vizual, iar următoarele două furnizează informaţii privind proprietăţile elementului grafic şi codul care se execută la interacţiunea cu acesta, respectiv.

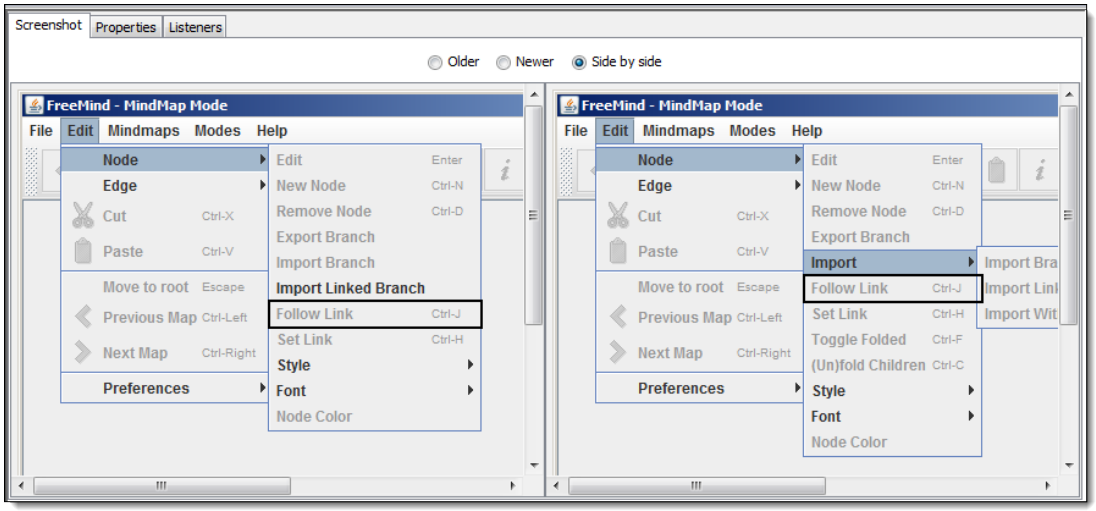

Figure 3.3: Widget Information View

În Figura 3.3 componenta este utilizată pentru a afişa doi itemi echivalenţi funcţionali din meniul aplicaţiei FreeMind. Precum în cazul componentei descrise mai sus, ex-

istă implementări alternative precum SwingExplorer, sau uneltele pentru construirea interfeţelor grafice Swing GUI Builder[1] din NetBeans sau Window Builder[2] din Eclipse.

3.3.3 Componenta Call Graph View

Această componentă reprezintă una din cele mai importante contribuţii ale noastre prezentate în cadrul acestui Capitol. Componenta *Call Graph View* furnizează funcţionalităţile necesare pentru vizualizarea grafului de apeluri al aplicaţiilor ţintă. Precum am prezentat în primul Capitol, graful de apeluri a unei aplicaţii Java constă de regulă din multe zeci de mii de noduri, din care cele mai multe aparţin de platforma în sine. Din acest motiv graful nu poate fi afişat în mod complet. Pentru a rezolva această problemă, implementarea noastră grupează toate metodele din platforma Java şi biblioteci în noduri etichetate cu *"Framework"*, precum se vede în Figura 3.4.

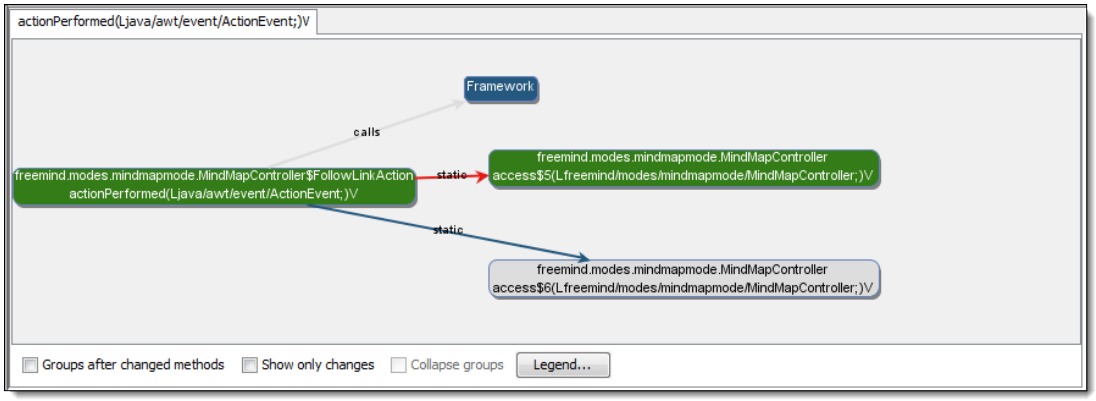

Figure 3.4: Call Graph View

În Figura 3.4 sunt afişate subgrafele de apel ce pornesc de la metodele asociate cu evenimentele elementului de interfaţă grafică afişată în Figura 3.3. Metodele aplicaţiei sunt etichetate folosind semnătura lor, iar arcele orientate semnifică direcţia apelurilor în program.

Deoarece subgraful din Figura 3.4 este un colaj al subgrafelor din versiunile aplicaţiei FreeMind din Noiembrie şi Decembrie 2000, se reutilizează codificarea prin culoare pentru a conferi informaţii adiţionale. Culorile utilizate în Figura 3.4 semnifică:

[1] http://netbeans.org/features/java/swing.html
[2] http://www.eclipse.org/windowbuilder

- *Gri* e utilizat pentru metodele şi apelurile care nu au suferit modificări între versiuni.
- *Verdele* e utilizat pentru afişarea metodelor care au suferit modificări în cod.
- *Roşul* se foloseşte pentru metodele şi apelurile şterse în noua versiune a aplicaţiei.
- *Albastrul* simbolizează metode şi apeluri noi.

Ţelul noastru este de a furniza cât mai multe informaţii posibil prin afişarea unei cantităţi minime de date. Pentru a îndeplini acest deziderat, abordarea noastră constă din abstractizarea unor date. Acest lucru se poate face utilizând bara de unelte din partea de jos a interfeţei acestei componente, aşa cum se vede şi în Figura 3.4. Această bară de unelte conţine controalele necesare ce permit gruparea metodelor care nu au suferit modificări în subgrafe, lăsând vizibile doar aspectele considerate "interesante".

3.4 jSET - Java Software Evolution Tracker

Această unealtă este primul rezultat al implementării framework-ului noastru. jSET este o unealtă de vizualizare şi analiză ce furnizează posibilităţile necesare examinării evoluţiei unui sistem software dintr-o perspectivă de sus în jos, începând cu modificările de la nivelul interfeţei grafice până la examinarea codului sursă.

Unealta jSET a fost construită folosind cele trei componente descrise mai sus şi utilizează ca date de intrare mecanismul de *Proiecte* detaliat în Subcapitolul 2.2. jSET poate fi utilizat atât pentru explorarea unui proiect, cât şi pentru compararea a două versiuni ale aceleaşi aplicaţii, detaliat în Secţiunile următoare.

3.4.1 Explorarea proiectelor

Acest mod de funcţionare este utilizat la alegerea unui singur *Proiect* în momentul lansării în execuţie a uneltei. În Figura 3.5 se vede aplicaţia în modul de explorare având ca date de intrare proiectul de reprezintă aplicaţia FreeMind, versiunea din Ianuarie 2001.

Captura de ecran din Figura 3.5 relevă configuraţia în care sunt utilizate componentele implementate pentru unealta jSET: în partea stângă a ecranului este posibilă examinarea interfeţei grafice afişate de componenta *GUI Compare View*, iar în partea

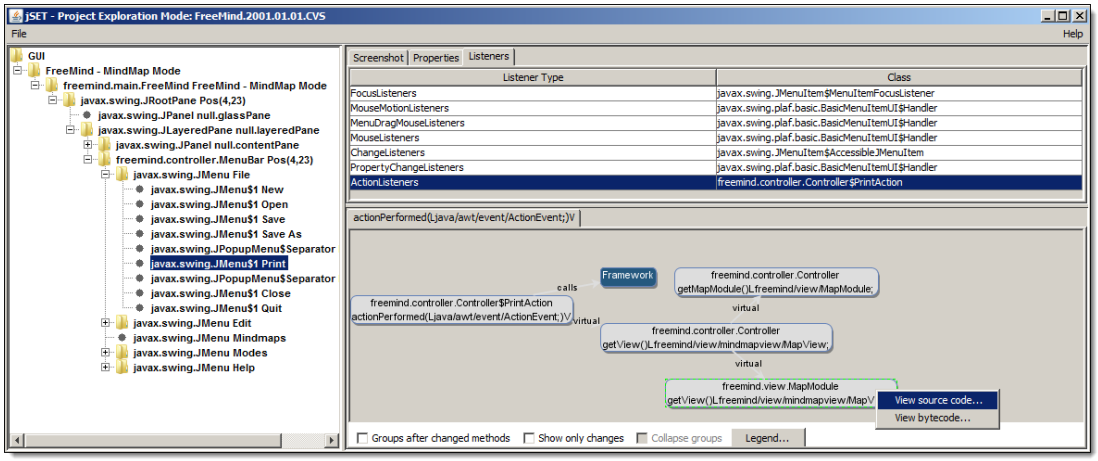

Figure 3.5: jSET în modul de explorare

dreaptă a ecranului se pot consulta informaţii referitoare la elementele grafice şi graful de apeluri asociat.

Aceste componente independente sunt controlate de o instanţă a clasei *jSETController*, care este inima uneltei. Când aplicaţia este lansată în execuţie în acest mod, ierarhia interfeţei grafice e afişată în partea stângă. Când utilizatorul alege unul din nodurile ce reprezintă elemente grafice, informaţiile relevante sunt afişate prin componenta *Widget Information View* care permite identificarea vizuală a elementului selectat în cadrul ferestrei părinte. Figura 3.5 prezintă al treilea tab al acestei componente care e utilizat pentru afişarea metodelor responsabile cu prelucrarea evenimentelor asociate elementului grafic. Alegerea unei metode actualizează subgraful de metode din partea de jos a ecranului care va afişa subgraful metodelor ce pot fi executate la interaşiunea cu elementul grafic ales.

Implementarea modului pentru explorarea de proiecte permite studierea aplicaţiei ţintă la toate nivelurile, indiferent de modul de implementare. Interfaţa grafică poate fi răsfoită şi elementele sale sunt uşor de recunoscut, fiind scoase în evidenţă în capturile de ecran oferite.

