Verifikacija softvera

Milena Vujošević Janičić

6. novembar 2023.

Matematički fakultet, Univerzitet u Beogradu

Verifikacija softvera

Copyright ©Milena Vujošević Janičić
Ovo delo zaštićeno je licencom Creative Commons CC BY-NC-ND 4.0 (Attribution-NonCommercial-NoDerivatives 4.0 International License). Detalji licence mogu se videti na veb-adresi `http://creativecommons.org/licenses/by-nc-nd/4.0/`. Dozvoljeno je umnožavanje, distribucija i javno saopštavanje dela, pod uslovom da se navedu imena autora. Upotreba dela u komercijalne svrhe nije dozvoljena. Prerada, preoblikovanje i upotreba dela u sklopu nekog drugog nije dozvoljena.

Izdavač
Matematički fakultet, Univerzitet u Beogradu. Studentski trg 16, Beograd.
Za izdavača: *prof. dr XX*, dekan
Jun, 2024.

– *Mami*

Predgovor

XXX

Zahvaljujem se studentima koji su pomogli u izradi ilustracija: Vukan Antić — slike 2.9, **??**, Milica Gnjatović — slike **??**, **??**, **??**, Pavle Cvejović — slike **??**, **??**, **??**, Bojan Bardžić — slike 5.1, 5.2, 5.3, 5.4, 5.5, 5.6.

Milena Vujošević Janičić

Sadržaj

Predgovor **v**

Sadržaj **vii**

1 Uvod **1**
1.1 Razvoj softvera i greške 1
1.1.1 Razvoj softvera 1
1.1.2 Kvalitet softvera 2
1.1.3 (Ne)ispravanost softvera 5
1.2 Tehnike verifikacije softvera 7
1.2.1 Dinamička verifikacija softvera 7
1.2.2 Statička verifikacija softvera 8

Dinamička verifikacija softvera **11**

2 Testiranje **13**
2.1 Testiranje i razvoj softvera 13
2.1.1 Cena grešake u kontekstu vremena otkrivanja 13
2.1.2 Uloga testera u razvoju softvera 15
2.1.3 Faze testiranja softvera 16
2.2 Vrste testiranja 19
2.2.1 Testiranje jedinica koda 19
2.2.2 Komponentno i integraciono testiranje 20
2.2.3 Sistemsko testiranje 21
2.2.4 Nefunkcionalno sistemsko testiranje 22
2.2.5 Regresiono testiranje 23
2.3 Tehnike testiranja 23
2.3.1 Testiranje crne kutije 24
2.3.2 Testiranje bele kutije 30
2.3.3 Metamorfno testiranje 34
2.4 Načini testiranja 36
2.4.1 Automatsko generisanje test primera 36
2.4.2 Automatsko izvršavanje test primera 36
2.4.3 Manuelno testiranje 37

3 Debagovanje **39**
3.1 Debagovanje 39
3.1.1 Prevođenje sa ciljem debagovanja 39
3.1.2 Kako rade debageri? 41

3.1.3 Potrebna podrška . . . 42
3.1.4 Vrste debagovanja . . . 43
3.1.5 Primeri debagera . . . 44
3.1.6 Debagovanje drugih programskih jezika . . . 45
3.1.7 Otvoreni problemi . . . 46
3.1.8 `Print` umesto debagera . . . 46

4 Profajliranje **47**
4.1 Osnovni pojmovi . . . 47
4.1.1 Vrste optimizacija na osnvou rezultata profajliranja . . . 48
4.1.2 Podrška profajliranju . . . 48
4.1.3 Implementacija profajliranja . . . 48
4.2 Profajliranje uzimanjem uzoraka . . . 49
4.3 Instrumentalizacija . . . 49
4.3.1 Osnovne vrste profajliranja . . . 50
4.3.2 Instrumentalizacija u kombinaciji sa uzimanjem uzoraka . . . 52
4.4 Kompajlerski zasnovana dinamička analiza — sanitajzeri . . . 54
4.4.1 Detektovanje grešaka u radu sa memorijom i nitima . . . 54
4.5 Napredna analiza izvršnog programa . . . 55
4.5.1 Platforma Valgrind . . . 55
4.5.2 Kako radi Valgrind? . . . 56
4.5.3 Translacija . . . 57
4.5.4 Memcheck . . . 58
4.5.5 Massif . . . 62
4.5.6 Cachegrind . . . 62
4.5.7 Callgrind . . . 64
4.5.8 Helgrind i DRD . . . 65
4.6 Alat Perf . . . 66

Statička verifikacija softvera **69**

5 Pregledi koda **71**
5.1 Šta se pregleda? . . . 71
5.2 Vrste pregleda koda . . . 73
5.2.1 Formalni pregledi . . . 73
5.2.2 Neformalni pregledi . . . 74
5.2.3 Uticaj pregleda . . . 77
5.2.4 Pravila efikasnog pregledanja . . . 79

