Testiranje funkcionalnosti i robusnosti programskog koda metodama klasa ekvivalencije i graničnih vrednosti

Zlatko Veličković, Darko Tasovac, Lazar Saranovac

Apstrakt—**U radu je predstavljen sistem za testiranje funkcionalnosti i robusnosti programskog koda. Testiranje se vrši konvencionalnim metodama *Equivalence Partitioning* (EP) i *Boundary Value Analysis* (BVA), koje su nezavisne od platforme, tehnologije, namene ili jezika u kome je pisan programski kod. Kao alat za ekstrakciju parametara funkcija i integraciju odgovarajućeg kompajlera i *debugger*-a korišćeno je okruženje za testiranje pretežno *embedded* softvera napisanog u C-u ili C++ - u.**

Ključne reči—**Testiranje softvera, *Equivalence Partitioning*, EP, *Boundary Value Analysis*, BVA, kompajler, debugger, *embedded,* industrijski softver.**

I. Uvod

Tokom godina, sve je veće interesovanje za sisteme koji testiraju funkcionalnost napisanog programskog koda za određenu namenu, kako bi se eventualne greške uočile, a zatim i ispravile. Pored tačnosti, veliki imperativ u današnje vreme predstavljaju efikasnost i robusnost: da li je programski kod sistematično napisan, da li se dovoljno brzo izvršava, da li ostaje zaglavljen u beskonačnim petljama, itd.

Za tu namenu, neophodno je da se testiranje zasniva prvenstveno na razumevanju načina razmišljanja dizajnera koda i specifikacijama projekta, tj. kakav je algoritam tim kodom implementiran. Zadaju se ulazi, i, na osnovu samog algoritma, određuju se očekivani izlazi koje bi kod (funkcija) nakon kompajliranja trebalo da vrati. Analizira se pokrivenost koda, tj. da li postoje određeni segmenti programskog koda u koje program nikada ne ulazi, za bilo koju kombinaciju ulaza, tj. stimulusa.

Postoje dva moda testiranja: regularni mod i testiranje robusnosti. Regularnim modom testiranja ispituje se funkcionalnost programskog koda u normalnim uslovima, tj. kada stimulusi odgovaraju propisanom opsegu važenja. U testiranju robusnosti analizira se uticaj nevalidnih vrednosti stimulusa na izvršavanje programskog koda i na vrednosti koje odgovarajuća funkcija vraća. Na taj način se može kompletno analizirati programski kod, a njegovo izvršavanje verno simulirati odgovarajućim *debugger*-om pre hardverske implementacije.

U ovom radu, predstavljen je sistem koji je baziran na metodi klasa ekvivalencije (*Equivalence Partitioning*) i metodom graničnih vrednosti (*Boundary Value Analysis*) za testiranje i analiziranje rada programskog koda, koji je za korišćenje pretežno namenjen manjim i većim kompanijama u industrijskom sektoru, zarad testiranja postojećih implementiranih, kao i još uvek neimplementiranih softverskih modula.

Rad je organizovan na sledeći način: poglavlje 2 opisuje tehniku klasa ekvivalencije, poglavlje 3 tehniku graničnih vrednosti, poglavlje 4 opisuje sistem korišćen za testiranje i rezultate testiranja. Najzad, u poglavlju 5 je dat zaključak rada.

Zlatko Veličković – NovelIC Microsystems, Veljka Dugoševića 54, 11000 Beograd, Srbija (e-mail: zlatko.velickovic@novelic.com).

Darko Tasovac – NovelIC Microsystems, Veljka Dugoševića 54, 11000 Beograd, Srbija (e-mail: darko.tasovac@novelic.com).

dr Lazar Saranovac – Elektrotehnički fakultet, Univerzitet u Beogradu, Bulevar Kralja Aleksandra 73, 11020 Beograd, Srbija (e-mail: laza@el.etf.bg.ac.rs).

II. Metoda klasa ekvivalencije

Metoda klasa ekvivalencije (*Equivalence Partitioning*) [1] je metoda koja se zasniva na razdvajanju stimulusa funkcije u zasebne, ekvivalentne particije, tj. skupove odgovarajućih vrednosti. Svaku particiju je dovoljno pokriti jednim test slučajem koji reprezentuje jednu vrednost stimulusa, a koji se nalazi u okviru granica particije [2].