3.4.2 Compararea proiectelor

Modul de comparare a uneltei jSET este cea mai importantă contribuţie a noatră în domeniul uneltelor software din acest Capitol. Prin alegerea a două versiuni ale aceleaşi aplicaţii la lansarea în execuţie, programul va furniza informaţii despre modificările

survenite în aplicaţia ţintă pe toate straturile ei. Modificările interfeţei grafice sunt afişate folosind componenta GUI Compare View, în timp ce modificările la nivelul proprietăţilor elementelor interfeţei grafice sunt afişate în taburile componentei Widget Information View. Mai mult, componenta Callgraph View descrisă în Subcapitolul 3.3.3 prezintă schimbările la nivelul grafului de apeluri al aplicaţiei.

În plus faţă de funcţionalităţile deja detaliate, modul de comparare permite compararea codului obiect şi a codului sursă pentru metodele afişate în graful de apeluri prin utilizarea meniului contextual asociat cu nodurile ce reprezintă metodele aplicaţiei. Personalul de testare poate utiliza modul de comparare pentru a determina zonele interfeţei grafice nou implementate sau recent schimbate şi pot ajusta planurile de testare în mod corespunzător. Unealta permite utilizatorilor accesul la studierea modificărilor ce au survenit între versiunile aplicaţiei şi în spectru mai larg să urmărească evoluţia aplicaţiei ţintă de-a lungul unui număr mare de versiuni.

3.4.3 Limitări

Limitările mai importante ale uneltei jSET sunt legate de analizarea şi vizualizarea interfeţelor grafice dinamice, a elementelor grafice care interacţionează, a implementărilor de metode native şi folosirea mecanismului de introspecţie în cadrul programelor analizate. Ţelul nostru pentru următoarele versiuni este de a furniza opţiuni adiţionale pentru a controla sau elimina aceste probleme.

3.5 Concluzii

Framework-ul şi uneltele descrise în aceste pagini sunt utilizate pe tot parcursul cercetărilor noastre. Implementări adiţionale de componente şi configuraţii sunt descrise în cadrul Capitolelor următoare unde sunt cele mai relevante. Printre îmbunătăţirile posibile amintim adăugarea de suport pentru urmărirea modificărilor din mai multe versiuni şi integrarea acestui framework în cadrul unor medii de dezvoltare binecunoscute folosind mecanismul de plugin-uri.

4

Un proces euristic pentru potrivirea elementelor GUI echivalente între versiunile unei aplicații

Acest Capitol detaliază cercetările noastre în dezvoltarea unnui proces euristic de acuratețe ridicată ce permite întreținerea pe termen lung a cazurilor de testare GUI precum și a efectuării de vizualizări și analize ce țintesc modificările interfeței grafice. Cu excepția primului Subcapitol care prezintă preliminariile necesare acest capitol prezintă cercetări originale.

Acest Capitol este structurat după cum urmează: Subcapitolul 4.1 prezintă preliminariile necesare, iar în Subcapitolul 4.2 prezentăm procesul euristic pentru potrivirea elementelor interfeței grafice. În Subcapitolul 4.3 detaliem implementările euristicilor. Aceste dezvoltări originale sunt supuse testării în cadrul Subcapitolului 4.5 unde detaliem un studiu de caz complex întreprins cu scopul de a evalua acuratețea procesului propus pentru aplicațiile din repozitoriul nostru software. Mai mult, efectuăm o analiză riguroasă a claselor de erori euristice întâlnite și propunem noi direcții de cercetare pentru îmbunătățirea procesului.

Contribuțiile originale prezentate în acest Capitol sunt trecute în revistă în Capitolul introductiv și urmează a fi prezentate în cadrul conferinței MaCS 2012 (4), urmând ca versiunea extinsă să fie publicată (7).

4.1 Preliminarii

Obiectivul cercetărilor descrise în acest Capitol privește îmbunătățirea testării automate a aplicațiilor GUI prin furnizarea metodelor necesare reutilizării cazurilor de testare existente pentru mai multe versiuni ale aplicației țintă. Problema întreținerii cazurilor de testare nu a trecut neobservată și mai multe abordări au fost propuse. Un prim studiu de caz efectuat de Memon și Soffa (35) studiază întreținerea cazurilor de testare pentru două versiuni ale aplicației Adobe Acrobat Reader și a unei clone Microsoft WordPad dezvoltate intern. Cercetări ulterioare (31) propun inserarea sau ștergerea de evenimente din cazurile de testare ce devin ne-executabile pentru a le repara, în timp ce Huang et al. (20) detaliază o abordare pentru a mări acoperirea cazurilor de testare prin utilizarea de algoritmi genetici.

Cercetările noastre țintesc refolosirea cazurilor de testare GUI prin identificarea corectă a modificărilor interfeței grafice și utilizarea acestei informații pentru actualizarea cazurilor de testare pentru noile versiuni ale aplicației. Acest lucru este îndeplinit folosind un proces euristic capabil de a detecta elementele funcționale echivalente ale interfeței grafice între versiunile acesteia. Identificarea corectă a acestor modificări permite reconstruirea exactă a multor cazuri de testare, permițând testarea de regresie a aplicației țintă. Preliminariile necesare au fost furnizate de lucrarea lui McMaster și Memon (24) care furnizează următoarea definiție pentru problema potrivirii elementelor grafice echivalente funcțional:

Definition 4.1 *Fiind date două interfețe grafice* G *și* G‘*, pentru fiecare element GUI acționabil* e_i *din* G*, să se găsească un element corespunzător* e_j *din* G’ *al cărui acțiuni să implementeze aceleași funcționalități. Mai formal, fiecare element grafic din* G *și* G’ *trebuie asignat uneia din următoarele structuri de date:*

- Șterse - *Conține elementele găsite doar în GUI-ul* vechi.
- Create - *Conține elementele găsite doar în GUI-ul* nou.
- Păstrate - *Acesta este o mulțime de mapări de tipul* $(e_i \mapsto e_j)$ *ce conține perechi de elemente echivalente, cu* $e_i \in$ G *și* $e_j \in$ G’ *(Din (24)).*

4.2 Procesul

Scopul procesului nostru este categorisirea tuturor elementelor interfețelor grafice din perechea de modele supuse analizei într-unul din categoriile descrise în (24). Pentru aceasta, procesul propus funcționează în trei pași:

1. *Potrivirea ferestrelor.* În prima fază sunt găsite ferestrele echivalente. Acesta este un pas necesar pentru a permite potrivirea elementelor din cadrul ferestrelor, așa că precizia întregului proces este sensibilă la erorile din cadrul acestui pas.

2. *Potrivirea elementelor.* În cadrul acestui pas sunt potrivite elementele din ferestrele interfețelor grafice.

3. *Finalizarea.* Acesta e ultimul pas al procesului și servește ca o fază de finalizare pentru a asigura că nu rămân elemente neclasificate.

Este important de notat că procesul propus poate potrivi elemente doar în cazul în care acestea se găsesc în cadrul unor ferestre detectate ca fiind echivalente. Aceasta scoate în evidență importanța detectării ferestrelor care se potrivesc și introduce limitări privind identificarea elementelor grafice ce au fost mutate între ferestre.

Procesul este configurabil folosind un fișier XML ce permite furnizarea setului de euristici utilizate împreună cu o strategie de execuție.

Euristicile sunt furnizate sub forma unei liste prioritizate $L_p = (h_1, h_2, ..., h_n)$, ordonate astfel încât $\forall 0 < i < j \leq n$, $P_i > P_j$, unde P_i reprezintă prioritatea asociată euristicii i_{th}.

Strategia de execuție controlează ordinea în care implementările sunt utilizate și este responsabilă cu lansarea în execuție a euristicilor într-o manieră care să furnizeze acuratețe maximă. Implementarea curentă furnizează următoarele strategii care pot fi utilizate pentru a controle primele două faze ale procesului.

- *Strategie de execuție simplă.* Euristicile sunt rulate în ordine descrescătoare a priorităților asociate și fiecare implementare poate lua câte decizii dorește. După executarea ultimei implementări procesul trece la pasul următor.

- *Strategie de execuție prioritizată.* Această strategie încearcă să asigure că deciziile sunt luate de cea mai precisă euristică. Pentru asta, euristicile sunt executate

în ordine descrescătoare a priorității, procesul fiind reluat după fiecare decizie. Această strategie are efectul că deciziile luate de euristicile cu prioritate mică pot fi utilizate de implementările mai precise ăn găsirea de noi elemente echivalente.

4.3 Euristici implementate

Acest Subcapitol descrie euristicile implementate. Acestea se împart în două tipuri, depinzând de pasul în care sunt utilizate:

- *Euristici pentru ferestre.* Aceste euristici sunt utilizate în primul pas al procesului pentru a potrivi ferestrele echivalente. În mod curent avem o singură implementare descrisă în detaliu în această secțiune.
- *Euristici pentru elementele grafice.* Aceste euristici sunt utilizate în a doua etapa a procesului și pot găsi elementele grafice echivalente din cadrul ferestrelor. În cadrul acestei secțiuni vom descrie și implementările acestor euristici.

WindowMatchingHeuristic[1]

Această euristică e utilizată pentru găsirea ferestrelor echivalente. Acuratețea ei este crucială deoarece erorile în potrivirea ferestrelor se propagă în faza următoare cu consecințe grave asupra preciziei de ansamblu. Implementarea aceasta folosește titlurile ferestrelor pentru a potrivi întâi acele ferestre ce apar la pornirea aplicației, urmând ca mai apoi să examineze perechile de ferestre rămase. Dacă ambele versiuni ale interfeței grafice au o singură fereastră ea este potrivită indiferent de titlu.

PropertyValuesHeuristic[2]

Aceasta este o fabrică de instanțe capabilă de a genera euristici care potrivesc elemente echivalente examinând valoarea proprietăților asociate elementelor interfeței grafice folosind anumite criterii. Criteriile posibile sunt:

- *Egalitate.* Valorile proprietăților trebuie să fie nenule și egale.

[1] EuristicăPentruFerestre - Euristicile sunt implementate folosind clase cu acelasi nume
[2] EuristicăValoareProprietăți

- *Similaritate.* Valorile proprietăților trebuie să fie similar conform algoritmului diff (51). Un parametru număr întreg specifică numărul maxim de operații diff de *adăugare* sau *ștergere* permise ca valorile să fie considerate similare. Când acest criteriu este utilizat, numărul total de operații de *adăugare* sau *ștergere* reprezintă scorul de potrivire. Perechile de elemente grafice candidate sunt potrivite în ordinea crescătoare a scorului. Această abordare oferă flexibilitate în cazul modificării valorilor proprietăților elementelor grafice între versiuni.
- *Nulitate.* Ambele valori trebuie să fie nule.

