UNIVERSITATEA ALEXANDRU IOAN CUZA IAŞI

FACULTATEA DE INFORMATICĂ

LUCRARE DE LICENŢĂ

Generare automată de teste

propusă de

Iuliana Gabor

Sesiunea: *februarie, 2018*

Coordonator ştiinţific

Lect. dr. Andrei Arusoaie

UNIVERSITATEA ALEXANDRU IOAN CUZA IAŞI

FACULTATEA DE INFORMATICĂ

Generare automată de teste

Iuliana Gabor

Sesiunea: *februarie, 2018*

Coordonator ştiinţific

Lect. dr. Andrei Arusoaie

DECLARAȚIE PRIVIND ORIGINALITATE ȘI RESPECTAREA DREPTURILOR DE AUTOR

Prin prezenta declar ca Lucrarea de Licență cu titlul *Generare automată de* teste este scrisă de mine și nu a mai fost prezentată niciodată la o altă facultate sau instituție de învățământ superior din țară sau din străinătate. De asemenea, declar că toate sursele utilizate, inclusiv cele preluate de pe Internet, sunt indicate în lucrare, cu respectarea regulilor de evitare a plagiatului:

- toate fragmentele de text reproduse exact, chiar și în traducere proprie din altă limbă, sunt scrise între ghilimele și dețin referința precisă a sursei;
- reformularea în cuvinte proprii a textelor scrise de către alți autori deține referință precisă;
- codul sursă, imagini etc. preluate din proiecte open-source sau alte surse sunt utilizate cu respectarea drepturilor de autor și dețin referințe precise;
- rezumarea ideilor altor autori precizează referința precisă la textul original.

Iași,

Absolvent: Gabor Iuliana

DECLARAȚIE DE CONSIMȚĂMÂNT

Prin prezenta declar că sunt de acord ca Lucrarea de licență cu titlul *Generare automată de teste,* codul sursă al programelor și celelalte conținuturi (grafice, multimedia, date de test, etc) care însoțesc această lucrare să fie utilizate în cadrul Facultății de Informatica. De asemenea, sunt de acord ca Facultatea de Informatica de la Universitatea Alexandru Ioan Cuza Iași să utilizeze, modifice, reproducă și să distribuie în scopuri necomerciale programele-calculator, format executabil și sursă, realizate de mine în cadrul prezentei lucrări de licență.

Iași,

Absolvent: Gabor Iuliana

Cuprins

1. Introducere

Odată cu apariția primelor sisteme informatice a apărut și testarea software. La început programele erau testate de cei care le scriau iar procesul era denumit verificare. Dar odată cu dezvoltarea sistemelor informatice, a crescut și complexitatea acestora iar testarea lor a devenit tot mai complicată.

Chiar dacă, între timp, oamenii au început să se specializeze și pe testarea software, acest proces nu poate fi făcut perfect. Nici un program nu poate fi testat 100%. De multe ori costurile corectării unei erori după ce produsul a fost livrat sunt foarte mari. Există nenumărate cazuri în care, din cauza erorilor, produsele fie nu au mai fost livrate, fie nu au mai putut fi corectate, fie pagubele provocate au fost uriașe.

În această situație, tot ce poate face o echipă este să se asigure că produsul la care a lucrat și pe care urmează să îl livreze, nu are erori majore, care să producă daune. Problema e cum ar putea fi eficientizat procesul de testare pentru a fi descoperite cat mai multe erori, in special erorile majore, pentru a fi corectate la timp?

Pentru eficientizarea procesului de testare fiecare echipă care dezvoltă un anumit produs încearcă să adapteze tipurile și metodele de testare necesare în funcție de tipul produsului, cerințele acestuia, resurse necesare, tehnologii utilizate, nivelul de dezvoltare la care a ajuns produsul, etc.

În funcție de tipul de testare pentru unele produse este mai potrivită testarea manuală, pentru altele testarea automată dar pentru majoritatea este mai eficientă combinarea celor două moduri de testare.

În funcție de metoda de testare pentru unele produse este mai potrivită metoda white-box, pentru altele metoda black-box dar pentru majoritatea este utilizată metoda grey-box.

În funcție de nivelul de dezvoltare la care a ajuns produsul poate fi necesară testarea funcțională (testarea unitară, testarea integrării, testarea de sistem, regresie, testare de acceptare) sau testare non-funcțională (testarea performanței, utilizabilității, securității și portabilității).

Dar toate aceste modalități de testare au în spate aceeași idee: pentru anumite date de intrare aplicate unor module, funcții, instrucțiuni ori porțiuni de cod se verifică datele de ieșire. De multe ori această abordare necesită timp si efort pentru găsirea datelor de intrare potrivite pentru datele de ieșire așteptate.

1.1. Motivație

Consider că optimizarea și eficientizarea procesului de testate este o problemă ce merită dezbătută și pentru care este necesară găsirea de soluții care să poată fi dezvoltate și aplicate în practică. Pe lângă acestea, găsirea unei soluții de bază care să nu depindă de limbajul, platforma ori contextul în care este utilizată ar crește gradul de eficientizare și ar reduce cu mult timpul necesar procesului de testate.

În acest context, mi-am propus să prezint un alt mod de testare, ca soluție la reducerea efortului depus și a timpului necesar căutării și găsirii cazurilor de test potrivite: generarea automată de teste utilizând execuția simbolică, tehnică independentă de limbaj.

Deși mai există soluții bazate pe execuție simbolică, utilizate în practică, acestea sunt strict specifice unor contexte și/sau limbaje de programare.

1.2. Structura lucrării

Lucrarea este structurată în șase capitole care prezintă pas cu pas problema ce se dorește a fi rezolvată, contextul în care se produce și soluția propusă pentru rezolvarea problemei.

Primul capitol cuprinde o scurta introducere completată de motivație, structura lucrării și contribuțiile personale.

În al doilea capitol este prezentat contextul problemei: ce este testarea programelor, tipurile și modurile de testare, care sunt în prezent soluțiile aplicate în practică pentru rezolvarea problemei testării software si instrumente utilizate pentru ușurarea procesului de testare.

Al treilea capitol prezintă problema ce se dorește a fi rezolvată, o scurta prezentare a soluției propuse și noutatea adusă de soluția propusă comparativ cu soluțiile deja existente.

Capitolul al patrulea cuprinde prezentarea instrumentelor necesare implementării soluției, modul cum acestea interacționează si descrierea mai detaliată a soluției.

În al cincilea capitol este prezentat modul de implementare a aplicației care rezolvă problema testării, tehnologiile utilizate și modul cum acestea interacționează pentru a furniza rezultatele.

Al șaselea capitol prezintă rezultatele obținute in urma aplicării soluției propuse pe limbajele IMP și KOOL.

Al șaptelea capitol prezintă concluziile finale si posibilități de dezvoltare a soluției.

1.3. Contribuții personale

- **Soluția propusă constă în generarea de teste, utilizând execuția simbolică, pornind de la definiția limbajelor, în contextul în care, în prezent, testarea se face executând cod scris într-un anumit limbaj.**
- **Pe lângă aceasta, gradul de noutate este susținut și de faptul că soluția propusă este independentă de limbaj, în situația în care, soluțiile existente bazate pe execuție simbolică sunt limitate la un anumit limbaj sau context.**
- **Utilizând anumite tool-uri deja existente, necesare obținerii rezultatelor dorite, am implementat modul de comunicare între acestea, translatând la fiecare pas rezultatele obținute la pasul curent, din forma în care au fost generate într-o formă acceptată la pasul următor.**

2. Testarea programelor

Orice produs, înainte de a fi utilizat sau comercializat, este supus unor teste prin care se verifică dacă respectă anumite cerințe și/sau dacă funcționează conform planului după care a fost construit. Toate aceste verificări au ca scop descoperirea eventualelor defecțiuni și corectarea acestora.

Industria IT nu face excepție. Este necesar ca orice produs software, înainte de a fi furnizat clientului sau utilizatorilor, să treacă printr-un proces de testare numit si testare software.

2.1. Ce este testarea software?

Conform [1] testarea software este acțiunea de efectuare a unu sau mai multe teste, unde un teste este o operațiune tehnică care determină una sau mai multe caracteristici ale unui element software sau a unui sistem dat, conform unei proceduri specifice.

În ciclul de dezvoltare a unui produs software, testarea are un rol foarte important, chiar dacă uneori nu i se acordă atenția necesară. Sunt cunoscute multe cazuri în care programele au fost livrate cu bug-uri care au produs pagube uriașe iar în unele cazuri ducând la pierderea în totalitate a produsului.

- *The Lion King Animated Storybook* (1994 – 1995)

  A fost primul joc pentru copii lansat de Disney și distribuit pe CD. Deoarece a fost intens promovat iar lansarea a fost spre sfârșitul anului, foarte mulți părinți l-au cumpărat pentru a face o surpriză plăcută, de Crăciun, copiilor lor. Doar că lucrurile au luat o întorsătură neașteptată: jocul nu funcționa pe foarte multe calculatoare. Această eroare nu a fost detectată la timp deoarece produsul nu a fost testat pe diferite tipuri de sisteme printre care să se numere și cele comune din comerț, ci doar pe un număr restrâns de calculatoare printre care cele pe care a fost dezvoltat jocul. [2]
- *Y2K* (până în anul 1999)

  Pentru acest bug nu există o dată exactă la care s-a manifestat.

În anii 1970 – 1980, când spațiul de stocare al calculatoarelor era foarte mic comparativ cu cel al sistemelor de care dispunem astăzi, programatorii erau nevoiți să optimizeze la maxim programele pe care le dezvoltau, pentru ca acestea să poată fi utilizate. Una dintre aceste optimizări consta în scrierea anului, din datele calendaristice, cu două cifre în loc de patru, primele două fiind considerate ca fiind 1 și 9. Problema care se punea era trecerea de la anul 1999 la anul 2000, pe care calculatoarele, în condițiile în care acestea lucrau cu ani în format de două cifre, ar fi văzut-o ca fiind trecerea de la anul 1999 la anul 1900. Acest lucru ar fi dat peste cap toate sistemele informatice ceea ce ar fi dus la un dezastru economic la nivel mondial.

Se presupune că, s-au cheltuit sute de milioane de dolari pentru a repara acest tip de eroare. [2]

Conform [1] într-un raport NASA realizat în anul 2000, sunt prezentate o parte din pierderile suferite din cauza unor erori grave ce nu au fost descoperite la timp. După cum se poate observa și în Tabel 1 preluat din [1], nu s-au înregistrat doar pierderi financiare (care sunt enorme) ci și pierderi de date și chiar pierderi de vieți omenești.