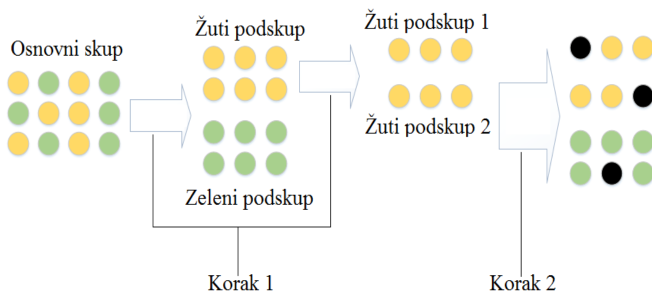
Za primenu ove metode važi i pretpostavka da se stimulusi jedne particije ponašaju slično. Dakle, ako je vrednost ulaza jedne particije dovela do greške u funkciji, podrazumeva se da će i ostale vrednosti ulaza iz te iste particije dovesti do greške. Takve particije zovemo homogenim [3-4]. Na taj način, preklapanje vrednosti dveju ili više particija nije dozvoljeno, jer gorepomenuta pretpostavka ne bi bila ispunjena u slučaju heterogenih particija.

Primena ove metode se sastoji iz dva koraka: (1) identifikacija particija i (2) definisanje test slučaja. Identifikacija, a samim tim i primena metode, je moguća samo ako se može napraviti najmanje dve particije stimulusa. Postoji dva tipa particija (klasa) ekvivalencije: validne klase (reprezentuju validne ulaze funkcije) i nevalidne klase (reprezentuju nevalidne ulaze funkcije, tj. one ulaze za koje se dešava greška u funkciji). Distinkcija između ova dva osnovna tipa klasa se vrši prvenstveno na osnovu specifikacija, tj. uslova za odgovarajuće promenljive funkcije. Takođe, distinkcija veoma zavisi i od izabranog tipa promenljive, pa npr. kod nabrojivog tipa (*enum*) u programskom jeziku C u validnu klasu potpadaju samo definisane vrednosti nabrojivog tipa, a u nevalidnu sve ostale vrednosti. Ako postoji i najmanji razlog da program ne procesira sve elemente jedne klase na isti način, tada klasu treba podeliti na više manjih. [4]

Drugi korak u testiranju jeste korišćenje klasa, tj. particija iz prvog koraka, radi određivanja test slučaja. Ukoliko je prvi

korak precizno urađen, tj. ne postoji više ni jedan razlog za podelu klasa na manje, za testiranje funkcionalnosti programa dovoljno je za svaku klasu nameniti jedan test slučaj stimulusa i očekivanih izlaza (što je zapravo i hipoteza EP testiranja). Ta vrednost stimulusa može se nalaziti bilo na početku, bilo na kraju opsega važenja, bilo negde između krajnjih vrednosti. Postoji više konvencionalno korišćenih metoda za određivanje vrednosti promenljive, koje su opisane u odgovarajućoj literaturi [5-9].

Na slici 1 se nalazi ilustracija EP algoritma. U prvom koraku je prvo grubo podeljen osnovni skup na dve particije, da bi se kasnije jedna od te dve particije, koja nije homogena, podelila na dve homogene particije. Tako, u drugom koraku postoje 3 particije, i po jedna vrednost iz svake koju treba obuhvatiti. Može se videti da su proizvoljno selektovane vrednosti iz svake particije obojene crnom bojom, nakon drugog koraka algoritma.



Sl. 1. Ilustracija *Equivalence Partitioning* postupka

III. Metoda graničnih vrednosti

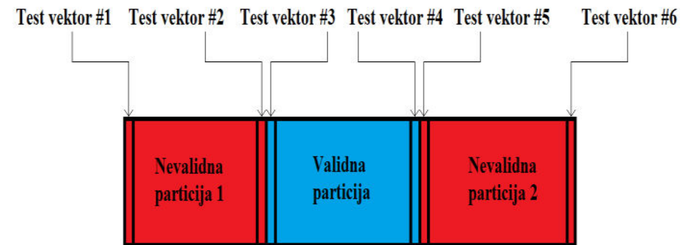
Metoda graničnih vrednosti (*Boundary Value Analysis*) [2] je metoda za testiranje softvera u kojoj su test slučajevi napravljeni tako da reprezentuju granice odgovarajućih particija, tj. klasa. Osnovu čini princip prethodno objašnjene EP metode, a podrazumeva se da su particije susedne, te da postoji izvesna "granica" između njih. Test vektori na obe strane granice se nazivaju graničnim vrednostima particije i oni čine osnovu ove metode testiranja [10-11].