Folosind aceste ccriterii se poate genera un număr mare de euristici, făcând din această implementare una din bazele procesului nostru. Numărul sau tipul criteriilor nu este limitat. Astfel, se poate crea o implementare ce va potrivi elemente grafice ce au valori egale asociate proprietății *Class*, valori similare pentru proprietatea *Text* și *Icon* și valoare nulă pentru proprietatea *Accelerator*.

PropertyValuesHierarchyHeuristic[1]

Aceasta este o fabrică de euristici care extinde PropertyValuesHeuristic prin generarea de implementări care caută elemente echivalente doar printre descendenții elementelor deja clasificate ca echivalente. Această implementare este datorată observației noatre ce sugerează că de multe ori descendenții elementelor echivalente sunt tot echivalente. Implementarea separată permite optimizarea codului sursă în sensul îmbunătățirii vitezei de execuție pentru interfețele grafice complexe.

SingletonComponentHeuristic[2]

Această fabrică euristică creează instanțe capabile de a potrivi elemente grafice folosind unicitatea valorii unei proprietăți date. Instanțele sunt generate prin furnizarea numelui proprietății verificate. Valoarea acestei proprietăți va fi apoi utilizată în detectarea elementelor echivalente. De exemplu, o instanță ce utilizează proprietatea *Class* va considera echivalente o pereche de elemente ce au valoarea proprietății egală cu *javax.swing.JButton* doar dacă ambele elemente sunt unicele ce au această valaore în

[1] EuristicăIerarhicăValoareProprietăți
[2] EuristicăComponenteSingleton

cadrul ferestrei pe care se află. Numele de *Singleton* trimite către acest comportament, căci fiecare element trebui să fie cumva unic pe fereastra sa.

InverseHierarchyHeuristic[1]

Această implementare este unica euristică neconfigurabilă din arsenalul nostru. Ea se dovedește utilă în potrivirea elementelor container care au fost supuse unor modificări majore și nu au fost recunoscute de alte implementări. Fiind dată componenta $A \in GUI_{vechi}$ și componenta $B \in GUI_{nou}$, această euristică potrivește pe A cu B dacă:

- Toți descendenții lui A au o componentă echivalentă care e descendent al lui B.
- A are același număr de descendenți ca și B.

FinalHeuristic[2]

Această euristică a fost implementată pentru a gestiona ultima fază a procesului. Rolul său este de a atribui toate elementele grafice care au rămas neclasificate în unul din mulțimile *Ștearsă* sau *Creată*, depinzând de modelul GUI de care aparțin elementele. Această ultimă fază a procesului nu poate fi configurată și este implementată pentru a asigura categorisirea tuturor elementelor interfețelor grafice.

4.4 Metrici euristice

În această Secțiune definim metrici utilizabile în caz general pentru a evalua acuratețea unui proces de găsire a elementelor grafice echivalente. Începem prin a defini câteva metrici ce pot fi determinate utilizând doar informația disponibilă prin oracol:

Definition 4.2 Correct Decision Count (CDC)[3]. *Numărul de decizii corecte pentru o pereche de versiun dată. Definim o decizie ca acțiunea de a atribui un element grafic uneia din structurile* Ștearsă *sau* Creată*, sau a unei perechi echivalente în* Păstrate*.*

Definition 4.3 Correct Match Count (CMC)[4]. *Numărul de elemente din mulțimea* Păstrate*. Aceasta reprezintă numărul de elemente grafice echivalente funcțional pentru cele două interfețe grafice.*

[1] EuristicăIerarhicăInversă
[2] EuristicăFinală
[3] Număr Corect Decizii
[4] Număr Corect Potriviri

Definition 4.4 Dissimilar Widget Count (DWC)[1]. *Numărul de elemente grafice modificate. Aceasta include toate elementele din mulțimile* Create *și* Șterse *precum și numărul de elemente echivalente unde cel puțin una din proprietățile care nu se referă la mărimea sau localizarea pe ecran a componentei și-a modificat valoarea.*

După rularea procesului algoritmii noștri de evaluare analizează rezultatele obținute și calculează valori pentru următoarele metrici:

Definition 4.5 Heuristic Correct Decision Count (HCDC)[2]. *Reprezintă numărul deciziilor corecte luate de proces. Acesta e un număr între 0 și valoarea CDC.*

Definition 4.6 Heuristic Correct Match Count (HCMC)[3]. *Reprezintă numărul de potriviri corect detectate. Acesta este numărul de elemente corecte din mulțimea mapărilor* Păstrate *calculat de euristică. Este un număr între 0 și valoarea CMC.*

Definition 4.7 Heuristic Correct Decision in Dissimilar Widgets Count (HCDDWC)[4]. *Reprezintă numărul de decizii corecte ce privesc elementele nesimilare. Aceasta este o metrică similară cu* HCDC *dar ia în considerare doar elementele nesimilare. Ea poate lua valori între 0 și valoarea DWC.*

Pentru a evalua mai bine acuratețea procesului independent de complexitatea interfeței grafice am definit următoarele măsurători:

Definition 4.8 Heuristic Decision Rate (HDR)[5]. *Reprezintă procentajul deciziilor corecte luate, calculat ca* $\frac{HCDC}{CDC}$.

Definition 4.9 Heuristic Match Rate (HMR)[6]. *Reprezintă procentajul de potriviri corecte, calculat ca* $\frac{HCMC}{CMC}$.

Definition 4.10 Heuristic Dissimilar Widgets Decision Rate (HDWDR)[7]. *Reprezintă procentajul deciziilor corecte pentru elementele grafice nesimilare, calculat ca* $\frac{HCDDWC}{DWC}$.

Decizia definirii acestor metrici separate și neefectuarea unei analize clasice fals pozitiv/negative a fost luată din cauza aspectelor particulare ale procesului nostru care credem că este mai bine caracterizat prin valorile acestor metrici.

[1] Număr Elemente Nesimilare
[2] Număr Corect de Decizii Euristice
[3] Număr Corect de Potriviri Euristice
[4] Număr Corect de Decizii Euristice Privind Elemente Nesimilare
[5] Rata de decizie euristică
[6] Rata de potrivire euristică
[7] Rata de decizie euristică pentru elemente nesimilare

4.5 Studiu de caz

Această secțiune prezintă un studiu de caz complex ce țintește să evalueze acuratețea și fezabilitatea procesului euristic în utilizarea sa pentru aplicații GUI complexe. Examinăm rezultatele obținute prin executarea procesului pentru a răspunde la următoarele întrebări:

1. Care e configurația optimă a procesului euristic?
2. Care e precizia procesului când acesta este utilizat pentru aplicații complexe GUI în scopul întreținerii pe termen lung a cazurilor de testare și a oferirii de vizualizări software?
3. Care e tipul erorilor euristice ce pot fi întâlnite și cum le putem limita numărul?

4.5.1 O configurație euristică de mare precizie

Mai multe experimente au fost întreprinse pentru a răspunde la întrebarea (1). Aceasta cere găsirea unei configurații euristiuce optimale împreună cu un set euristic de mare precizie. Însă din cauza diversității de implementare a euristicilor, precum și a dicersității aplicațiilor GUI am descoperit că nu se poate construi o configurație euristică "perfectă". Astfel am schimbat abordarea și ne-am concentrat pe construcția unei configurații precise cu ajutorul căreia să obținem un echilibru optim între precizie, generalitate și viteza execuției.

Primul pas în construirea setului euristic a fost utilizarea unei abordări combinatoriale pentru a genera euristici bazate pe valoarea proprietăților din Tabela 4.1.Pentru a genera implementările am rulat două bucle peste lista de proprietăți. Bucla exterioară controlează număul de proprietăți ignorate (n_{dp}) și ia valori între 0 și numărul de proprietăți minus 2. A doua buclă controlează care sunt proprietățile ignorate, traversând tabelul de jos în sus. La fiecare pas al buclei interioare o nouă euristică este generată.

Pentru a obține versiunea finală a setului euristic propus am adăugat implementări ale euristicilor *SingletonComponentHeuristic* și *InverseHierarchyHeuristic* și am înlăturat implementările imprecise. Toate aceste artefacte sunt disponibile pe site-ul nostru (37).

Precizie	Proprietate
Foarte precisă	Icon Class Text Accelerator
Precisă	Index
Imprecisă	Width Height X Y

Table 4.1: Precizia proprietăților înregistrate

4.5.2 Rezultatele obținute

Această secțiune caută răspunsul la întrebarea (2) prin prezentarea rezultatelor obținute în aplicarea setului euristic dezvoltat în secțiunea anterioară celor 28 de perechi de versiuni ale FreeMind și jEdit din repozitoriul nostru. Tabela 4.2 prezintă rezultatele agregate obținute de euristicile descrise în secțiunea anterioară pentru cele 28 de versiuni studiate. Trebuie să notăm că rezultatele prezentate au fost obținute prin eliminarea subcomponentelor elementelor complexe și ignorarea elementelor de delimitare.

Metrica	FreeMind	jEdit	Total
Correct Decision Count	1799	8976	10775
Correct Match Count	1524	7461	8985
Dissimilar Widget Count	797	4115	4912
Heuristic Decision Count	1787	8953	10740
Heuristic Match Count	1505	7321	8826
Heuristic Correct Decision Count	1743	8436	10179
Heuristic Correct Match Count	1502	7194	8696
Heuristic Decision Rate	96.89%	93.98%	94.47%
Heuristic Match Rate	98.56%	96.42%	96.78%
Heuristic Dissimilar Widget Decision Rate	89.46%	80.46%	81.92%

Table 4.2: Rezultatele procesului euristic

Cele mai importante rezultate se găsesc în rândurile subliniate. Luând în calcul intervalul de timp studiat (7 ani în cazul FreeMind și 10 în cazul jEdit) considerăm ratele de decizie de peste 90% ca fiind foarte promițătoare. Credem că rate de potriviri corecte de peste 95% permit implementarea mecanismelor de întreținere pe termen lung a cazurilor de testare pentru aplicațiile cu interfețe grafice. De asemenea, precizia ridicată în luarea de decizii privind elementele disimilare arată că procesul este fezabil în potrivirea elementelor grafice supuse schimbărilor din cadrul aplicațiilor care evoluează rapid.

4.5.3 Analiza erorilor euristice

Această secțiune își dorește să răspundă la întrebarea (3) prin studierea tipurilor de erori întâlnite în studiul de caz efectuat.