	Airbus A320 (1993)	Ariane 5 Galileo Poseidon Flight 965 (1996)	Lewis Pathfinder USAF Step (1997)	Zenit 2 Delta 3 Near (1998)	DS-1 Orion 3 Galileo Titan 4B (1999)
Aggregate cost		$640 million	$116.8 million	$255 million	$1.6 billion
Loss of life	3	160			
Loss of data		Yes	Yes	Yes	Yes

Tabel 1

Cazurile și cifrele prezentate mai sus sunt doar o mică parte din efectele pe care le-au avut testarea insuficientă sau ineficientă a programelor de-a lungul timpului. Tocmai pentru a fi evitate astfel de situații, oamenii au încercat definirea cât mai strictă a ceea ce înseamnă testarea programelor cât și identificarea noțiunilor și metodelor care ar duce la eficientizarea acestui proces.

Testarea software are ca scop identificarea comportamentului anormal al programelor. La baza acestui proces stau verificarea și validarea.

Conform [3] *validarea* este procesul de evaluare a programului, după finalizarea dezvoltării acestuia, pentru a ne asigura că acesta funcționează conform cerințelor clientului și poate fi utilizat în scopul pentru care a fost dezvoltat. Acest proces necesită un minim de cunoștințe despre domeniul în care va fi utilizat produsul. Validarea răspunde la întrebarea: *Am dezvoltat produsul corect/cerut?*

Conform [3] *verificarea* este procesul ce determină dacă produsul aflat la un anumit stadiu în procesul de dezvoltare îndeplinește cerințele stabilite într-o fază de dezvoltare anterioară. Cu alte cuvinte este procesul prin care se determină dacă produsul are toate funcționalitățile cerute de client. Acest proces necesită atât cunoștințe tehnice cât și un minim de cunoștințe în domeniul pentru care este dezvoltat produsul. Verificarea răspunde la întrebarea: *Am dezvoltat corect produsul?*

Așa cum este precizat în [4] de procesul de testare sunt strâns legate noțiunile: defect, eroare si eșec.

Defectele sunt principalele cauze ale erorilor. Acestea pot fi descoperite doar dacă porțiunea de cod în care sunt prezente este executată în anumite condiții. De multe ori trec perioade de timp îndelungate până la descoperirea unor defecte. De exemplu: vrem să adunăm a două numere pe 32 de biți iar rezultatul să îl reprezentăm tot pe 32 de biți.

Erorile sunt efectele activării defectelor. Revenind la exemplul anterior, defectul se manifestă doar atunci când rezultatul adunării celor două numere nu mai poate fi reprezentat pe 32 de biți deoarece se produce depășire. În acest caz apare eroarea.

Eșecurile (cunoscute și sub denumirea de bug-uri) apar atunci când sistemul trece peste erorile care apar sau le ignoră, furnizând un rezultate eronate. Continuând exemplul anterior, eșecul este rezultatul eronat obținut în urma încercării de a reprezenta rezultatul adunării pe doar 32 de biți.

2.2. Tipuri și moduri de testare

Testarea programelor este un proces ce necesită timp și resurse. Uneori acest proces, în funcție de natura programului care trebuie testat, devine monoton și repetitiv. De multe ori, pentru a reduce din timpul necesar testării, sunt scrise programe care să execute o anumită suită de teste, fără a mai fi necesară intervenția unei persoane la fiecare pas. Dar nu toate testele pot fi rulate în mod automat. Astfel se pot distinge două tipuri de testare: manuală și automată.

Testarea manuală presupune prezența unui tester care să joace rolul de utilizator final. Spre deosebire de utilizatorul final, testerul verifică comportamentul programului testat pe baza unui set de cazuri de test stabilit pe baza cerințelor. Rezultatele obținute sunt comparate cu rezultatele așteptate.

Testarea automată este tipul de testare care presupune rularea unui set de teste cu ajutorul unui program, fără a fi necesară intervenția la fiecare pas a testerului. În funcție de modul cum a fost gândit și implementat programul de automatizare a testelor, rezultatele obținute pot fi oferite pe rând, în timpul rulării testelor sau la final, după încheierea rulării întregului set de teste.

Deși testarea automată poate reduce, cu mult, timpul necesar testării programelor, există numeroase cazuri în care acest tip de testare fie nu poate fi aplicat fie ar fi mai greoi decât testarea manuală. De exemplu testarea autentificării într-o aplicație ar fi dificil de automatizat deoarece sunt multe cazuri distincte: user și parolă corecte, user corect și parolă greșită, autentificare multiplă și lista poate continua.

De cele mai multe ori este ales tipul de testare potrivit în funcție de componenta care trebuie testată.

Deoarece procesul de testare nu presupune doar verificarea comportamentului produsului final ci și verificarea comportamentului anumitor funcții implementate în interiorul programului, se disting trei moduri sau metode de testare: metoda black-box, metoda white-box și metoda gray-box.

Metoda black-box: este metoda prin care testerul nu are acces la structura internă a programului ci cunoaște doar rezultatele care ar trebui obținute pentru un anume set de date de intrare. Acest lucru poate fi și un avantaj dar și un dezavantaj. Avantaj deoarece testerul, necunoscând structura internă a aplicației nu cunoaște nici modul cum aceasta a fost gândită și implementată să funcționeze, crescând astfel probabilitatea descoperirii cazurilor în care aplicația se comportă anormal. Dezavantaj deoarece testerul verifică comportamentul întregii aplicații, astfel fiind mai greu de găsit cauzele erorilor.

Metoda white-box: este metoda prin care testerul, având acces la codul sursă, poate testa și părți mici de cod nu doar toată aplicația ca în cazul metodei black-box. Acest lucru

este în același timp și un avantaj dar și un dezavantaj. Avantaj deoarece testerul, având acces la codul sursă poate testa porțiuni mici de cod, reducând astfel timpul necesar găsirii și corectării porțiunilor de cod care generează comportamentul nedorit. Dezavantaj deoarece, cunoașterea structurii interne și a comportamentului aplicației, ar putea limita testerul la un anumite tipuri de cazuri de testare.

Metoda grey-box: este o combinație între metoda black-box și metoda white-box. În acest caz testerul are acces doar la anumite părți de cod.

2.3. Instrumente existente

Ciclul de dezvoltare a unui produs cuprinde mai multe etape care diferă în funcție de modelul ales pentru dezvoltarea produsului scopul final fiind același: livrarea unui produs de calitate care să respecte cerințele clientului. Pentru a atinge acest scop este necesară testarea produsului la fiecare nivel de dezvoltare.

De exemplu în modelul V, așa cum se poate observa și în Fig. 1 preluată din [4] există câte un mod de testare adaptat fiecărui nivel de dezvoltare a produsului.

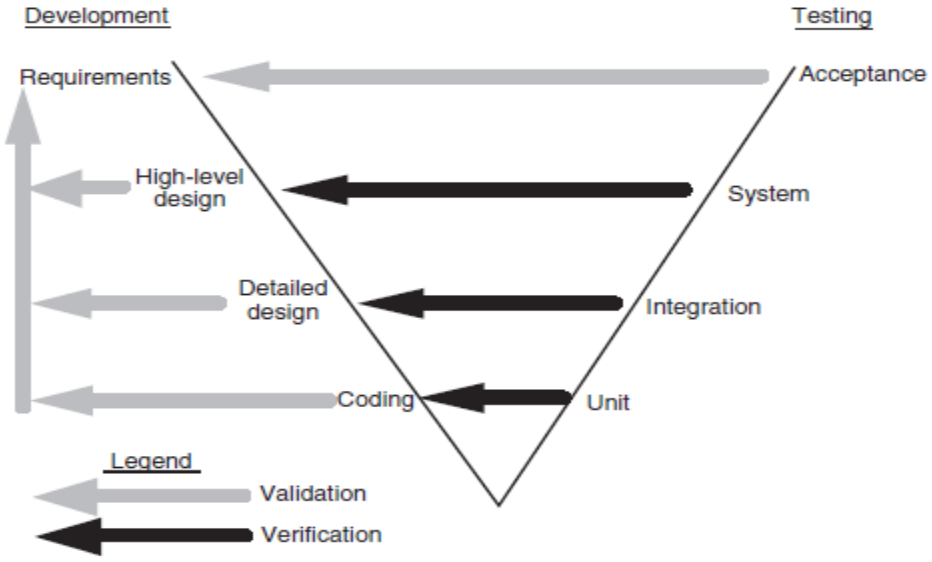

Fig. 1

În testarea unitară sunt izolate și testate funcțiile, clasele, ori blocurile de cod ce alcătuiesc programul, cu scopul de a se testa comportamentul fiecărei porțiuni de cod.

În testarea integrării este verificat modul cum comunică componentele între ele cu scopul de a dezvolta un sistem stabil.

În testarea de sistem este verificat comportamentul programului în diferite situații și pe diferite sisteme.

În testarea de acceptare, clientul primește produsul și face propriile teste, după ce, în prealabil, produsul final a fost testat în interiorul companiei.

Deoarece, pe parcursul dezvoltării produsului, o serie de pași se repetă în procesul de testare a produselor, au fost dezvoltate diferite tool-uri care să reducă timpul alocat testării și ușureze munca celor ce testează produsul.

Embunit [5] este un tool pentru testare unitară a programelor scrise în C sau C++, în special a programelor embedded. Prin intermediu acestui tool, parcurgând o serie de pași ca în Fig. 2 preluată de pe [5], este generat codul sursă pentru testarea unitară. Astfel, având codul sursă, testerul/programatorul nu va mai trebui să scrie codul care se repetă ci doar să definească comportamentul testelor.