Dakle, vrednosti na donjoj i gornjoj granici određene particije (klase) ekvivalencije su test vektori. Razlog za korišćenje ovih graničnih vrednosti za test vektore jesu česte greške softvera baš na granicama opsega važenja. Hipoteza ove metode testiranja kaže da je dovoljno uzeti gornje i donje granice particije za test slučajeve kako bi se testirala funkcionalnost algoritma opisanog odgovarajućom funkcijom [11].

Ova tehnika za testiranje softvera je više nego dvostruko efikasnija od EP tehnike [2]. Ono što je nedostatak BVA jeste da je neophodno značajno više vremena za generisanje očekivanih izlaza nekom od prethodno pomenutih konvencionalnih metoda za to korišćenih.

Kao jednostavan primer za ilustraciju BVA metoda testiranja, uzeta je funkcija zatvorenog tipa (crna kutija, tj. *black*-box) o kojoj se zna samo ograničenje argumenta. Argument je neoznačenog celobrojnog tipa, predstavljen sa 8 bita, a označava redni broj meseca u godini. Tako, vrednosti test vektora bi bile: 0 za testiranje prve nevalidne particije (trebalo bi da postoje dva test vektora, jedan za donju granicu tipa promenljive, a drugi za gornju granicu te particije – ali je to u ovom slučaju ista vrednost jer je argument neoznačen broj), 13 i 255 za testiranje dve nevalidne particije, tj. greške do koje dolazi istupanjem iz zadatog opsega, i 1 i 12 za testiranje jedine validne particije argumenta funkcije.

Na slici 2 se može videti ilustracija selekcije test vektora, tako da je jasno uočljivo da su test vektori (reprezenti test slučajeva) uzeti na gornjoj i donjoj granici svake particije.



Sl. 2. Ilustracija *Boundary Value Analysis* selekcije

IV. Sistem za testiranje softvera

Softver koji je testiran je pretežno industrijski, napisan za implementaciju na unapred određenom mikrokontroleru i odgovarajućoj platformi, sa unapred definisanim kompajlerom koji bi se koristio. Sistem koji je realizovan se sastoji iz više celina, i to:

- **Kompajler**, koji analizira i prevodi kod iz višeg u niži programski jezik i tako sintetiše objektni kod, zajedno sa linkerom koji od objektnog koda pravi izvršni fajl.
- **Simulator ciljne arhitekture-platforme**, koji pokreće napravljeni izvršni fajl, tj. simulira implementaciju koda na platformi, pri čemu je moguće odabrati željeni mikrokontroler i definisati parametre implementacije.
- **Integrisano razvojno okruženje**, koje inicira izvršavanje test slučajeva i pokreće kompajler i linker, a zatim i implementaciju u simulatoru ciljne arhitekture-platforme.

Simulator ciljne arhitekture-platforme obavlja niz koraka i prati odziv programa. Neke operacije simulirani procesor sam obavlja, kao što su npr. aritmetičko-logičke operacije. Simulacija je zapravo virtuelna reprezentacija realnih procesa i dešava se u memoriji računara. Simulator zauzima određeni deo memorije i, u tom delu, izvršava naredbe. Kao i na realnom hardveru, simulator ima mogućnosti da izvršava naredbe korak po korak, da zaustavi izvršavanje programa na unapred zadatom mestu u programu (*breakpoint*), kao i da prati vrednosti promenljivih od interesa tokom izvršavanja.

Zarad što veće pouzdanosti testiranja softvera, u našem sistemu su korišćene obe metode opisane u ovom radu, EP i BVA. Tako, za svaku promenljivu čija promena vrednosti utiče na izvršavanje koda, uzimaju se tri vrednosti, i to: jedna na sredini intervala važenja (particije) što je česta odlika EP metoda, i dve vrednosti na krajevima intervala što je odlika BVA metoda.