Detectarea schimbărilor multiple

Așa cum am așteptat, găsirea elementelor grafice echivalente devine dificilă în momentul în care acestea sunt supuse mai multor schimbări. În Figura 4.1 observăm meniul *File* în cazul a două versiuni ale aplicației FreeMind. Din cauza schimbărilor multiple suferite de elementul grafic evidențiat, el nu este clasificat în mod corect ca fiind Persistat între versiunile studiate.

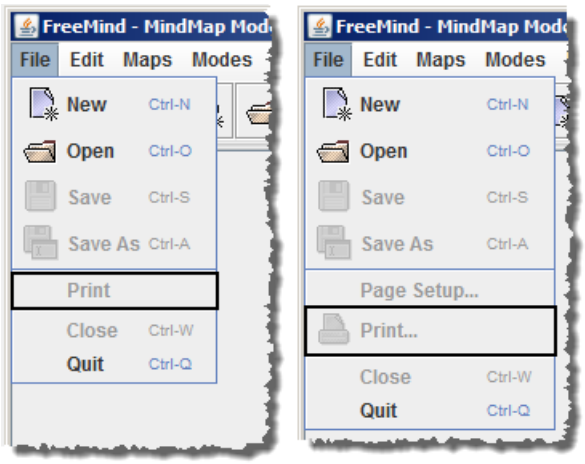

Figure 4.1: Pereche de elemente echivalente care nu au fost detectate corect

Detectarea modificărilor în cazul elementelor complexe

Una din conluziile studiului de caz efectuat este că avem nevoie de mai multe informații pentru a detecta elementele complexe echivalente ale interfețelor grafice. Figura 4.2 furnizează un asemenea exemplu folosind o perche de versiuni a aplicației FreeMind. Problema din figură constă în recunoașterea greșită a elementelor combo-box din cauza lipsei informațiilor suplimentare ce țin de modelul de date al controalelor.

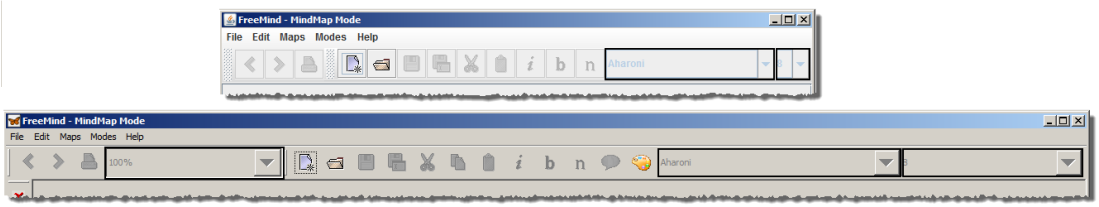

Figure 4.2: Controale combo-box detectate eronat

Modificări în spatele stratului GUI

Modificările din cadrul aplicației FreeMind între Ianuarie și Februarie 2004 au adus o schimbare în meniul *Edit* al aplicației care nu putea fi detectată exclusiv prin analiza interfeței grafice. Aceasta s- datorat unor modificări atât la nivelul interfeței grafice precum și la nivelul codului sursă asociat care a necesitat analizarea codului pentru a stabili perechile de elemente grafice echivalente. În Figura 4.3 găsim elementele grafice în cauză cu decizia eronată evidențiată.

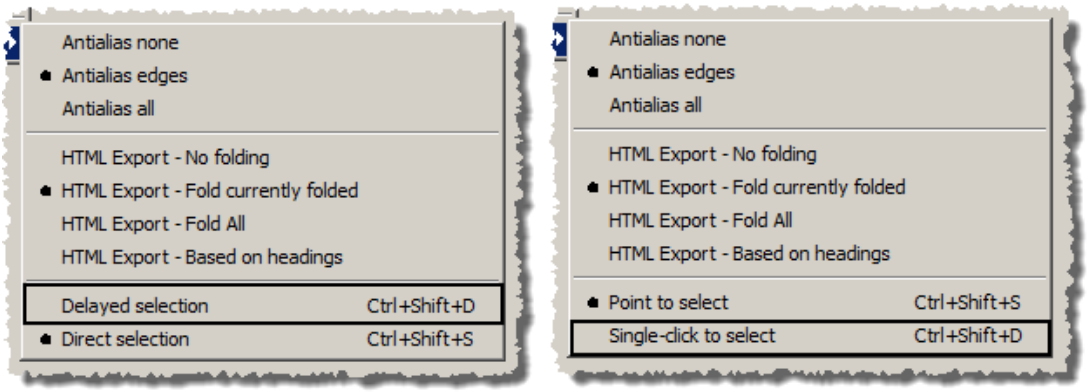

Figure 4.3: Elemente ce nu sunt echivalente deși au același Accelerator

Chiar dacă sunt rare, acest tip de eroare împiedică construcția unui set "perfect" de euristici din cauza lipsei tehnicilor de analiză a codului capabile de a detecta astfel de modificări.

Modificări a fișierelor de iconițe

Una din punctele slabe ale procesului propus privește modul de lucru cu valoarea proprietății *Icon*. Implementarea curentă utilizează numele fișierului pentru a lua deciziile

euristice. Acest lucru prezintă avantajul că deciziile rămân consistente odată cu schimbarea conținutului fișierului, dar că modificările de nume ale acestuia pot cauza apariția de erori euristice. Figura 4.4 prezintă un astfel de exemplu utilizând butonul "Print" din cadrul a două versiuni FreeMind.

Figure 4.4: Modificările iconițelor pot duce la erori euristice

Schimbări în tipul elementelor grafice

În Figura 4.5 se pot observa două grupuri de elemente grafice evidențiate care nu au fost detectate ca fiind echivalente din cauza modificării tipului lor.

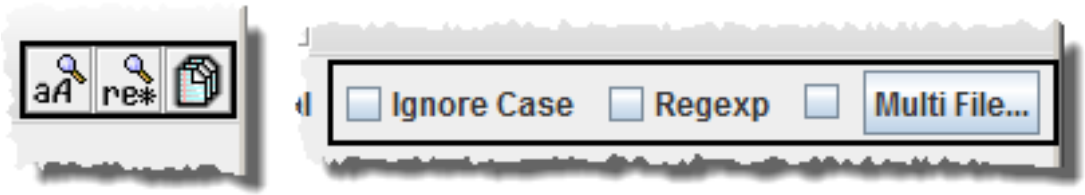

Figure 4.5: Modificările în tipul implementării pot duce la erori

4.5.4 Riscuri asupra validității studiului de caz

Deși am depus toate eforturile pentru a elimina riscurile ce pot afecta validitatea cercetărilor noastre unele aspecte au nevoie de detaliere. În primul rând, natura automatizată a procesului face ce existența defectelor software să poată afecta rezultatele obținute. Pentru a elimina această posibilitate am implementat multiple verificări software care analizează fiecare pas al procesului. Un alt aspect se referă la generalitate. Ambele aplcații studiate sunt complexe și au fost utilizate în studii de caz anterioare (21, 55, 56, 57). Deoarece ele reprezintă doar o mică parte a tuturor aplicațiilor GUI există riscul ca ele să nu fie reprezentative pentru toate aplicațiile.

4.6 Limitări curente

Deși procesul nostru a fost gândit pentru a fi flexibil, am identificat unele limitări ce pot împiedica utilizarea sa în anumite circumstanțe. Unele din aspectele identificate se datorează uneltelor utilizate, iar altele pot fi rezolvate prin depunerea de eforturi viitoare. Cele mai importante limitări țin de analizarea interfețelor grafice dinamice, implementări de controale particulare și analizarea perechilor de interfețe grafice ce conțin schimbări majore.

4.7 Concluzii și cercetări viitoare

O direcție viitoare de cercetare constă din includerea unor aplicații .NET și SWT în cadrul unui studiu de caz mai extins. Dorim să evaluăm procesul euristic dezvoltat folosind și alte aplicații pentru a strânge mai multe date care să arate cum se modifică acuratețea procesului propus în utilizarea de build-uri zilnice sau săptămânale. De asemenea, un astfel de studiu ar releva modul în care implementări particulare de controale afectează precizia procesului.

5

Managementul testării GUI

Acest Capitol prezintă contribuţiile noatre originale privind testarea şi vizualizarea aplicaţiilor cu interfeţe grafice. Subcapitolul 5.1 prezintă unealta GUI Test Suite Manager construită folosind componentele din framework-ul nostru detaliat în Capitolul 3. În Secțiunea 5.2 prezentăm rezultatele unui studiu de caz ce evaluează posibilitatea de a repara cazurile de testare a interfeței grafice utilizând cercetările prezentate în Capitolul 4. La final unim direțiile de cercetare privind vizualizarea și testarea aplicațiilor GUI într-un tot unitar prin descrierea unui proces de testare a regresiilor utilizabil în cazul aplicațiilor GUI.

Contribuțiile noastre originale din acest Capitol urmează a fi trimise spre publicare (8) și sunt trecute în cadrul Capitolului introductiv al acestei teze.

5.1 O unealtă software pentru managementul cazurilor de testare GUI

Această secțiune detaliază contribuțiile noastre originale în domeniul gestionării cazurilor de testare GUI. Introducem noi componente ale framework-ului nostru software descris în Capitolul 3 și prezentăm o nouă unealtă software numită GUI Test Suite Manager ce integrează eforturile noastre prezentate în cadrul capitolelor anterioare prin furnizarea unei noi abordări în administrarea pe termen lung a cazurilor de testare.

5.1.1 Măsurarea acoperirii testelor

Metrici precum acoperirea liniilor sau a bifurcațiilor codului sunt utilizate de mult timp în industrie și literatura de specialitate abundă cu evaluări ale avantajelor și dezavantejelor acestor abordări. În schimb, aplicațiile GUI sunt mai bine exprimate folosind evenimentele generate decât codul sursă, astfel că o nouă direcție de cercetare constă în definirea de noi criterii de acoperire bazate pe evenimente. Memon et al. au prezentat criterii importante pentru măsurarea calității testelor precum *acoperirea evenimentelor*, *acoperirea interațiunilor între evenimente* și generalizarea acestora *acoperirea secvențelor de lungime-n*, descrise în (36). De asemenea ei au legat aceste noi metrici cu criteriile "tradiționale" bazate pe codul sursă utilizând un studiu de caz ce scoate în evidență cum acoperirea secvențelor de evenimente de lungime 2 și 3 duce la o acoperire bună a codului sursă.