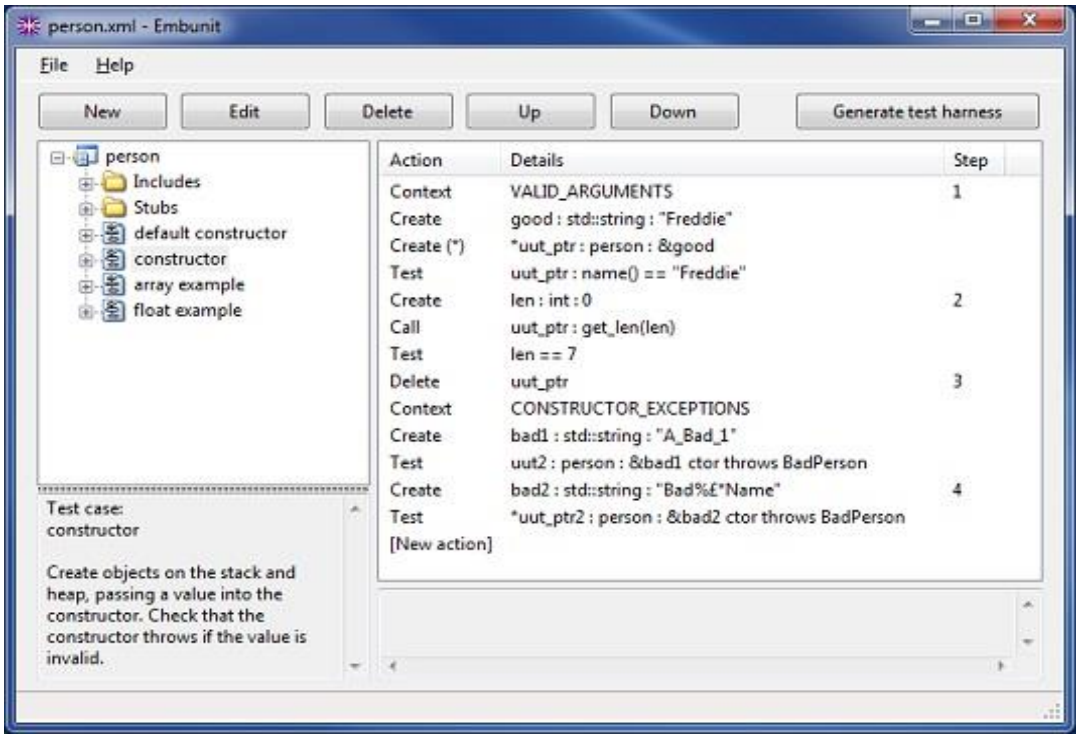

Fig. 2

JUnit [6] este un tool pentru generare de elemente pentru testarea unitară a programelor dezvoltate în Java (Fig. 3 preluată de pe [7]). Ca și în cazul *Embunit*, codul de bază este generat automat, programatorul având sarcina de a determina comportamentul testelor.

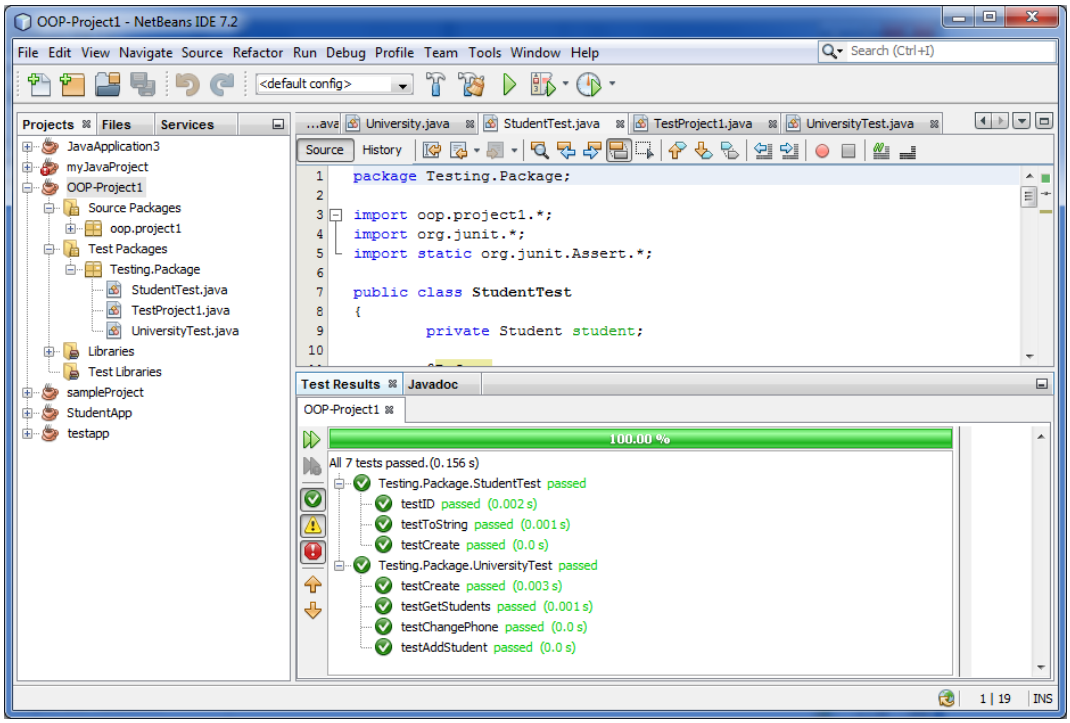

Fig. 3

Astfel de tool-uri pentru testare unitară există și pentru limbaje ca: python (PyUnit), javascript (JsUnit), etc.

3. Problema generării automate de teste

Așa cum am prezentat în capitolele anterioare, există diferite modalități de a testa programele: testare manuală atunci când cel ce testează programul joacă rolul unui utilizator, testare automată atunci când testele sunt rulate în mod automat cu ajutorul unor scripturi ori programe, testarea având sau nu cunoștințe despre modul de implementare a programului utilizând una din cele trei metode: black-box, white-box sau gray-box, testarea utilizând anumite tool-uri. Indiferent de modalitățile alese, scopul testării programelor este găsirea a cât mai multe erori și defecte, în special a celor care ar putea provoca pierderi de orice fel.

3.1. Problema ce se dorește a fi rezolvată

În ultimii ani industria IT s-a dezvoltat foarte mult datorită dorinței oamenilor de a-și simplifica, din anumite puncte de vedere, viața. Aceasta a dus la creșterea complexității programelor dezvoltate și apariția a numeroase tehnologii specializate strict pe anumite ramuri ale acestei industrii. Odată cu această dezvoltare, s-au diversificat cerințele iar programele ce se doresc a fi dezvoltate sunt din ce în ce mai sofisticate, lucru care determină creșterea complexității codului din spatele programelor.

Dar cum un program nu poate fi utilizat înainte de a fi testat, creșterea complexității programelor și a codului din spatele acestora a dus la creșterea complexității procesului de testare. Un proces mai complex necesită mai mult timp și mai mult efort.

Pentru a optimiza procesul de testare s-au dezvoltat diferite tool-uri care să reducă din volumul de muncă și timpul necesar testării programelor. Însă problemele principale care rămân și care, dacă ar fi rezolvate, ar reduce mult din timpul și efortul necesar testării programelor, sunt căutarea de cazuri de test potrivite și diversitatea limbajelor de programare utilizate.

Căutarea de cazuri de test potrivite este un proces care poate dura destul de mult în funcție de complexitatea programului ce urmează a fi testat.

Diversitatea limbajelor reprezintă o problemă în procesul de testare deoarece, o idee de optimizare a acestuia, pentru a putea fi pusă în practică, trebuie adaptată fiecărui limbaj.

Prin această lucrare propun ca soluție la cele două probleme menționate anterior generarea de cazuri de test, pornind de la definiția limbajelor, utilizând execuția simbolică, tehnică independentă de limbaj.

3.2. Diferența comparativ cu soluțiile deja existente

Utilizarea execuției simbolice în testare nu este o noutate. Există diferite tool-uri, cum ar fi KLEE sau TRACER, care au la bază această tehnică și care sunt utilizate și în practică. Problema e că acestea au fost dezvoltate pentru anumite sisteme și limbaje de programare, astfel, acestea pot fi utilizate doar în anumite situații.

Noutatea soluției propuse în această lucrare constă în utilizarea execuției simbolice ca tehnică independentă de limbaj, lucru ce nu este întâlnit la alte tool-uri. Aplicarea acestei tehnici este posibilă doar pentru limbajele de programare ce au definiția scrisă în K Framework [8], deoarece respectă aceleași reguli pentru definirea limbajelor.

4. Generare de teste utilizând execuție simbolică

Limbajele de programare permit executarea multor tipuri de instrucțiuni și operații, mai simple sau mai complexe. Dar pentru a putea genera cazuri de test, pornind de la definiția limbajelor, este necesar ca acestea să permită atribuirea de valori simbolice variabilelor, efectuarea de operații aritmetice și logice cu acest tip de valori iar la verificarea condițiilor să poată construi șiruri de condiții. Dar cum limbajele de programare uzuale nu permit aceste lucruri, a fost necesară găsirea unei soluții: modificarea definiției limbajelor astfel încât acestea să permită lucrul cu valori simbolice. Pare o soluție acceptabilă doar ca aceasta ar însemna modificarea definițiilor tuturor limbajelor pentru care vrem să aplicăm această soluție și adaptarea acesteia la modul cum e definit limbajul, ceea ce ar necesita mult timp și foarte multă muncă.

Pentru a reduce timpul de lucru și volumul de muncă necesar am ales să lucrez cu definițiile limbajelor în K Framework. Avantajul utilizării K Framework constă în faptul că definițiile limbajelor sunt scrise în același mod deci modificările făcute unui limbaj pot fi aplicate tuturor limbajelor.

4.1. Definirea limbajelor de programare in K Framework

K [8] este un framework prin intermediul căruia se pot rescrie definițiile limbajelor de programare, se pot defini limbaje de programare noi sau pot fi adăugate elemente noi pentru limbajele deja existente.

Pentru a defini un limbaj în K Framework sunt necesare:

- definirea sintaxei
- definirea semanticii

Definirea sintaxei constă în definirea cuvintelor, a unor șiruri de caractere sau a unor tipuri de date predefinite care să fie recunoscute ca parte din limbajul nou definit.

Definirea semanticii constă în definirea regulilor ce vor fi aplicate asupra sintaxei pentru a determina modul de funcționare a limbajului. Aceasta începe prin definirea unei configurații care pune la dispoziție o serie de celule computaționale utilizate la parsarea sintaxei și aplicarea regulilor specifice acesteia.

Utilizarea K Framework vine cu o serie de avantaje printre care:

- Existența unor tipuri de date predefinite care ajută la definirea mai rapidă a limbajelor dar și posibilitatea definirii unor noi tipuri de date.
- Utilizarea modulelor la definirea limbajelor. Astfel, la o definiție dată, dacă dorim adăugarea unui operator, a unui tip de comportament sau creșterea complexității limbajului, nu suntem obligați să rescriem toată definiția limbajului ci putem defini un nou modul pe care să îl integrăm în definiția deja existentă.
- Utilizarea acelorași tehnici de definire a sintaxei și semanticii, indiferent de complexitatea și instrumentele de care va dispune noul limbaj. Aceasta înseamnă că descoperirea unei tehnici revoluționare la nivel unui limbaj de programare ar putea fi adaptată și extinsă foarte ușor și la alte limbaje ceea ce ar duce la o evoluție mai rapidă și mai eficientă a domeniului.