6 Simboličko izvršavanje **81**
6.1 Simboličko izvršavanje kroz primer . . . 81

6.2 Istorija, alati, stablo izvršavanja 82
6.2.1 Simboličko stablo izvršavanja 82
6.3 Izazovi simboličkog izvršavanja 84
6.4 Principi dizajna i konkoličko izvršavanje 86
6.4.1 Dinamičko simboličko izvršavanje 87
6.4.2 Selektivno simboličko izvršavanje 89
6.5 Strategije obilaska puteva 90
6.5.1 Naivni pristupi: DFS i BFS 91
6.5.2 Random strategija 91
6.5.3 Izvršavanje vođeno pokrivenošću koda 91
6.5.4 Izvršavanje vođeno najkraćim rastojanjem 93
6.5.5 Kombinovana strategija 93
6.5.6 Izvršavanje unazad 93
6.6 Modelovanje memorije 94
6.6.1 Potpuno simbolička memorija 95
6.6.2 Kompromisi 99
6.6.3 Lenja inicijalizacija 100
6.7 Eksplozija broja stanja i putanja 101
6.7.1 Odsecanje nedostižnih putanja 101
6.7.2 Spajanje stanja 101
6.7.3 Aproksimacije petlji 102
6.8 Rešavači 104
6.9 Binarni kôd 104

7 Semantika programskih jezika **107**
7.1 Neformalna semantika 108
7.2 Osnovne vrste formalnih semantika 110
7.2.1 Rezonovanje o osobinama programa 111
7.3 Operaciona semantika 114
7.3.1 Prirodna semantika 114
7.3.2 Strukturna operaciona semantika 117
7.4 Denotaciona semantika 118
7.5 Aksiomatska semantika 124

8 Proveravanje modela **127**
8.1 Osnovni pojmovi i primene 127
8.1.1 Primene proveravanja modela 130
8.2 Pravljenje modela 131
8.2.1 Tranzicioni sistem 131
8.2.2 Modelovanje hardvera 134
8.2.3 Modelovanje softvera 135
8.3 Formalna specifikacija 138
8.3.1 Tipovi svojstava 138

8.3.2 Temporalne logike 139
8.3.3 Linearna temporalna logika 141
8.3.4 CTL* i CTL 143
8.4 Algoritmi za proveravanje modela 144
8.4.1 Bihijevi automati 145
8.4.2 Kombinatorna eksplozija 147
8.4.3 Apstrakcija predikata 148
8.4.4 Binarni dijagram odlučivanja 150
8.4.5 Primena BDD-ova u proveravanju modela 150
8.4.6 Ograničena provera modela 151

9 Apstraktna interpretacija **157**
9.1 Odnos konkretne i apstraktne semantike 157
9.1.1 Provera važenja sigurnosnih svojstava 158
9.2 Primeri 160
9.3 Izbor apstrakcije 162
9.3.1 Pogrešne apstrakcije 162
9.4 Primene 164

Alphabetical Index **165**

Spisak slika

1.1 Verifikacija i validacija softvera . . . 2
1.2 Atributi softvera su skladu sa kategorizacijom standarda ISO/IEC 25010. . . . 3
1.3 Veze između atributa koji utiču na održavanje softvera: strelice sa punim linijama predstavljaju jak uticaj jednog atributa na drugi, dok isprekidane strelice odgovaraju indirektnom uticaju. . . . 5

2.1 Težina ispravljanja grešake . . . 14
2.3 Osnovni razlog neslaganja . . . 15
2.2 Cena ispravljanja grešake . . . 15
2.4 Tester u supermarketu . . . 16
2.5 Testiranje crne kutije . . . 23
2.6 Testiranje bele kutije . . . 24
2.7 Prikaz prelaza u okviru dijagrama stanja. . . . 28
2.8 Slika uz primer sa ugrađenom kasom. . . . 29
2.9 Hijerarhija pokrivenosti . . . 31

4.1 Profil dobijen profajliranjem blokova (a) i profajliranjem grana (b) . . . 51
4.2 Profajliranje grana . . . 52
4.3 Proveravajući i duplirajući kod . . . 53
4.4 Delimično dupliranje . . . 54
4.5 Primer pokazivača na memorijski blok . . . 60
4.6 Rezime curenja memorije . . . 61
4.7 Izveštaj o curenju memorije . . . 61
4.8 Izveštaj alata *Cachegrind* . . . 64

5.1 Tipičan proces rada za „formalni“ pregled. Artefakti dobijeni tokom pregleda nisu prikazani. To su log defekata, beleške sa sastanaka i log metrika. Neki pregledi takođe imaju završnu anketu na propratnom sastanku. . . . 74
5.2 Osnovne vrste neformalnih pregleda . . . 75
5.3 Tipičan proces rada pregleda preko ramena. . . . 75
5.4 Tipičan proces pregleda preko mejla za kôd koji je ubačen u sistem za verzionisanje. Ove faze u stvarnosti nisu tako jasno razdvojene zato što ne postoji opipljiv predmet pregledanja. . . . 76
5.5 Tipičan proces pregleda preko mejla za kôd koji nije ubačen u sistem za verzionisanje. Ove faze u stvarnosti nisu tako jasno razdvojene zato što ne postoji opipljiv predmet pregledanja. . . . 77
5.6 Pregledanje koda pomoću alata za automatski pregled i od strane programera u timu. . . . 77
5.7 Proces pregleda . . . 78