În acest Subcapitol propunem o nouă abordare ce combină metricile de acoperire "tradiționale" a codului cu informatții obținute utilizând analiza statică prin folosirea framework-ului Soot în contextul definiției lui Memon a cazurilor de testare GUI (29). Începem cu furnizarea următoarelor definiții ce constituie baza implementării noastre:

Definition 5.1 Subgraf static de apel al evenimentului. *Dându-se un eveniment GUI e_i, definim subgraful static de apel al său acel subgraf al grafului de apeluri al programului care conține toate metodele aplicației executate la declanșarea evenimentului împreună cu toate metodele aplicației ce pot fi apelate în mod tranzitiv.*

Definition 5.2 Acoperirea statică a pasului. *Fiind dat un caz de testare GUI T cu secvența de evenimente $\{e_1, e_2, ..., e_n\}$, definim* acoperirea statică a pasului *asociat cu evenimentul e_i, unde $0 < i \leq n$ ca raportul dintre enunțurile acoperite din codul sursă supra toate enunțurile din subgraful static de apel al evenimentului asociat pasului de testare.*

Acoperirea statică a pasului e definită pentru a furniza o măsură a cât de bine este codul ce s-ar putea executa acoperit de pasul de testare. Fără a avea la dispoziție unelte pentru analiza statică am putea utiliza doar unelte de instrumentare a codului pentru a măsura acoperirea fiecărui pas al cazului de testare fără a avea însă informații privind codul care *ar fi putut* fi rulat. Folosind aplicații precum Soot devine posibilă furnizarea acestor așteptări apriori fără executarea aplicației.

Definition 5.3 Acoperirea statică inclusivă a pasului de testare. *Fiind dat un caz de testare GUI T având secvența de evenimente $\{e_1, e_2, ..., e_n\}$, definim* acoperirea statică inclusivă a pasului de testare *a evenimentului e_i, având $0 < i \leq n$ ca raportul dintre enunțurile acoperite din codul sursă din mulțimea metodelor obținută prin reuniunea tuturor metodelor din subgrafele statice de apel al evenimentelor asociate cu pașii $E_{set} = \{e_1, e_2, ..., e_i\}$, considerând acoperirea maximă pentru fiecare din metode.*

Definition 5.4 Acoperirea statică a cazului de testare. *Fiind dat un caz de testare GUI T având secvența de evenimente $\{e_1, e_2, ..., e_n\}$, definim* acoperirea statică *a T ca acoperirea statică inclusivă a pasului asociat cu evenimentul e_n.*

5.1.2 Componenta Test Suite Manager View

Acest Subcapitol detaliază componenta *Test Suite Manager View*, care a fost implementată pentru a permite administrarea cazurilor de testare GUI create cu ajutorul framework-ului GUITAR. Scopul princial al acestei componente este de a permite lansarea în execuție și furnizarea de feedback privind acoperirea statică pentru cazurile de testare GUITAR. Componenta noastră încarcă suita de teste împreună cu informațiile despre acoperirea codului pentru cazurile de testare deja rulate și le procesează pentru a obține date de ansamblu privind procesul de testare. În Figura 5.1 se observă această componentă în cadrul aplicației *GUI Test Suite Manager* descrisă în Secțiunea următoare.

5.1.3 Componenta Code Coverage View

Această componentă este o extensie a componentei *Call Graph View*, detaliată în cadrul Subcapitolului 3.3.3. Această componentă a fost implementată pentru a furniza informații detaliate privind acoperirea statică a codului pentru fiecare pas al cazului de testare.

În Figura 5.1 se poate vedea această componentă în cadrul uneltei *GUI Test Suite Manager*. Această componentă poate afișa subgraful static de apeluri al evenimentelor ce compun cazul de testare și este astfel util pentru a examina acoperirea statică a pașilor cazului de testare. Culorile sunt utilizate pentru a oferi informații privind acoperirea de cod statică atinsă. Verdele închis e utilizat pentru porțiunea de cod acoperită efectiv în cadrul pasului, în timp ce verdele (fără a ține cont de nuanță) denotă codul acoperit în cadrul cazului de testare.

5.1.4 Unealta GUI Test Suite Manager

Această unealtă software a fost implementată pentru a facilita administrarea procesului de execuție și examinare a cazurilot de testare a aplicațiilor GUI.

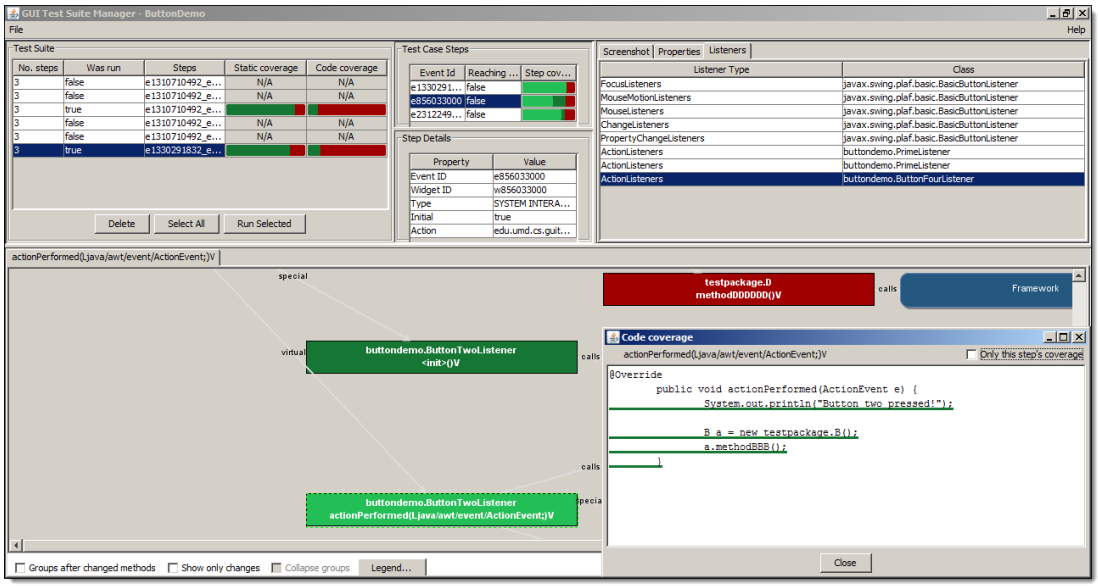

Figure 5.1: Unealta GUI Test Suite Manager

Interfața cu utilizatorul a uneltei este vizibilă în Figura 5.1 unde cititorul poate recunoaște cele trei componente care o compun. Similar cu unelta jSET, aplicația GUI Test Manager utilizează mecanismul de proiecte descris în Capitolul 2. Componenta Widget Information View este reutilizată pentru a furniza informații despre elementele GUI. De fiecare dată când utilizatorul alege un pas al cazului de testare, proprietățile și captura de ecran asociate sunt afișate pentru a putea fi examinate. De asemenea, componenta Code Coverage View afișează informații despre acoperirea pașilor de testare deja executați, permițând o examinare în detaliu a acoperirii fiecărui pas de testare precum și a acoperirii inclusive obținute.

5.2 Un studiu privind reluarea cazurilor de testare GUI

În acest Subcapitol vom prezenta un studiu de caz ce examinează reluarea cazurilor de testare GUI existente pe versiuni noi ale aplicațiilor țintă. Pentru aceasta vom reutiliza aplicațiile prezentate în cadrul Capitolului 2 al acestei lucrări. Sintetizăm

cercetarea noastră prin următoarele întrebări: (1) Cum sunt afectate cazurile de testare de modificările specifice evoluției unei aplicații GUI? (2) Cât de eficient este procesul nostru euristic în păstrarea executabilității cazurilor de testare pentru aplicațiile GUI?

Acest Subcapitol este împărțit în două Secțiuni. Prima Secțiune prezintă un "caz ideal" ce privește utilizarea de informații complete și corecte într-o evaluare a numărului de cazuri de testare ce rămân utilizabile în versiuni modificate ale aplicației țintă. A doua Secțiune studiază eficacitatea procesului propus în cadrul Capitolului 4 în păstrarea cazurilor de testare executabile prin repararea acestora pentru versiuni noi, modificate ale aplicației țintă.

Vom utiliza generatorul de cazuri de testare din cadrul GUITAR pentru a obține cazuri de testare pentru cele 28 de versiuni ale aplicaţţilor FreeMind și jEdit din repozitoriul nostru. Vom simula apoi execuția acestor cazuri de testare folosind modelele GUI și EFG disponibile pentru a evalua dacă ele rămân utilizabile în cazul evoluției aplicațiilor țintă. Deoarece studii anterioare au stabilit legături puternice între lungimea și acuratețea cazurilor de testare (Xie și Memon (54)) am decis să generăm toate cazurile de testare de lungime 2, împreună cu câte 5000 de cazuri de testare de lungime 3 și respectiv 4. Astfel am obținut un număr total de 451062 de cazuri de testare pentru toate versiunile aplicațiilor.

5.2.1 Utilizând informații complete și corecte

Această secțiune răspunde la întrebarea (1). În cadrul ei examinăm o situație ideală unde avem disponibile informații corecte despre elementele GUI echivalente funcțional. În cazul nostru este vorba de informația de oracol construită pentru studiul de caz din cadrul Capitolului 4. Țelul nostru este de a studia dacă cazurile de testare GUI pot fi reutilizate odată cu evoluția aplicației țintă. Pentru aceasta vom împărți cazurile de testare în patru categorii:

1. *Reutilizabile folosind Id-ul.* Aceasta simulează modul de funcționare al unor unelte mai puțin sofisticate ce folosesc Id-uri pentru a identifica elementele interfeței grafice.

2. *Reutilizabile folosind un proces euristic.* Această categorie reprezintă cazurile de testare ce pot fi executate pe noua versiune a aplicației țintă fără modificări ai pașilor de testare. Aceasta înseamnă că fiecare element grafic testat trebuie să

aibă corespondent în noua versiune iar secvența de pași trebuie să fie validă în noua implementare.

3. *Reparabile folosind un proces euristic.* Aici relaxăm cerința ca secvența de pași să rămână validă și cerem doar existența elemetenlor grafice echivalente în noua versiune a aplicației.

4. *Nereparabile.* Această ultimă categorie conține cazurile de testare ce nu pot fi reparate.