4.2. Execuția simbolică

Execuția simbolică, conform [9], este o tehnică de analiză a programelor utilizată în testare, verificare, debugging, și alte aplicații. Principiul de bază al acestei tehnici este executarea programelor înlocuind datele de intrare obișnuite cu valori arbitrare numite și valori simbolice.

De exemplu setul de instrucțiuni:

```
int x = -10;
while   ( x < 0 ) {
        x = x + 1;
}
```

poate fi scris astfel:

```
int x = sym(y);
while ( x < 0 ) {
        x = x + 1;
}
```

unde *sym(y)* este o valoare simbolică.

Notăm cu:

- *C0* condiția *(x < 0)*, *C1* condiția *((x + 1) < 0)*, *C2* condiția *((x + 2) < 0)*, …, *Cn* condiția *((x + n) < 0)*
- *F1* negația condiției *C0*, *F2* negația condiției *C1*, … , *Fn* negația condiției *Cn-1*

În urma execuției simbolice a programului de mai sus, rezultă un arbore binar ca în Fig. 4 unde nodurile sunt condițiile: *C0, C1, C2, ... , Cn* iar frunzele reprezentate de negațiile condițiilor *C0, C1, C2, ... , Cn* și anume: *F1, F2, ... , Fn*.

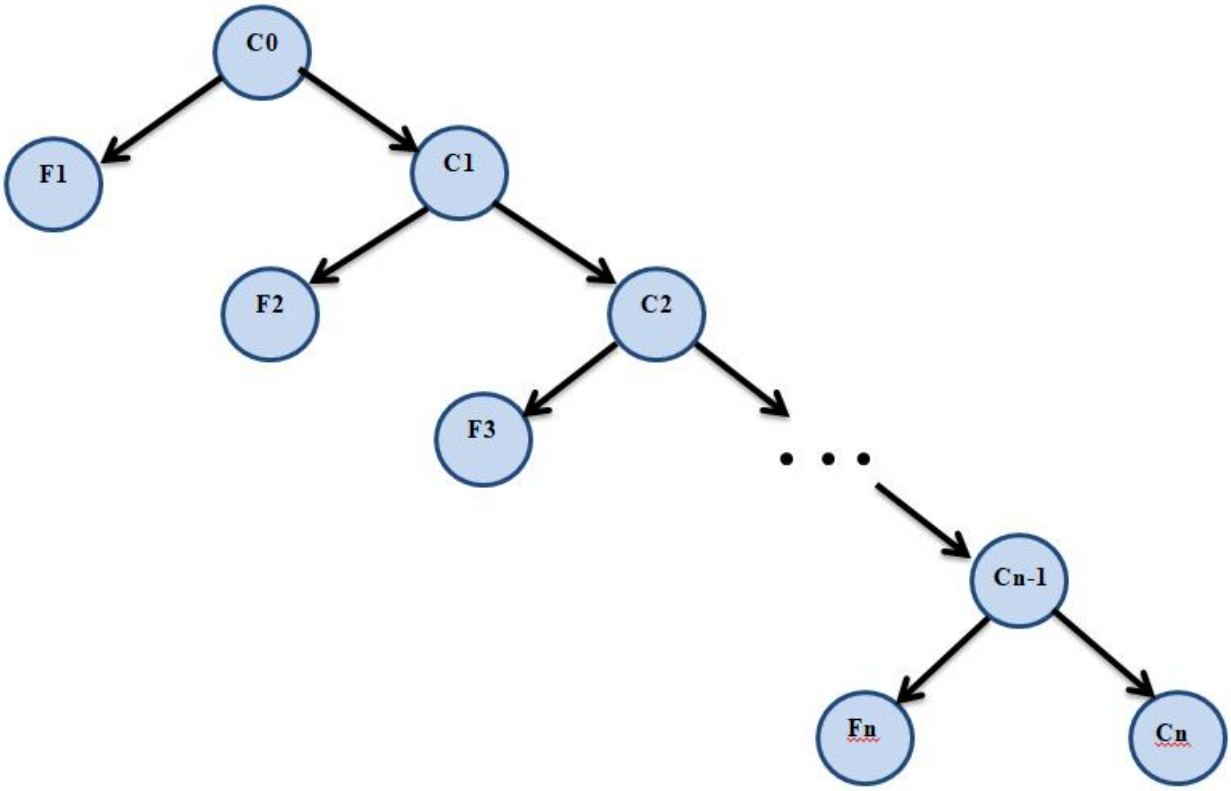

Fig. 4

Fie $i \in \mathrm{N},\ 0 \le i < n$.

Orice condiție *Ci* poate fi scrisă sub forma *Ci = (Fi+1 and Ci+1).*

Parcurgând astfel arborele din Fig. 4 de la rădăcină până la frunze obținem o formule din care pot fi generate valori posibile pentru valorile simbolice.

Așa cum este specificat și în [9], execuția simbolică are următoarele proprietăți:

- *Acoperire*: fiecărei execuții concrete îi corespunde o execuție simbolică.
- *Precizie*: fiecărei execuții simbolice îi corespunde o execuție concretă.

Demonstrațiile acestor proprietăți nu vor fi redate în această lucrare deoarece nu acesta este subiectul lucrării, dar pot fi găsite în [9].

4.3. SMT solver: Z3

Z3 [10] este un SMT solver (Satisfiability Modulo Theories solver) sau demonstrator de teoreme dezvoltat de Microsoft Research. Acesta este utilizat pentru stabilirea satisfiabilității unui subset de formule din logica de ordinul I.

Pentru a putea evalua formule în Z3 sunt necesare respectarea mai multor reguli. Voi prezenta pe cele aplicabile proiectului pe baza căruia am scris această lucrare:

- Declararea funcțiilor și a constantelor se face înaintea adăugării formulei.

  Declararea funcțiilor se face prin sintaxa:

  (declare-fun f (Int Bool) Int)

  În acest exemplu este definită o funcție f ce are doi parametri, unul de tip Int celălalt de tip Bool, a rezultatul fiind de tip Int.

  Declararea constantelor se face prin sintaxa:

  (declare-const a Int)

În acest exemplu este definită o constantă de tip Int. Constantele mai pot fi definite și ca funcții fără parametri.

- Formulele sunt adăugate prin intermediul sintaxei:

(assert (operator operand1 operand2))

unde: *operator* poate fi orice operator aritmetic sau logic

operand1 și *operand2* pot fi funcții sau constante definite anterior, valori întregi, reale, booleene sau formule de forma (operator' operand1' operand2')

rezultatul formulei (operator operand1 operand2) trebuie să fie de tip boolean.

Pot fi adăugate oricâte formule operatorul dintre ele fiind default *and*.

- Verificarea satisfiabilității formulei se face adăugând la final sintaxa:

(check-sat)

- Generarea de valori posibile aplicate constantelor sau funcțiilor se face prin adăugarea la final a sintaxei:

(get-model)

Dacă formula evaluată nu va genera erori (respectă standardele acceptate de Z3), atunci SMT solverul va returna unul din rezultatele:

- *sat* (și eventual un model pentru constante) dacă formula este satisfiabila
- *unsat* dacă formula este nesatisfiabilă

De exemplu pentru formula:

(declare-const a Int)

(declare-const b Int)

(assert (> a 10))

(assert (> b 10))

(assert (< a b))

(check-sat)

Z3 va returna *sat*. Daca la sfârșitul formulei anterioare este adăugat și *(get-model)*, SMT solverul va oferi și modelul:

```
(model
  (define-fun b () Int
    12)
  (define-fun a () Int
    11)
)
```

Dar dacă, în formula anterioară, înlocuim *(assert (> b 10))* cu *(assert (< b 10))*, Z3 va returna *unsat*.

4.4. Descrierea soluției

Soluția propusă este rezultatul îmbinării într-un mod cât mai eficient a celor 3 noțiuni prezentate anterior.

Pentru a ajunge la rezultatul dorit și anume generarea de cazuri de test utilizând execuția simbolică, am luat definiția in K Framework a două limbaje asupra cărora am aplicat o serie de transformări, astfel încât variabilele să poată lua și valori simbolice, să se poată efectua operații aritmetice și logice cu valori simbolice și să poată fi construite șirurile de condiții necesare generării de cazuri de test. Dar mai multe detalii legate de implementare vor fi discutate în capitolul următor.

Șirurile de condiții sunt preluate și transformate în formule. Cu ajutorul unui SMT solver (pentru această aplicație am ales Z3) se determină satisfiabilitatea formulelor.

Dacă o formulă este evaluată ca fiind satisfiabilă, se generează valori posibile în locul valorilor simbolice pentru variabilele implicate. Valorile generate sunt cazurile de test.

Dar dacă o formulă este evaluată ca fiind nesatisfiabilă atunci șirul de condiții din care a fost construită formula nu va putea fi îndeplinit niciodată, deci nu există valori posibile care să ducă la îndeplinirea acelui șir de condiții.

Pentru ca aplicația să poată fi utilizată mai ușor, datele de intrare să poată fi introduse mai rapid dar și pentru a evita erorile care pot să apară din cauza introducerii greșite a datelor de intrare, de la tastatură, într-o consolă sau într-un terminal, am proiectat și implementat pentru aceasta și interfață grafică. Unul din avantajele acesteia este că utilizatorul are o vizibilitate mai bună a datelor introduse, a rezultatelor obținute dar și a etapelor ce trebuie parcurse pentru a obține cazuri de test lucruri ce pot fi observate în Fig. 5 de la pagina 24.