Pored testiranja funkcionalnosti i robusnosti namenski dizajniranog softvera, ovim sistemom testirana je i pokrivenost, tzv. *Code coverage* [12]. Pokrivenost koda je pokazatelj da li su određene naredbe suvišne (*Statement coverage*) – da li ih program ikada, za neku vrednost test promenljive izvršava, a takođe i pokazatelj da li su određene

grane uslovnih struktura u kodu suvišne (*Branch coverage*). Ukoliko je bar jedan od ova dva pokazatelja manji od 100%, kod ne može biti potpuno pokriven i postoji “višak” linija koda koji kompajler prevodi i time bespotrebno povećava vreme potrebno za prevođenje, linkovanje i izvršavanje softvera, a povećava se i zauzeće memorije. Takođe, nepokrivenost koda može biti i posledica nedostatka test slučaja koji bi bio „odgovoran“ za određeni deo programskog koda. Mada, neki alati za testiranje imaju mogućnost automatskog dodavanja zaboravljenih test slučajeva, zajedno sa kojima se postiže najveća moguća pokrivenost. U našem slučaju, koristi se već gotov alat za merenje pokrivenosti koda.

Treba još pomenuti i slučaj pozivanja drugih funkcija ili procedura unutar funkcije koje se testira. Ukoliko ne postoji razlog za izvršavanje celokupne funkcije unutar testirane funkcije, već je jedino važna njena povratna vrednost koja biva upisana u neku promenljivu, izvršavanje te funkcije se često menja izvršavanjem tzv. *stub* funkcije. To znači da se celokupna unutrašnja funkcija menja njenom skraćenom verzijom koja je relevantna za dalje izvršavanje okružujuće (spoljašnje) funkcije, a najčešće samo jednom linijom koda ‘return *value*;’ gde je *value* željena povratna vrednost. Ukoliko, pak, postoji razlog da se cela unutrašnja funkcija izvršava (npr. određenim globalnim promenljivim se vrši dodela vrednosti unutar nje), onda se *stub* ne koristi.

Na slici 3 je prikazan izgled interfejsa integrisanog razvojnog okruženja koje je korišćeno za izvršavanje test slučajeva. Kao ilustrativni primer funkcionisanja sistema, izabrana je jednostavna funkcija ‘*is_value_in_range*()’ napisana u programskom jeziku C, koja vraća nabrojani podatak ‘*yes*’ ukoliko se zadata vrednost nalazi unutar zadatog opsega, odnosno ‘*no*’ ukoliko se nalazi van opsega ili ako je manja od vrednosti polja ‘*range_start*’ prethodno definisane strukture ‘*range*’:

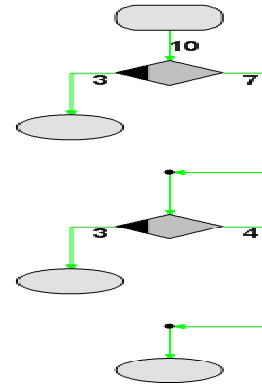
```
result is_value_in_range (struct range r1, value v1)
{
   if (v1 < r1.range_start)
      return no;

   if (v1 > (r1.range_start + r1.range_len))
      return no;

   return yes;
}
```

	1.1	2.1	3.1	4.1	5.1	6.1	7.1	8.1	9.1	10.1
Inputs										
Globals										
Parameter										
struct range r1										
int range_start	10	3	0	-13	50	0	-7	42	-42	0
int range_len	20	5	5	10	0	0	0	-50	-10	-20
int v1	15	20	5	-12	60	-1	-7	20	-43	1
Dynamics										
Outputs										
Globals										
Parameter										
Return										
enum	yes	no	yes	yes	no	no	yes	yes	yes	no
Dynamics										

Sl. 3. Interfejs okruženja za testiranje

Može se primetiti da test slučajevi numerisani sa 8.1 i 9.1 „padaju“, tj. stvarni izlazi ne odgovaraju očekivanim, dok svi ostali test slučajevi „prolaze“, tj. stvarni izlazi odgovaraju očekivanim. U polja „Return-enum“ na slici 3 upisane su očekivane vrednosti izlaza funkcije za zadate vrednosti stimulusa. Razlog „padanja“ test slučajeva 8.1 i 9.1 se može i intuitivno shvatiti na osnovu programskog koda funkcije, a razlog je pogrešna procena ishoda odgovarajućih test slučajeva, tj. stvarni izlazi se ne poklapaju sa upisanim (očekivanim).