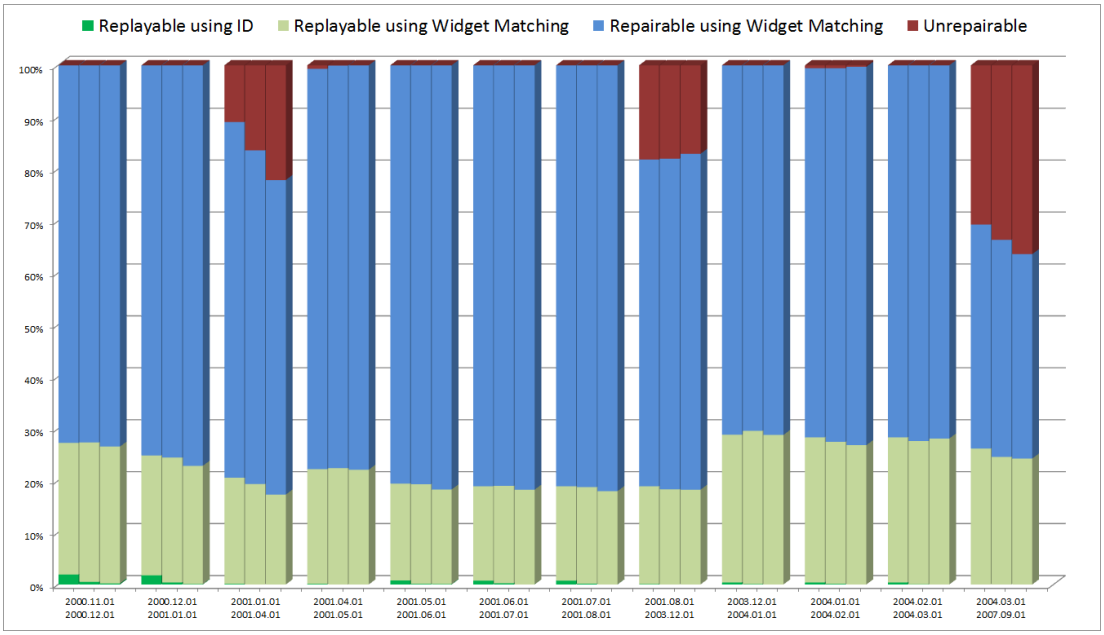

Figure 5.2: Cazurile de testare pentru FreeMind folosind informații complete și corecte

În Figura 5.2 se pot studia rezultatele obținute pentru aplicația FreeMind. Fiecare pereche de versiuni e reprezentată folosind 3 coloane. Începând din partea stângă, ele afișează informații privind categoria cazurilor de testare de lungimi 2,3 și respectiv 4. Observăm că având la dispoziție informații complete și corecte duce la posibilitatea de reparare a majorității cazurilor de testare.

În Figura 5.3 putem observa rezultatele obținute pentru aplicația jEdit. Diferența principală o reprezintă faptul că multe cazuri de testare devin nereparabile între versiunile studiate. Chiar dacă intervalul de timp între versiunile jEdit este comparabil cu

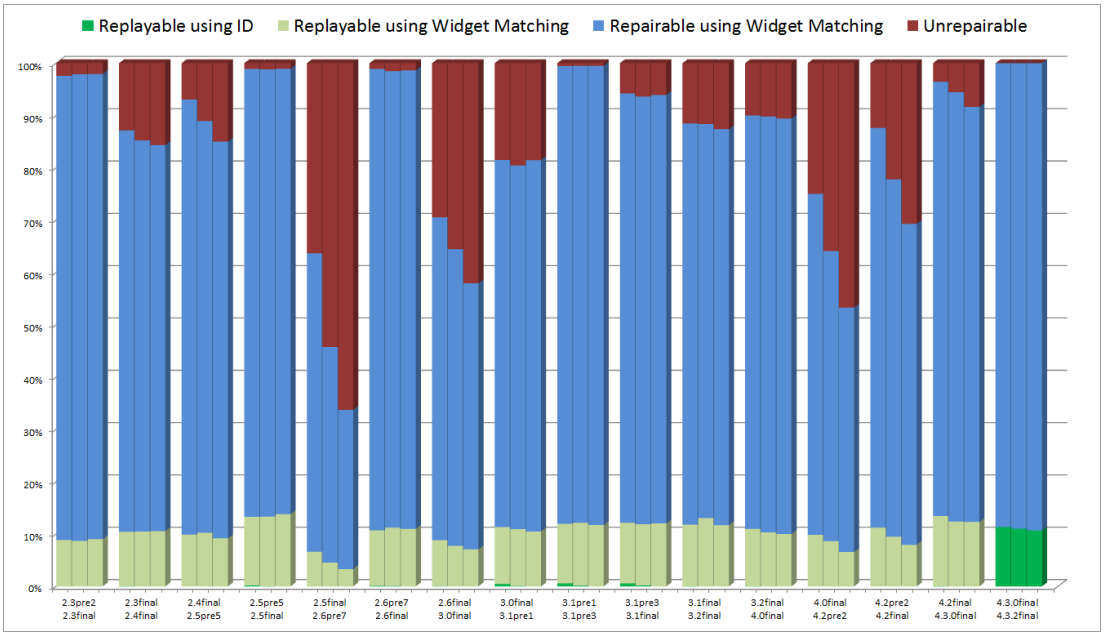

Figure 5.3: Cazurile de testare pentru jEdit folosind informații complete și corecte

cel din cazul FreeMind, complexitatea mărită a aplicației duce la aceste rezultate mai puțin încurajatoare.

5.2.2 Folosind procesul euristic

În această Secțiune vom repeta experimentul prezentat mai sus, dar de această dată folosind informațiile obținute prin utilizarea procesului euristic propus în cadrul Capitolului 4. Astfel vom da un răspuns întrebării (2).

În Figura 5.4 prezentăm rezultatele obținute prin folosirea procesului euristic în cazul aplicației FreeMind. Conform așteptărilor, rezultatele sunt asemănătoare cu cele obținute în Secțiunea anterioară. Deoarece procesul euristic are o acuratețe de 98.56% în cazul acestei aplicații ea este deja aproape de situația ideală prezentată anterior.

Figura 5.5 prezintă aceleași informații pentru aplicația jEdit. Din cauza complexității mărite a aplicației și a preciziei mai slabe de 96.42% observăm că aceste rezultate sunt semnificativ mai slabe față de cele prezentate anterior. Este de notat cazul celor patru versiuni în care majoritatea cazurilor de testare devin nereparabile.

De asemenea este interesant de notat efectul pe care lungimea cazurilor de testare îl are asupra posibilității de a le reutiliza, căci cazurile de testare ce constau din mai

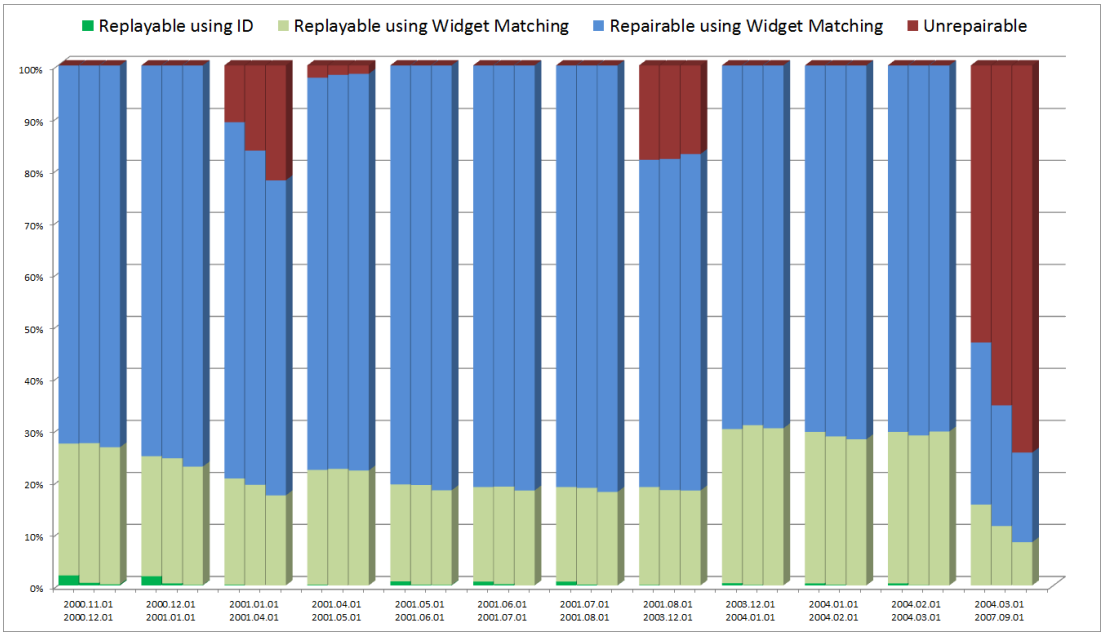

Figure 5.4: Cazurile de testare pentru FreeMind folosind procesul euristic

mulți pași sunt mai susceptibile de a scoate la iveală defectele aplicației. Totuși, luând în calcul că datele prezentate reprezintă un interval lung de timp (7 ani în cazul FreeMind, 10 în cazul jEdit) credem că abordarea noastră este utilă și poate îmbunătăți semnificativ procesul de testare a aplicațiilor GUI.

5.2.3 Riscuri asupra validității studiului de caz

Un prim risc e reprezentat de aplicațiile alese. Deși ele au fost utilizate în multiple studii de caz, nu putem generaliza rezultatele obținute pentru toate aplicațiile GUI. Alte aplicații pot prezenta provocări punctuale care nu au fost adresate în cadrul acestui studiu.

Un alt risc e reprezentat de procesul obținut în obținerea datelor. Din cauza numărului mare al cazurilor de testare ele nu au fost executate efectiv ci doar simulate folosind algoritmi proprii. Deși modulul Test Replayer din cadrul GUITAR utilizeaza algoritmi asemănători în rularea cazurilor de testare, există riscul ca unele erori sau situații specifice aplicațiilor studiate să nu permită executarea acestor cazuri de testare.