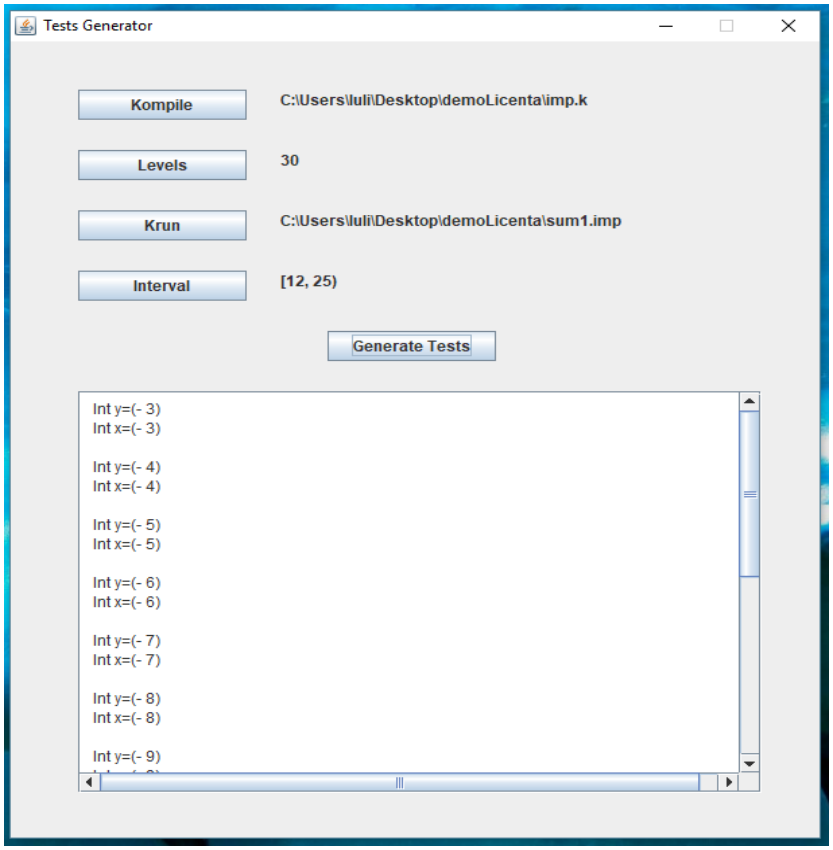

Fig. 5

Un alt avantaj este și faptul că utilizatorul nu poate omite etape în procesul de generare a cazurilor de test deoarece, dacă executarea unei instrucțiuni I2 necesită rezultatele obținute în urma executării altei instrucțiuni I1, butonul care pornește executarea instrucțiunii I2 va fi activat în momentul în care vor fi disponibile rezultatele oferite de instrucțiunea I1.

Inițial, la pornirea aplicației, va fi activat doar butonul *Kompile*, lucru ce poate fi observat în Fig. 6 de la pagina 25. Acest buton permite utilizatorului să aleagă un fișier *.k*, care să cuprindă definiția limbajului dorit, și să execute comanda *kompile* pentru fișierul ales.

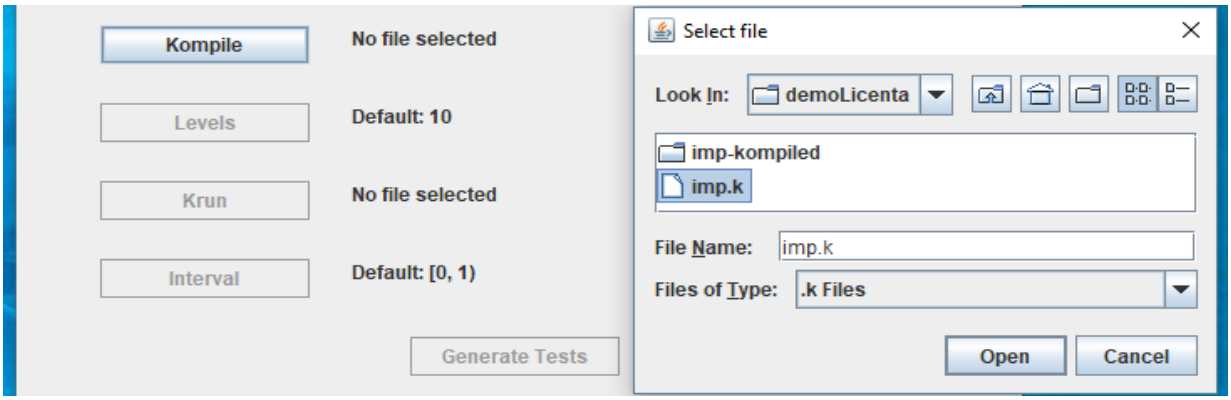

Fig. 6

Butonul *Levels* va fi activat în același timp cu butonul *Krun*, doar dacă execuția comenzii kompile, având ca parametru fișierul ales anterior, se va finaliza cu succes. Acest lucru poate fi observat în fereastra centrală din Fig. 7 de la pagina 25.

Butonul *Levels* permite utilizatorului să aleagă numărul maxim de șiruri de condiții care să fie construite și care vor fi necesare la generarea de cazuri de test. Dacă nu este aleasă nici o valoare, va rămâne valoarea default 10. Aceste lucruri pot fi observate în Fig. 7 de la pagina 25: în fereastra din centru, în dreapta butonului *Levels* este afișat *Default: 10* deoarece încă nu a fost ales alt număr, iar fereastra din partea stângă este pentru alegerea numărului dorit de șiruri de condiții care să fie construite.

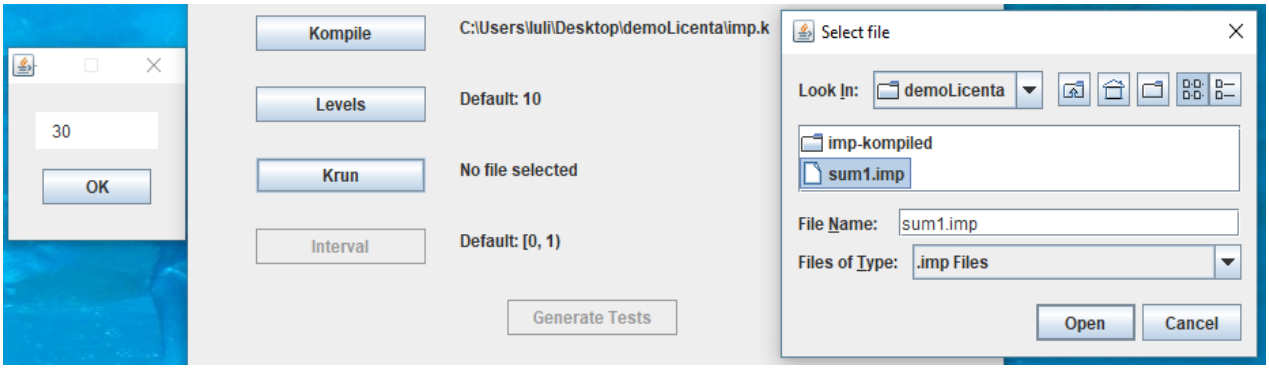

Fig. 7

Butonul *Krun* permite alegerea fișierului ce conține programul pentru care se dorește generarea de teste (scris în limbajul definit în fișierul compilat la pasul precedent), lucru care poate fi observat în fereastra din partea dreaptă din Fig. 6 de la pagina 22, și construirea unui număr de șiruri de condiții egal cu valoarea setată pentru Levels, șiruri care nu sunt vizibile dar sunt necesare la generarea soluțiilor.

Butoanele *Interval* și *Generate Tests* vor fi activate doar dacă comanda *krun*, executată cu parametrii aleși, nu va genera erori, permițând astfel trecerea la ultimii doi pași în procesul de generare de cazuri de test: setarea intervalului și generarea de teste. Acest lucru poate fi observat în Fig. 8 de la pagina 26.

Butonul *Interval* permite alegerea intervalului de șiruri de condiții, construite la pasul anterior, care să fie evaluate și din care să fie generate cazurile de test. Dacă intervalul nu este setat sau se vor alege valori incorecte atunci va rămâne intervalul default [0,1), lucru care se poate observa în Fig. 8 de la pagina 26.

Fig. 8

Apăsarea butonului *Generate Tests* va avea ca efect parsarea șirurilor de condiții din intervalul ales și generarea, din acestea, de cazuri de test așa cum se poate observa în Fig. 5 de la pagina 24.

5. Descrierea modului de implementare

Pentru realizarea proiectului a fost necesară împărțirea acestuia în trei module mari care să comunice între ele prin procese și apeluri de funcții. Schema aplicației poate fi observată în Fig. 9.

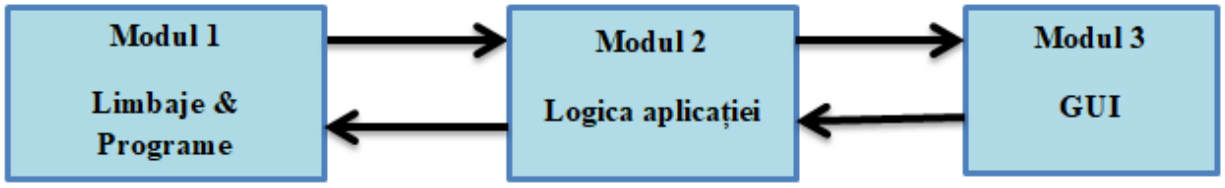

Fig. 9

- Primul modul cuprinde partea transformare a definițiilor unor limbaje, definite in KFramework, într-o manieră care să permită aplicarea execuției simbolice asupra programelor scrise in limbajul definit și scrierea unor exemple de programe în acesta.
- Al doilea modul este implementat in Java și este partea responsabilă de logica aplicației: lansarea, pe rând, a proceselor, preluarea și prelucrarea rezultatelor obținute în urma încheierii execuției acestora și furnizarea rezultatelor finale.
- Al treilea modul este interfața grafică, implementată în Java, gândită și construită pentru a simplifica utilizarea aplicației și pentru identificarea mai ușoara și rapidă a rezultatelor finale.

5.1. Modulul 1: Transformări făcute definițiilor limbajelor

Așa cum am menționat și în capitolul anterior, pentru a obține cazuri de test pornind de la definiția limbajelor, este necesară modificarea acestora din urmă astfel încât:

a) să poată recunoaște asignarea de valori simbolice variabilelor;

b) să permită efectuarea de operații aritmetice și logice cu valori simbolice;

c) să poată genera șirurile de condiții necesare pentru obținerea soluțiilor.