Na slici 4 se može videti da je pokrivenost algoritma opisanog funkcijom ‘*is_value_in_range*()’ jednaka 100%, tj. ne postoji niti naredbe, niti grane u koje program ne ulazi ni za jednu vrednost iz opsega važenja ulaznih argumenata, a što se takođe može intuitivno zaključiti iz analize opisa funkcije i test slučaja.



Sl. 4. Analiza pokrivenosti algoritma

V. Zaključak

U radu je predstavljen sistem za testiranje funkcionalnosti i robusnosti programskog koda korišćenjem metoda *Equivalence Partitioning* i *Boundary Value Analysis*. Pored opisa svake od metoda, opisan je celokupan sistem i način njegovog funkcionisanja. Dati su i odgovarajući ilustrativni primeri i najvažniji rezultati testiranja softvera. Ovakav sistem ima značajnu primenu u procesima analize industrijskog, ali i softvera bilo koje druge namene. Za razliku od sistema koji se zasnivaju na slučajnom (*random*) ili totalnom (*brute force*) testiranju, ovaj sistem ima i odgovarajuću tačnost i sposobnost nalaženja “bagova”, kao i jednostavnost implementacije i minimalnu potrošnju resursa – vremena i memorije.

Literatura

[1] I. Burnstein, *Practical Software Testing*, Springer-Verlag, p. 623, 2003.

[2] S. C. Reid, “An empirical analysis of equivalence partitioning, boundary value analysis and random testing”, 4th International Software Metrics Symposium Proc., Albuquerque, USA, 1997.

[3] N. Juristo, S. Vegas, M. Solari, S. Abrahao, I. Ramos, “Comparing the Effectiveness of Equivalence Partitioning, Branch Testing and Code Reading by Stepwise Abstraction Applied by Subjects”, IEEE 5th International Conference on Software Testing, Verification and Validation Proc., Montreal, Canada, 2012.

[4] A. Mathur, *Foundations of Software Testing: Fundamental Algorithms and Techniques*, Pearson India, 2007, p. 96.

[5] W.R. Adrion, M.A. Branstad, J.C. Cherniavsky, “Validation, verification, and testing of computer software”, *ACM Computing Surveys*, vol. 14, no. 2, New York, USA, June 1982, pp. 159-192.

[6] J.B. Goodenough, S.L. Gerhart, “Toward a theory of test data selection”, International Conference on Reliable Software Proc., Los Angeles, USA, 1975, pp. 493-510.

[7] W.E. Howden, “A survey of dynamic analysis methods”, *Software Testing and Validation Techniques*, IEEE Computer Society, Long Beach, USA, 1978, pp. 184-206.

[8] D.J. Richardson, L.A. Clarke, “A partition analysis method to increase program reliability”, 5th International Conference on Software Engineering Proc., San Diego, USA, March 1981, pp. 244-253.

[9] E.J. Weyuker, T.J. Ostrand, “Theories of program testing and the application of revealing subdomains”, *IEEE Transactions on Software Engineering*, vol. SE-6, no. 3, May 1980, pp. 236-246.
[10] E.J. Weyuker, B. Jeng, “Analyzing Partition Testing Strategies”, *IEEE Transactions on Software Engineering*, vol. 17, no. 7, July 1991, pp. 703-711.
[11] R. Hamlet, R. Taylor, “Partition Testing Does Not Inspire Confidence”, *IEEE Transactions on Software Engineering*, vol. 16, no. 12, Dec. 1990, pp. 1402-1411.
[12] K. Burr, W. Young. “Combinatorial test techniques: Table-based automation, test generation and code coverage”, International Conference on Software Testing Analysis & Review Proc., 1998.

ABSTRACT

In this paper the system for testing of functionality and robustness of programming code has been described. Testing is performed by using conventional methods *Equivalence Partitioning* (EP) and *Boundary Value Analysis* (BVA), which are platform, technology, purpose and programming language independent. As a tool for parameter and function extraction and compiler and debugger integration, an environment for testing embedded software, written in C or C++, has been used.

Functionality and robustness software testing by using Equivalence Partitioning and Boundary Value Analysis

Zlatko Veličković, Darko Tasovac, Lazar Saranovac