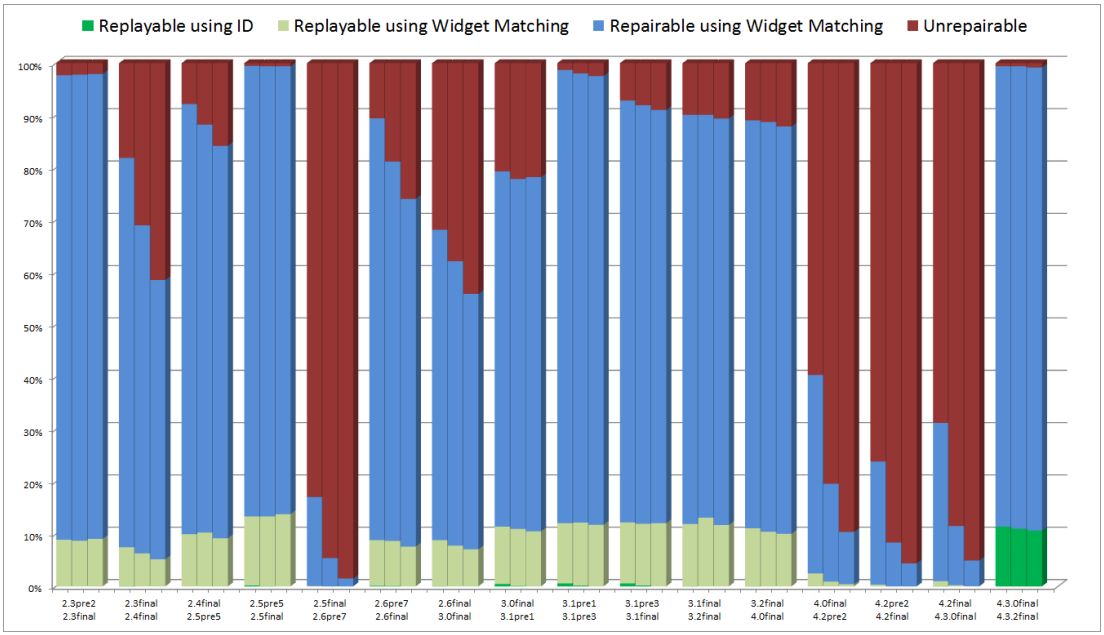

Figure 5.5: Cazurile de testare pentru jEdit folosind procesul euristic

5.2.4 Limitări curente

Aspecte ce pot fi îmbunătățite prin eforturi viitoare privest implementarea unor metrici de acoperire centrate pe evenimente (54), o mai bună evaluare a rezultatelor cazurilor de testare și dezvoltarea unei abordări semi-automate pentru construirea cazurilor de testare.

5.3 Integrarea într-un mediu de producție

Acest Subcapitol este dedicat prezentării unei metodologii de integrare a cercetărilor noastre în dezvoltarea unei aplicații GUI cu scopul de a permite testarea automată de regresie a acesteia. În Figura 5.6 se pot observa pașii procesului propus.

Procesul propus funcționează în următorii cinci pași:

1. *Build zilnic.* Acest pas demareaza procesul propus.
2. *Crearea proiectului.* În Capitolul 2 am prezentat sistemul de proiecte utilizat pentru a construi repozitoriul de software împreună cu metodele de automatizare ce permit construirea automată de proiecte pentru fiecare build al aplicației.

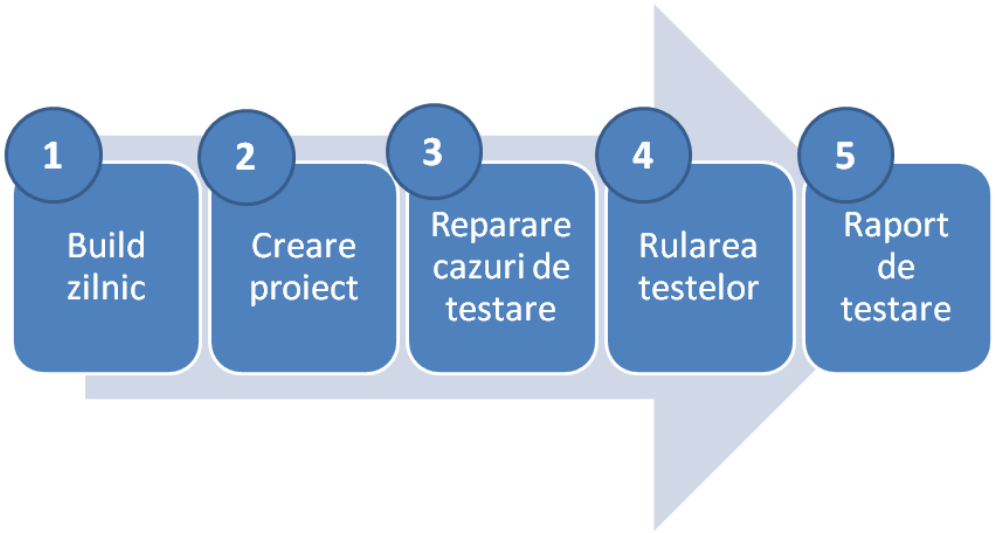

Figure 5.6: Proces de testare a regresiilor

3. *Repararea cazurilor de testare.* Acest pas constă din rularea implementării procesului noastru euristic pentru a repara cazurile de testare existente și a le adapta noii versiuni a interfeței grafice.

4. *Rularea cazurilor de testare.* Acest pas folosește componenta Test Replayer din cadrul GUITAR pentru a rula cazurile de testare reparate pe noua versiune a aplicației și a înregistra starea interfeței grafice.

5. *Raportul de testare.* Uneltele noastre oferă informații importante privind acoperirea cazurilor și pașilor de testare. Aceasta face procesul fezabil pentru descoperirea zonelor de cod care nu au fost acoperite și pentru crearea de noi teste care să rezolve problemele astfel depistate.

Procesul propus nu este primul de acest tip. Eforturi similare găsim în teza lui Memon (29) unde un proces de testare a regresiilo este propus. Abordarea va fi rafinată în (32) prin introducerea procesului numit DART. Comparând aceste abordări cu procesul propus de noi descoperim că acestea nu furnizează un sistem integrat de proiecte care să permită vizualizarea și analizarea aplicațiilor țintă. De asemenea, ele nu furnizează un mecanism pentru repararea cazurilot de testare. bazându-se pe numele constant al elementelor interfeței grafice pentru acest lucru.

O abordare de dată recentă o găsim în teza lui Xie (52), unde ea propune un proces continuu pentru testarea GUI ce are o importantă componentă pentru testarea de regresie ce utilizează framework-ul GUITAR. Opinia noastră este că abordări precum cea din (52) sunt integrabile cu procesul prezentat de noi și pot fi utilizate în testarea automată a aplicațiilor GUI.

5.4 Concluzii și cercetări viitoare

Acest Capitol a furnizat o abordare holistică a cercetărilor noastre privitoare la vizualizarea și testarea aplicațiilor GUI. Am prezentat o nouă instanțiere a framework-ului nostru de componente și am detaliat noile vizualizări oferite. Am continuat munca din Capitolul 4 și am studiat eficiența procesului propus în repararea cazurilor de testare pentru noi versiuni ale aplicațiilor țintă.

Provind eforturile viitoare, dorința noastră este de a studia cum uneltele dezvoltate pot fi integrate într-un mediu de testare continuă precum cel prezentat de Porter et al. (39). De asemenea, considerăm că eforturi noi trebuie depuse pentru construirea de noi unelte care să permită generarea de cazuri de testare ghidate de utilizator. Astfel de unelte vor permite crearea de suite de testare eficiente care vor îmbina experiența umană cu capacitățile automate de generare și execuție a unui număr mare de cazuri de testare.

6

Concluzii

Cercetarea noastră a ţintit două zone. Prima este vizualizarea aplicaţiilor cu interfeţe grafice cu accentul pe dezvoltarea de noi metode pentru analiza şi vizualizarea acestor aplicaţii. Capitolul 3 a introdus framework-ul nostru de componente ce constă din mai multe implementări ce sprijină dezvoltarea de unelte software. Prima asemenea unealtă descrisă a fost jSET, care propune o abordare holistică a vizualizării programelor prin încorporarea de unelte de analiză la fiecare nivel al aplicaţiei ţintă şi anume la nivelul interfeţei grafice, a relaţiilor de apel dintre metode precum şi la nivelul codului sursă. O altă implementare ce utilizează framework-ul nostru a fost detaliată in Capitolul 5: aplicaţia GUI Test Suite Manager permite administrarea cazurilor de testare a interfeţei grafice şi furnizează capabilităţi avansate pentru examinarea execuţiei cazurilor de testare.

A doua direcţie de cercetare abordată a vizat testarea regresiilor aplicaţiilor cu interfeţe grafice. În Capitolul 4 am introdus un nou proces euristic pentru potrivirea elementelor grafice echivalente bazat pe lucrarea lui McMaster şi Memon (24). De asemenea am întreprins un studiu de caz extensiv pentru a evalua acurateţea procesului propus. La fel de important, am efectuat o analiză amănunţită a claselor de erori euristice întâlnite şi am propus soluţii pentru astfel de situaţii.

Direcţiile noastre de cercetare s-au unit in Capitolul 5, unde am studiat eficienţa procesului nostru euristic în repararea cazurilor de testare pentru interfeţele grafice a unor aplicaţii GUI complexe. Am comparat rezultatele obţinute de procesul nostru cu un scenariu ideal obţinut folosind informaţia garantat corectă şi am arătat că procesul

nostru este eficient ca bază a unui proces automatizat pentru testarea regresiilor unei aplicaţii complexe cu interfaţă grafică.

Desigur, cercetarea efectuată nu ar avea rost fără utilizarea unor aplicaţii potrivite. Pentru acest motiv, în Capitolul 2 am prezentat repoyitoriul nostru de aplicaţii complexe ce au fost utilizate în cadrul cercetării efectuate.

Credinţa noastră este că munca prezentată în cadrul acestei lucrări deshcide noi drumuri în ambele direcţii de cercetare studiate. În primul rând, am descoperit că vizualizarea software beneficiază de aportul adus de unelte de analiză avansate precum Soot şi GUITAR, care în mod curent sunt utilizate mai mult în cercetarea academică. O direcţie a cercetărilor viitoare priveşte integrarea framework-ului nostru de componente în medii de dezvoltare bine cunoscute precum Eclipse folosind mecanismul de pluginuri. Aceasta va permite dezvoltatorilor să beneficieze de pe urma cercetărilor efectuate de noi fără a modifica procesele industriale existente, ceea ce va duce la utilizarea pe scară largă a acestor unelte. Un alt aspect e privitor la cercetările efectuate în domeniu aplicaţiilor GUI. Planurile noastre de viitor prevăd încorporarea procesului euristic aici prezentat în cadrul framework-ului GUITAR pentru a permite repararea automată a cazurilor de testare când acest lucru devine necesar. Mai mult, ţintim să dezvoltăm un mediu de testare uşor de utilizat pentru a permite administrarea pe termen lung a cazurilor de testare care să includă şi un mecanism pentru generarea semi-automată a cazurilor de testare folosind tehnici AI (29, 34). Credem că integrarea acestor unelte cu medii de dezvoltare populare precum Eclipse va grăbi adoptarea noilor metodologii de testare în industrie, ceea ce va duce la crearea de aplicaţii software mai ieftine şi mai fiabile.