Pentru a putea urmări mai ușor modul în care sunt făcute aceste transformări voi lua ca exemplu definiția limbajului IMP, peste care voi efectua, pe rând, transformările necesare pentru a putea construi șiruri de condiții din care să poată fi generate cazuri de test. Definiția limbajului IMP cuprinde două module:

- un modul pentru sintaxa limbajului

```
module IMP-SYNTAX
  syntax AExp  ::= Int | Id
                 | AExp "/" AExp              [left, strict]
                 > AExp "+" AExp              [left, strict]
                 | "(" AExp ")"               [bracket]
  syntax BExp  ::= Bool
                 | AExp "<=" AExp             [seqstrict]
                 | "!" BExp                   [strict]
                 > BExp "&&" BExp             [left, strict(1)]
                 | "(" BExp ")"               [bracket]
  syntax Block ::= "{" "}"
                 | "{" Stmt "}"
  syntax Stmt  ::= Block
                 | Id "=" AExp ";"            [strict(2)]
                 | "if" "(" BExp ")"
                   Block "else" Block         [strict(1)]
                 | "while" "(" BExp ")" Block
                 > Stmt Stmt                  [left]
  syntax Pgm ::= "int" Ids ";" Stmt
  syntax Ids ::= List{Id,","}
endmodule
```

- un modul pentru semantica limbajului.

```
module IMP
  imports IMP-SYNTAX
  syntax KResult ::= Int | Bool
  configuration <T color="yellow">
                  <k color="green"> $PGM:Pgm </k>
                  <state color="red"> .Map </state>
                </T>
```

```
// AExp
  rule <k> X:Id => I ...</k> <state>... X |-> I ...</state>
  rule I1 / I2 => I1 /Int I2  requires I2 =/=Int 0
  rule I1 + I2 => I1 +Int I2
// BExp
  rule I1 <= I2 => I1 <=Int I2
  rule ! T => notBool T
  rule true && B => B
  rule false && _ => false
// Block
  rule {} => .   [structural]
  rule {S} => S  [structural]
// Stmt
  rule <k> X = I:Int; => . ...</k> <state>... X |-> (_ => I) ...</state>
  rule S1:Stmt S2:Stmt => S1 ~> S2  [structural]
  rule if (true)  S else _ => S
  rule if (false) _ else S => S
  rule while (B) S => if (B) {S while (B) S} else {}  [structural]
// Pgm
  rule <k> int (X,Xs => Xs);_ </k> <state> Rho:Map (.Map => X|->0) </state>
    requires notBool (X in keys(Rho))
  rule int .Ids; S => S  [structural]
// verification ids
  syntax Id ::= "n"     [token]
              | "sum"   [token]
endmodule
```

Recunoașterea asignării de valori simbolice se face prin:

- adăugarea a două noi categorii sintactice pentru recunoașterea unor noi tipuri de date:

```
syntax SymInt ::= "symInt" "(" Id ")" | Int
syntax SymBool ::= "True" | "symBool" "(" Id ")" | Bool
```

- modificarea categoriilor sintactice `AExp` și `BExp` astfel:

```
syntax AExp  ::= Int | Id
```

devine

```
syntax AExp  ::= SymInt | Id
```

iar

```
syntax BExp  ::= Bool
```

devine

```
syntax BExp  ::= SymBool
```

- modificarea tipurilor de date ce se doresc a fi obținute astfel:

```
syntax KResult ::= Int | Bool
```

devine

```
syntax KResult ::= SymInt | SymBool
```

- modificarea regulii pentru asignare astfel:

```
rule <k> X = I:Int; => . ...</k>
     <state>... X |-> (_ => I) ...</state>
```

devine

```
rule <k> X = I:SymInt; => . ...</k>
     <state>... X |-> (_ => I) ...</state>
```

Pentru ca limbajul definit să permită efectuarea de operații cu valori simbolice sunt necesare:

- extinderea categoriilor sintactice `SymInt` și `SymBool` prin adăugarea de simboluri pentru operațiile ce se doresc a fi efectuate cu aceste tipuri de date:

```
syntax SymInt ::= "symInt" "(" Id ")" | Int
                | SymInt "/sym"         SymInt [left]
                | SymInt "+sym"         SymInt [left]
                | "("             SymInt      [bracket[bracket]]
syntax SymBool ::= "True" | "symBool" "(" Id ")" | Bool
```

```
                    | SymInt "<=sym" SymInt
                    | "not" SymBool
                   | SymBool "and" SymBool
                    | SymBool "or" SymBool
                    | "("            SymBool ")" [bracket]
```

- adăugarea de reguli pentru operatorii definiți mai sus:

```
rule I1 / I2 => I1 /sym I2  requires I2 =/=Int 0
rule I1 + I2 => I1 +sym I2
rule I1 <= I2 => I1 <=sym I2
rule ! T => not T
rule B1 && B2 => B1 and B2
rule B1 || B2 => B1 or B2
```

Pentru ca din limbajul definit să poată fi construite șiruri de condiții sunt necesare următoarele:

- crearea unei noi celule computaționale în care vor fi colectate condițiile în care inițial se va găsi valoarea True:

```
<pc> True </pc>
```

- modificarea regulilor pentru instrucțiunea *if* astfel încât odată cu evaluarea condițiilor, acestea să poată fi colectate în celula <pc></pc>. Astfel:

```
rule if (true)  S else _ => S
```

devine

```
rule <k> (if (B)  S else _ => S) ...</k>
       <pc> P => (P and B) </pc> [transition]
```

iar

```
rule if (false) _ else S => S
```

devine

```
rule <k> if (B) _ else S => S ...</k>
       <pc> P => (P and not B) </pc> [transition]
```

5.2. Modulul 2: Obținerea soluțiilor

Pentru a genera cazuri de test este necesară parcurgerea mai multor pași:

a) Compilarea fișierului ce conține definiția limbajelor și transformările descrise în subcapitolul anterior;

b) Rularea programului sau programelor scrise în limbajul compilat la pasul anterior și construirea de șiruri de condiții ;

c) Preluarea, prelucrarea și transformarea șirurilor de condiții în formule ce pot fi rezolvate de Z3;

d) Rezolvarea de Z3 a formulelor construite la pasul anterior și generarea rezultatelor;

e) Preluarea rezultatelor și interpretarea acestora.

Pentru a lega pașii enumerați anterior între ei am implementat doua clase în Java:

- SolutionsGenerator
- PcToSmtTranslator

Clasa *SolutionsGenerator* este clasa care asigură funcționalitatea necesară pașilor a), b), d) și e) prin cele patru funcții implementate în interiorul acesteia:

- *public boolean kompile(String kompileFilePath)*: este funcția ce corespunde punctului a). Aceasta primește ca parametru un string ce reprezintă calea către fișierul ”.k” ce conține definiția limbajului țintă, definiție asupra căreia au fost făcute transformările explicate în subcapitolul anterior. În interiorul funcției se creează un proces în urma executării comenzii *”kompile.bat kompileFilePath”*, comandă necesară compilării fișierului dat. Apoi se verifică dacă procesul s-a încheiat cu succes. Dacă procesul a generat un cod de eroare sau, una din instrucțiuni a generat o excepție, funcția va returna false. Altfel se va considera că totul a decurs bine și funcția va returna true.

- *public boolean krun(String kompileDirectoryPath, String krunFilePath, int levels)*: este funcția ce corespunde punctului b). Aceasta primește ca parametri: calea directorului în care se află fișierul compilat, calea fișierului ce conține programul scris în limbajul definit în fișierul compilat și numărul maxim de șiruri de condiții pe baza cărora vor fi generate cazurile de test. În interiorul funcției se creează un proces în urma executării comenzii: *"krun.bat –directory kompileDirectoryPath krunFilePath –search –bound levels"*, unde opțiunea *--directory* urmată de directorul fișierului compilat indică locul de unde este luată definiția limbajului, opțiunea *–search* este utilizată pentru a căuta și construi șiruri de condiții iar opțiunea *–bound* este necesară pentru a construi un număr limitat de șiruri de condiții deoarece, fără această limitare, procesul tinde să se execute la infinit. Dacă procesul returnează un cod de eroare sau una din instrucțiuni va genera o excepție, funcția va returna false. Altfel șirurile de condiții vor fi reținute, sub formă de stringuri, într-un array.

- *private String callZ3(String formula)*: este funcția ce corespunde punctului d). Aceasta primește ca parametru un string ce reprezintă formula obținută din transformarea unui șir de condiții. Este *private* deoarece apelarea ei nu se face înafara clasei. În urma execuției funcției se creează fișier de tipul ".smt2", în care este scrisă formula primită ca parametru. Apoi se creează un proces în urma executării comenzii *"z3 tempFile.smt2"*. Se verifică rezultatul obținut. Dacă rezultatul obținut este *"unsat"* rezultă că formula evaluată este nesatisfiabilă, deci nu există soluții, iar funcția va returna stringul *"No solutions"*. Altfel rezultatul obținut va fi *"sat"* urmat de soluțiile generate. În acest caz, soluțiile sunt preluate sub formă de string și returnate.

- *public String buildFinalSolution(int index)*: este funcția ce corespunde punctului e). Aceasta primește ca parametru un int ce indică indexul ce corespunde șirului de condiții, din array-un în care au fost reținute acestea, pentru care se dorește încercarea de generare de cazuri de test. În cadrul acestei funcții se preia șirul de condiții, se creează un obiect de tipul PcToSmtTranslator prin intermediul căruia șirul va fi transformat într-o formulă, se apelează funcția *callZ3* având ca

parametru formula obținută, se preia rezultatul returnat de funcția *callZ3* și se returnează ca rezultat final pentru șirul de condiții solicitat.

Clasa *PcToSmtTranslator*, așa cum îi spune și numele, este clasa care transformă un string ce reprezintă un șir de condiții, încadrat între tagurile *<pc>* și *</pc>*, într-o formulă ce poate fi evaluată de Z3. Aceste transformări sunt posibile datorită constructorului cu parametru de tip string și celor patru funcții ce aparțin clasei:

- *public PcToSmtTranslator(String pathCondition)*: este constructorul definit al clasei cu același nume, ce primește ca parametru un string ce reprezintă un șir de condiții încadrat între tagurile *<pc>* și *</pc>*. În cadrul constructorului sunt inițializate mai multe variabile necesare parsării de stringuri din cadrul funcțiilor. Tot în constructor sunt eliminate spațiile de la începutul și sfârșitul stringului primit ca parametru iar spațiile consecutive din cadrul acestuia sunt înlocuite cu un singur spațiu, cu scopul de a reduce complexitatea și timpul de execuție al funcțiilor din cadrul clasei responsabile cu parsarea de stringuri.