Bibliography

[1] Bach, J. Test automation snake oil. *Windows Tech Journal* (1996), 40–44.

[2] Baresi, L., and Young, M. Test oracles. Technical Report CIS-TR-01-02, University of Oregon, Dept. of Computer and Information Science, Eugene, Oregon, U.S.A., August 2001. http://www.cs.uoregon.edu/ michal/pubs/oracles.html.

[3] Bertolini, C., and Mota, A. A framework for gui testing based on use case design. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops* (Washington, DC, USA, 2010), ICSTW '10, IEEE Computer Society, pp. 252–259.

[4] **Arthur-Jozsef, M.** A heuristic process for GUI widget matching across application versions - abstract. In *Abstracts of MaCS 2012*.

[5] **Arthur-Jozsef, M.** jSET - Java Software Evolution Tracker. In *KEPT-2011 Selected Papers*, Presa Universitara Clujeana, ISSN 2067-1180.

[6] **Arthur-Jozsef, M.** jSET - Java Software Evolution Tracker - extended abstract. *KEPT 2011 Conference, Cluj Napoca* (July 2011).

[7] **Arthur-Jozsef, M.** A heuristic process for GUI widget matching across application versions. *Annales Universitatis. Scientiarum Budapestinensis, Sectio Computatorica* (2012).

[8] **Arthur-Jozsef, M.** An initial study on GUI test case replayability. *IEEE International Conference on Automation, Quality and Testing, Robotics AQTR 2012, Cluj-Napoca - submission pending* (2012).

[9] **Arthur-Jozsef, M.** A software repository and toolset for empirical software research. *Studia Informatica UBB, Cluj-Napoca - submitted* (2012).

[10] Brooks, P., Robinson, B., and Memon, A. M. An initial characterization of industrial graphical user interface systems. In *ICST 2009: Proceedings of the 2nd IEEE International Conference on Software Testing, Verification and Validation* (Washington, DC, USA, 2009), IEEE Computer Society.

[11] Brooks, P. A., and Memon, A. M. Automated gui testing guided by usage profiles. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering* (New York, NY, USA, 2007), ASE '07, ACM, pp. 333–342.

[12] Cabral, G., and Sampaio, A. Formal specification generation from requirement documents. *Electron. Notes Theor. Comput. Sci. 195* (January 2008), 171–188.

[13] Dean, J., Grove, D., and Chambers, C. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming* (London, UK, UK, 1995), ECOOP '95, Springer-Verlag, pp. 77–101.

[14] Goodenough, J. B., and Gerhart, S. L. Toward a theory of test data selection. *SIGPLAN Not. 10* (April 1975), 493–510.

[15] Hackner, D., and Memon, A. M. Test case generator for GUITAR. In *ICSE '08: Research Demonstration Track: International Conference on Software Engineering* (Washington, DC, USA, 2008), IEEE Computer Society.

[16] Hammontree, M. L., Hendrickson, J. J., and Hensley, B. W. Integrated data capture and analysis tools for research and testing on graphical user interfaces. In *Proceedings of the SIGCHI conference on Human factors in computing systems* (New York, NY, USA, 1992), CHI '92, ACM, pp. 431–432.

[17] Hass, A. M. J. *Guide to Advanced Software Testing.* Artech House, Inc., Norwood, MA, USA, 2008.

[18] Hou, D. Studying the evolution of the eclipse java editor. In *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange* (New York, NY, USA, 2007), eclipse '07, ACM, pp. 65–69.

[19] Hou, D., and Wang, Y. An empirical analysis of the evolution of user-visible features in an integrated development environment. In *Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research* (New York, NY, USA, 2009), CASCON '09, ACM, pp. 122–135.

[20] Huang, S., Cohen, M. B., and Memon, A. M. Repairing gui test suites using a genetic algorithm. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation* (Washington, DC, USA, 2010), ICST '10, IEEE Computer Society, pp. 245–254.

[21] Jovic, M., Adamoli, A., Zaparanuks, D., and Hauswirth, M. Automating performance testing of interactive java applications. In *Proceedings of the 5th Workshop on Automation of Software Test* (New York, NY, USA, 2010), AST '10, ACM, pp. 8–15.

[22] Lhotak, O. Spark: A flexible point-to analysis framework for java. Tech. rep., McGill University, Montreal, 2002.

[23] Lhotak, O. *Program analysis using binary decision diagrams*. PhD thesis, Montreal, Que., Canada, Canada, 2006. AAINR25195.

[24] McMaster, S., and Memon, A. M. An extensible heuristic-based framework for gui test case maintenance. In *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops* (Washington, DC, USA, 2009), IEEE Computer Society, pp. 251–254.

[25] Memon, A. Gui ripping: Reverse engineering of graphical user interfaces for testing. In *In Proceedings of The 10th Working Conference on Reverse Engineering* (2003), pp. 260–269.

[26] Memon, A., and et al. What test oracle should i use for effective gui testing? In *PROC. IEEE INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING (ASE'03* (2003), IEEE Computer Society Press, pp. 164–173.

[27] Memon, A., Nagarajan, A., and Xie, Q. Automating regression testing for evolving gui software. *Journal of Software Maintenance 17* (January 2005), 27–64.

[28] Memon, A., and Xie, Q. Using transient/persistent errors to develop automated test oracles for event-driven software. In *Proceedings of the 19th IEEE international conference on Automated software engineering* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 186–195.

[29] Memon, A. M. *A comprehensive framework for testing graphical user interfaces.* PhD thesis, 2001. AAI3026063.

[30] Memon, A. M. An event-flow model of gui-based applications for testing. *Software Testing, Verification and Reliability 17*, 3 (2007), 137–157.

[31] Memon, A. M. Automatically repairing event sequence-based gui test suites for regression testing. *ACM Trans. Softw. Eng. Methodol. 18* (November 2008), 4:1–4:36.

[32] Memon, A. M., Banerjee, I., and Nagarajan, A. DART: A framework for regression testing nightly/daily builds of GUI applications. In *Proceedings of the International Conference on Software Maintenance 2003* (Sept. 2003).

[33] Memon, A. M., Pollack, M. E., and Soffa, M. L. Automated test oracles for guis. *SIGSOFT Softw. Eng. Notes 25* (November 2000), 30–39.

[34] Memon, A. M., Pollack, M. E., and Soffa, M. L. Hierarchical GUI test case generation using automated planning. *IEEE Trans. Softw. Eng. 27*, 2 (2001), 144–155.

[35] Memon, A. M., and Soffa, M. L. Regression testing of GUIs. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering* (New York, NY, USA, 2003), ACM Press, pp. 118–127.

[36] Memon, A. M., Soffa, M. L., and Pollack, M. E. Coverage criteria for GUI testing. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on*

Foundations of software engineering (New York, NY, USA, 2001), ACM Press, pp. 256–267.

[37] Navarro, G. A guided tour to approximate string matching. *ACM Comput. Surv. 33* (March 2001), 31–88.

[38] Nguyen, D. H., Strooper, P., and Suess, J. G. Model-based testing of multiple gui variants using the gui test generator. In *Proceedings of the 5th Workshop on Automation of Software Test* (New York, NY, USA, 2010), AST '10, ACM, pp. 24–30.

[39] Porter, A. A., Yilmaz, C., Memon, A. M., Schmidt, D. C., and Natarajan, B. Skoll: A process and infrastructure for distributed continuous quality assurance. *IEEE Trans. Software Eng. 33*, 8 (2007), 510–525.

[40] Ramler, R., and Wolfmaier, K. Economic perspectives in test automation: balancing automated and manual testing with opportunity cost. In *Proceedings of the 2006 international workshop on Automation of software test* (New York, NY, USA, 2006), AST '06, ACM, pp. 85–91.

[41] Robinson, B., and Brooks, P. An initial study of customer-reported gui defects. In *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops* (Washington, DC, USA, 2009), IEEE Computer Society, pp. 267–274.

[42] Staats, M., Whalen, M. W., and Heimdahl, M. P. Programs, tests, and oracles: the foundations of testing revisited. In *Proceedings of the 33rd International Conference on Software Engineering* (New York, NY, USA, 2011), ICSE '11, ACM, pp. 391–400.

[43] Strecker, J., and Memon, A. M. Relationships between test suites, faults, and fault detection in gui testing. In *ICST '08: Proceedings of the First international conference on Software Testing, Verification, and Validation* (Washington, DC, USA, 2008), IEEE Computer Society.

[44] Sundaresan, V. Practical techniques for virtual call resolution in java. Tech. rep., McGill University, 1999.

[45] Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., and Sundaresan, V. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research* (1999), CASCON '99, IBM Press, pp. 13–.

[46] Website. http://guitar.sourceforge.net/. Home of the GUITAR toolset.

[47] Website. http://www.sable.mcgill.ca/soot/. Soot home at McGill University.

[48] Website. https://svn.sable.mcgill.ca/wiki/index.cgi/SootUsers. Soot's list of users.

[49] Website. http://sourceforge.net/projects/freemind/. Home of the FreeMind project.

[50] Website. http://sourceforge.net/projects/jedit/. Home of the jEdit project.

[51] Website. http://code.google.com/p/google-diff-match-patch (Home of an implementation for diff-match-patch).

[52] Xie, Q. *Developing cost-effective model-based techniques for gui testing.* PhD thesis, College Park, MD, USA, 2006. AAI3241432.

[53] Xie, Q., and Memon, A. M. Model-based testing of community-driven open-source gui applications. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 145–154.

[54] Xie, Q., and Memon, A. M. Designing and comparing automated test oracles for gui-based software applications. *ACM Trans. Softw. Eng. Methodol. 16* (February 2007).

[55] Yuan, X., Cohen, M. B., and Memon, A. M. Gui interaction testing: Incorporating event context, 2011.

[56] Yuan, X., and Memon, A. M. Alternating gui test generation and execution. In *Proceedings of the Testing: Academic & Industrial Conference - Practice and Research Techniques* (Washington, DC, USA, 2008), IEEE Computer Society, pp. 23–32.

[57] Yuan, X., and Memon, A. M. Generating event sequence-based test cases using gui runtime state feedback. *IEEE Transactions on Software Engineering 36* (2010), 81–95.