- *private void translate(String text, String node)*: este funcția responsabilă cu preluarea și parsarea stringului dat ca parametru ce reprezintă șirul de condiții și construirea unui arbore binar în care nodurile sunt perechi de tipul cheie-valoare, unde cheia este operația (sub formă de string) din acel nod, iar valoarea este un sir de 0 și/sau 1 (tot sub formă de string), ce indică prioritatea efectuării operației din acel nod. Acest arbore este reținut într-un *Map<String,String>*. De exemplu pentru șirul de condiții:

```
<pc>(True and ((symInt(x) <sym 10) and (symInt(y) <sym
10)) and not (((symInt(x) +sym 1) <sym 10) and ((symInt(y) -
sym 1) <sym 10))</pc>
```

reprezentarea grafica a arborelui obținut este cea din Fig. 10

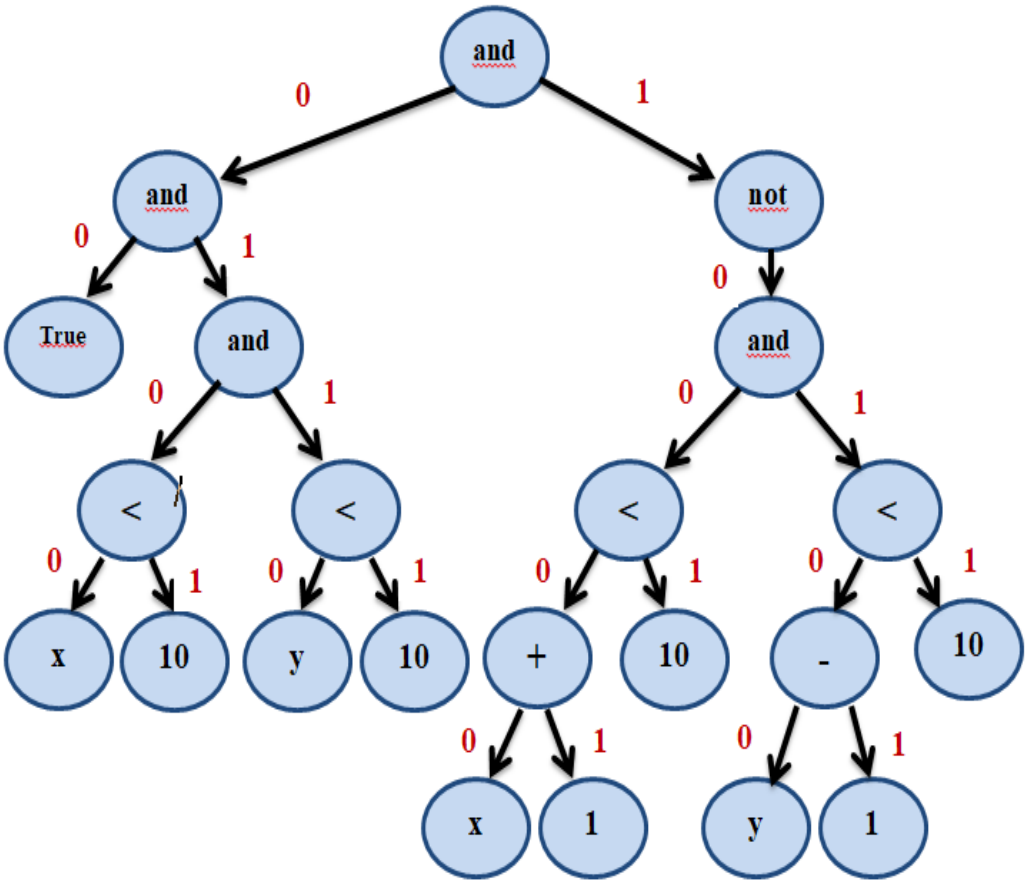

Fig. 10

- *private String buildFormula(String node)*: este funcția ce construiește recursiv, plecând de la arborele construit de funcția *translate*, formula de forma (*operator operand1 operand2)*, unde operand1 și operand2 pot fi subformule de aceeași formă. Prioritatea operatorilor e dată de valoarea corespunzătoarea cheii din nodul în care se află. Funcția returnează formula construită, sub formă de string.

- *private void buildFormulaForZ3 ()*: este funcția ce transformă formula obținută din șirul de formule, prin apelul (pe rând) a funcțiilor *translate* și *buildFormula*, într-o formulă ce poate fi evaluată de Z3. Formula rezultată este reținută într-o variabilă de tip String.

- *public String getFormulaForZ3()*: este singura funcție publică din cadrul acestei clase, deci singura funcție ce poate fi apelată din exteriorul clasei. În cadrul

acesteia este apelată funcția *buildFormulaForZ3* pentru a fi construită formula ce poate fi evaluată de Z3. Apoi această formulă este returnată sub formă de string.

5.3. Modulul 3: Interfața grafică

Proiectul final este funcțional și fără interfața grafică dar aceasta presupune introducerea de la tastatură a mai multor parametri (căi ale unor fișiere) și valori, lucru care ar duce la apariția frecventă de erori (de exemplu un caracter scris greșit când introducem calea către un fișier) si care ar îngreuna modul de utilizare a aplicației. Pe lângă acestea, dacă am utiliza aplicația fără interfața grafică, rezultatele ar fi mai greu de observat în consolă iar la fiecare pas ar fi necesară afișarea unui meniu din care utilizatorul sa aleagă o opțiune și să introducă de la tastatură valoarea asociată acelei opțiuni, așa cum se poate observa în Fig. 11.

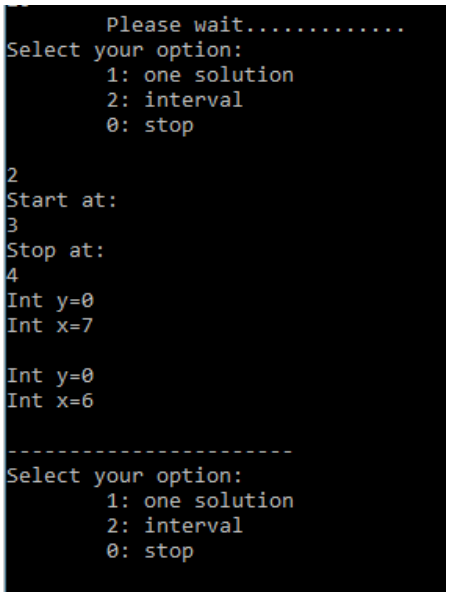

Fig. 11

Așa cum am explicat în capitolul anterior, interfața grafică a fost implementată cu scopul de a evita apariția unor erori cauzate de introducerea greșită, de la tastatură, a unor valori și căi către anumite fișiere. De asemenea, prin intermediul acesteia, utilizatorul nu mai trebuie să introducă de la tastatură calea către fișierul dorit ci, mai simplu, trebuie doar să îl selecteze dintr-o fereastră. Pe lângă acestea, din interfață, utilizatorul are acces permanent la un meniu pentru setarea preferințelor.

Interfața este construită din trei clase:

- *TestGeneratorFrame*
- *KFileChooser*
- *LimitFrame*

Clasa *KFileChooser* este clasa care construiește fereastra ce dă posibilitatea utilizatorului să aleagă fișierul dorit. În interiorul clasei este creat un obiect de tipul JFileChooser căruia i se aplică un filtru, impunând utilizatorului să aleagă doar un anumit tip de fișier. Această clasă are implementate trei funcții prin intermediul cărora sunt preluate anumite informații despre fișierul selectat:

- *public String getFileName()*: returnează numele fișierului selectat, fără calea către acesta;
- *public String getFilePath()*: returnează calea absolută către fișierul selectat;
- *public String getParentDirectory()*: returnează calea absolută către directorul în care se află fișierul selectat.

Clasa *LimitFrame* este clasa care construiește fereastra ce dă utilizatorului posibilitatea de a schimba anumite valori default. Parametrul constructorului indică numărul de valori ce pot fi setate de utilizator la pasul curent. Astfel, dacă valoarea parametrului este 1, utilizatorul va avea posibilitatea de a seta o valoare. Dacă valoarea parametrului este 2 atunci utilizatorul va avea posibilitatea de a seta două valori.

Clasa *TestGeneratorFrame* este clasa ce construiește fereastra principală (Fig. 5 de la pagina 24). În interiorul acestei clase este creat un frame de tipul JFrame în care, prin intermediul funcției *initMainFrame()* sunt adăugate cinci butoane de tipul JButton, patru labeluri de tipul JLabel și un textArea de tipul JTextArea.

Poziționarea elementelor în frame este stabilită prin intermediul funcției *setElementsPositions()*. Astfel, patru butoane sunt pozitionate unele sub altele, în partea stângă, sus. Prin intermediu acestora utilizatorul setează parametrii necesari executării comenzilor

kompile și krun. În partea dreaptă a fiecărui buton este adăugat câte un label în care este afișată valoarea parametrului setat de utilizator prin acționarea butonului corespunzător labelului. Al cincilea buton este pozitionat, la mijloc, sub cele patru labeluri și cele patru butoane. În partea de jos a ferestrei, sub al cincilea buton, este poziționat textArea.

Evenimentele produse la acționarea butoanelor sunt setate prin intermediul funcției *addButtonsActions()* astfel:

- La apăsarea butonului *Kompile* este creat un obiect de tip KFileChooser care va avea ca efect vizual deschiderea unei ferestre prin intermediul căreia, utilizatorul alege fișierul .k pe care dorește să îl compileze. După ce fișierul a fost ales este apelată funcția *kompile(filePath)* în interiorul căreia se executată comanda:

  kompile.bat filePath

  Dacă procesul generează un cod de eroare sau dacă va fi aruncată o excepție, funcția kompile(filePath) va returna false. Altfel va returna true, iar în labelul din dreapta butonului va fi afișată calea absolută către fișierul selectat. Într-o variabilă va fi reținută calea absolută a directorului în care se află fișierul selectat.

  Dacă fișierul selectat este valid atunci vor fi activate butoanele *Levels* și *Krun*. Altfel, toate butoanele, cu excepția butonului *Kompile,* vor rămâne dezactivate până când va fi ales un fișier valid, care va fi compilat cu succes.

- La apăsarea butonului *Levels* se creează un obiect de tipul *LimitFrame(1)* prin care se solicită utilizatorului să furnizeze o valoare egală cu numărul maxim de șiruri de condiții care să fie construite. Valoarea default este 10. Condițiile de validare a valorii oferite de utilizator sunt: să fie număr întreg, pozitiv și nenul. Dacă utilizatorul încearcă sa introducă o valoare ce nu îndeplinește criteriile de validare, limita setată pentru numărul de șiruri de condiții va rămâne default 10. Cu cât numărul solicitat este mai mare cu atât mai mult va dura furnizarea rezultatelor.

- Modul de funcționare a butonului *Krun* este asemănător cu cel a butonului *Kompile*: la acționarea butonului, se creează un obiect de tipul KFileChooser, care va avea ca efect vizual deschiderea unei ferestre prin intermediul căreia,

utilizatorul alege fișierul pe care dorește să îl ruleze. După ce fișierul a fost ales este apelată funcția *krun(kompiledFileDirectory, krunFilePath, levels)* în interiorul căreia se executată comanda:

krun.bat –directory kompiledFileDirectory krunFilePath –search –bound levels

Dacă procesul generează un cod de eroare sau dacă va fi aruncată o excepție, funcția krun(kompiledFileDirectory, krunFilePath, levels) va returna false. Altfel va returna true, iar în labelul din dreapta butonului va fi afișată calea absolută către fișierul selectat.

Dacă fișierul selectat este valid atunci vor fi memorate, într-o listă, șirurile de condiții rezultate în urma executării comenzii și vor fi activate butoanele Interval și Generate Tests. Altfel, lista va rămâne vidă iar cele două butoane vor rămâne dezactivate până când va fi ales un fișier valid, care va fi rulat cu succes.

- La apăsarea butonului *Interval* se creează un obiect de tipul *LimitFrame(2)* prin care se solicită utilizatorului să furnizeze două valori ce vor fi considerate a fi capetele unui interval închis la stânga și deschis la dreapta. Condițiile de validare a valorilor oferite de utilizator sunt: să fie numere întregi, pozitive, nenule. Prima valoare, fiind considerată capătul inferior al intervalului, trebuie să fie strict mai mică decât a doua și strict mai mică decât numărul de șiruri de condiții construite. A doua valoare, fiind considerată capătul superior al intervalului, trebuie să fie strict mai mare decât prima și cel mult egală cu numărul de șiruri de condiții construite. Intervalul default este [0, 1). Dacă utilizatorul încearcă sa introducă, pentru unul din capetele intervalului sau pentru ambele capete, valori ce nu îndeplinesc criteriile de validare, atunci se vor păstra valorile default: 0 pentru limita inferioară și 1 pentru limita superioară. Cu cât intervalul solicitat este mai mare cu atât mai mult va dura furnizarea rezultatelor.

- La apăsarea butonului *Generate Tests*, din lista de șiruri de condiții, vor fi obținute rezultate, prin apelarea funcției *buildFinalSolution(index)*, pentru acele șiruri de condiții a căror index este cuprins în intervalul ales de utilizator. Soluțiile astfel

obținute sunt concatenate într-un string și afișate pentru a fi vizuzlizate de utilizator.

Un lucru care ar trebui precizat este faptul că schimbarea preferințelor, la un anumit pas determină resetarea celor de la pașii următori, utilizatorul fiind astfel obligat să parcurgă din nou următorii pași pentru a obține rezultatele dorite.

6. Experimente

Pentru a testa aplicația dezvoltată am utilizat două limbaje: IMP și KOOL, disponibile în setul de exemple de limbaje cu definiții nu foarte complexe, scrise în K Framework, disponibile la adresa [11].

6.1. Experimente efectuate pe IMP

IMP [12] este un limbaj simplu, ce are definite:

- Operații aritmetice: adunarea și împărțirea;
- Operații booleene: comparația <=, && (și logic), negația;
- Instrucțiuni: atribuire, declarare variabile, if și while;
- Lucrul cu blocuri de instrucțiuni.

Pentru a putea lucra cu toate operațiile aritmetice și booleene de bază, am adăugat la definiția limbajului IMP:

- Operațiile aritmetice: scăderea, înmulțirea și modulo;
- Operațiile booleene: comparație (<, >, >=, ==), || (sau logic).

Următorul pas a fost aplicarea transformărilor descrise în Secțiunea 5.1 asupra limbajului IMP (pentru ca acesta să recunoască lucrul cu valori simbolice) și compilarea acestuia.

În limbajul definit și compilat am luat, pe lângă alte exemple în care am utilizat și valori simbolice, și următorul program:

```
int n, sum, m, prod;
n = symInt(x);
m = symInt(y);
sum = 0;
```

```
prod = m;
while ((n < 10) && (m < 20)) {
        sum = sum + n;
        n = n + 1;
        prod = prod * m;
        m = m + 1; }
```

La rularea programului am obținut șiruri de condiții de forma:

<pc> (True and ((symInt (x) <sym 10) and (symInt (y) <sym 20))) and not (((symInt (x) +sym 1) <sym 10) and ((symInt (y) +sym 1) <sym 20)) </pc>

a căror complexitate diferă în funcție de numărul de condiții ce trebuie evaluate.

Am translatat aceste șiruri de condiții în formule de forma:

```
(declare-fun x () Int)
(declare-fun y () Int)
(assert (and (and true (and (< x 10) (< y 20))) (not (and (< (+ x 1) 10) (< (+ y 1) 20)))))
(check-sat)
(get-model)
```

care au fost evaluate de Z3. În cele mai multe cazuri, rezultatele au fost de forma:

```
sat
(model
  (define-fun y () Int
    0)
  (define-fun x () Int
    9)
)
```

ceea ce înseamnă că formula evaluată este satisfiabilă și sunt oferite și valori posibile ce pot înlocui cu succes valorile simbolice. În acest caz valorile posibile sunt: *y = 0 și x = 9*.

Foarte rare au fost cazurile în care, în urma evaluării formulei de către Z3, am obținut rezultate de forma:

unsat

ceea ce înseamnă că formula nu este satisfiabilă, deci șirul de condiții din care a fost obținută formula nu poate fi niciodată îndeplinit.

6.2. Experimente efectuate pe KOOL

KOOL [13] este un limbaj orientat obiect definit în K Framework.

Deoarece nu a mai fost necesară adăugarea de operații suplimentare, ca în cazul IMP, am trecut direct la aplicarea transformărilor descrise în Secțiunea 5.1 asupra definiției limbajului KOOL. După finalizarea aplicării transformărilor, am compilat definiția obținută.

În limbajul astfel obținut am luat ca exemplu următorul program:

```
class Main {
        method f(x) {
                if (x <= 1) { return 1; }
                else { return x * f(x - 1); }
        }
        method Main() {
                print(f(f(f(symInt(y)))), "\n");
        }
}
```

Complexitatea acestuia este puțin mai mare comparativ cu exemplele scrise în limbajul IMP deoarece sunt utilizate clase, funcții și apelul recursiv al unui funcții.

La rularea programului am obținut șiruri de condiții de forma:

<pc> ((True and (symInt (y) <=sym 1)) and not (1 <=sym 1)) and ((1 -sym 1) <=sym 1) </pc>

a căror complexitate crește odată cu numărul de condiții ce trebuie evaluate.

Am translatat aceste șiruri de condiții în formule de forma:

```
(declare-fun y () Int)
(assert (and (and (and (and true (not (<= y 1))) (<= (- y 1) 1)) (not (<= (* y 1) 1))) (<= (- (* y 1) 1) 1)))
(check-sat)
(get-model)
```

care au fost evaluate de Z3. În foarte puține cazuri rezultatul a fost de forma:

```
sat
(model
  (define-fun y () Int
    2)
)
```

ceea ce înseamnă că formula evaluată este satisfiabilă și sunt oferite și valori posibile ce pot înlocui cu succes valorile simbolice. În acest caz valoarea posibilă este: *y = 2*.

Cele mai frecvente au fost cazurile în care, în urma evaluării formulei de către Z3, am obținut rezultate de forma:

```
unsat
```

ceea ce înseamnă că formula nu este satisfiabilă, deci șirul de condiții din care a fost obținută formula nu poate fi niciodată îndeplinit.

7. Concluzii

Așa cum am menționat și în introducere, scopul acestei lucrări a fost oferirea unei soluții care să reducă timpul și volumul de muncă necesar găsirii de cazuri de test. Am propus ca alternativă la soluțiile deja existente, generarea automată de cazuri de test utilizând execuția simbolică, pornind de la definiția limbajelor.

În dezvoltarea soluției am pornit de la definițiile limbajelor scrise în K Framework, peste care am aplicat câteva modificări pentru a putea lucra cu valori simbolice, oferind astfel o soluție independentă de limbaj. Șirurile de condiții construite le-am translatat în formule care să poată fi evaluate de Z3. Rezultatele obținute în urma evaluării formulelor de către Z3 le-am interpretat, obținând astfel soluțiile dorite.

Deoarece soluția a fost aplicată doar asupra unor limbaje mai simple, ca posibilități de dezvoltare a acesteia ar fi: aplicarea soluției asupra unor limbaje complexe cum ar fi Java sau C și creșterea complexității programelor și a condițiilor ce implică utilizarea valorilor simbolice.

8. Bibliografie

[1] RTI, „The Economic Impacts of Inadequate Infrastructure for Software Testing; Final Report," U.S Department of Commerce, Research Triangle Park, 2002.

[2] R. Patton, Software Testing, Indianapolis, Indiana: Sams, 2001, pp. 10-11, 13.

[3] P. Ammann și J. Offutt, Introduction to software testing, New York: Cambridge University Press, 2008, p. 11.

[4] P. Triparth și . K. Naik, SOFTWARE TESTING AND QUALITY ASSURANCE Theory and Practice, Hoboken, New Jersey: John Wiley & Sons, Inc., 2008, pp. 9, 16.

[5] [Interactiv]. Available: http://www.embunit.com/. [Accesat 24 January 2018].

[6] „Documentation: JUnit," [Interactiv]. Available: http://junit-tools.org/index.php/documentation. [Accesat 24 January 2018].

[7] [Interactiv]. Available: http://webdesignmoo.com/web-design/10-best-affordable-java-tools-for-programmers-you-may-have-not-heard-of. [Accesat 06 02 2018].

[8] G. Roșu și T. F. Șerbănuță, „An overview of the K semantic framework," *Journal of Logic and Algebraic Programming,* vol. 79(6), pp. 397-434, 2010.

[9] A. Arusoaie, "A generic framework for symbolic execution: Theory and applications," Ph.D. dissertation, Faculty of Computer Science, Alexandru I. Cuza, University of Iasi, https://fmse.info.uaic.ro/publications/193/, sep 2014, pp. 54, 65-70.

[10] „Z3 tutorial," [Interactiv]. Available: https://rise4fun.com/Z3/tutorial/guide. [Accesat 24 January 2018].

[11] „K Framework: tutorial," [Interactiv]. Available: https://github.com/kframework/k/tree/master/k-distribution/tutorial. [Accesat 28 01 2018].

[12] „K Framework: IMP,” [Interactiv]. Available: https://github.com/kframework/k/blob/master/k-distribution/tutorial/1_k/2_imp/lesson_4/imp.k. [Accesat 01 02 2018].

[13] „K Framework: KOOL,” [Interactiv]. Available: https://github.com/kframework/k/blob/master/k-distribution/tutorial/2_languages/2_kool/1_untyped/kool-untyped.k. [Accesat 01 02 2018].