5.8 Fabrikator 79

6.1 Kod (levo), konkretno izvršavanje (sredina) i simboličko izvršavanje (desno). 81
6.2 Primer stabla izvršavanja: kôd (leov) i uslovi za svaku putanja izvršavanja (desno) 83
6.3 Funkcija foobar 83
6.4 Simboličko izvršavanje funkcije `foobar`. 84
6.5 Primer nedostupne/nedostižne putanje 84
6.6 Eksponencijalni broj putanja: tri grananja daje 8 mogućih putanja 85
6.7 Kako pratiti funkcijske pozive za koje kôd nije dostupan? 88
6.8 Pravljenje novih stanja 95
6.9 Za kod sa leve strane može se utvrditi da je jedna putanja nedostižna (slika desno). 101
6.10 Stablo izvršavanja za kôd koji je prikazan sa leve strane: bez spajanja stanja (sredina) i sa spajanjem stanja korišćenjem *ite* formule (desno) 102

7.1 Deo neformalne semantike za programski jezik Algol 60 108
7.2 Deo neformalne semantike za programski jezik Java 108

8.1 Strukturni prikaz — sistemi uopšte 127
8.2 Strukturni prikaz — softver 128
8.3 Odnos temporalnih logika 140
8.4 LTL i CTL 140
8.5 Intuitivno značenje CTL operatora 144
8.6 CEGAR — prikaz procesa 149

Spisak tabela

2.1 Tabela stanja koja odgovara dijagramu stanja sa slike 2.8. 30

Uvod | 1

Pregled

- Na koji sve način neispravan softver utiče na svet oko nas?
- Šta obuhvata verifikacija softvera?
- Koji su osnovni pristupi i problemi koji se javljaju u okviru verifikacije softvera?
- Koje su specifičnosti automatske statičke verifikacije softvera?
- Kakvi su svetski trendovi u ovoj oblasti?

1.1 Razvoj softvera i greške 1
1.2 Tehnike verifikacije softvera 7

1.1 Razvoj softvera i greške

1.1.1 Razvoj softvera

Softver je sastvani deo svih aspekata naših života.

Razvoj softvera obuhvata metode, principe i procedure potrebne da se procesom izrade softvera dođe do efikasnog i pouzdanog proizvoda.

Razvoj softvera je složen proces izrade softvera koji obuhvata veliki broj različitih aktivnosti vezanih za:

- analizu sistema, specifikaciju zahteva,
- projektovanje i implementaciju softvera,
- obezbeđivanje kvaliteta
- održavanje softvera

Obezbeđivanje kvaliteta (eng. *quailty assurance*) obuhvata

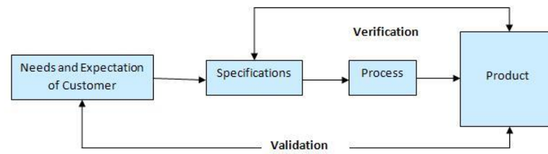
Validaciju — Da li specifikacija zadovoljava korisničke potrebe?

Verifikaciju — Da li softver zadovoljava specifikaciju? (Da li softver sadrži nedostatke? Da li je softver ispravan?)

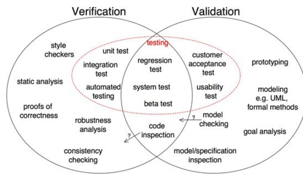
I validacija i verfikacija se najčešće sprovode testiranjem, ali postoje i druge napredne tehnike koje se takođe koriste.

Proces verifikacije i validacije kod studenata sprovodi se pri kraju procesa razvoja softvera korišćenjem malog broja test primera. Obično čini od 2% do 5% ukupnog napora razvoja. Proces verifikacije i validacije u industriji sprovodi se tokom celokupnog razvoja softvera i čini više od 30% od ukupnog napora, pri čemu taj procenat može značajno da varira, u zavisnosti od vrste projekta i od nivoa kvaliteta koji je neophodno postići. Za bezbednosno kritične sisteme ovaj procenat je značajno viši (čak i 70%). Osnovne razlike između studentskih i industrijskih projekata su

- Dužina upotrebe softvera
- Način upotrebe softvera
- Broj korisnika
- Očekivanja
- Cena pada
- ...



Slika 1.1: Verifikacija i validacija softvera



ISO 9126: Evaluation of Software Quality

- **Functionality** - *A set of attributes that bear on the existence of a set of functions and their specified properties. The functions are those that satisfy stated or implied needs.*
  - Suitability
  - Accuracy
  - Interoperability
  - Compliance
  - Security
- **Reliability** - *A set of attributes that bear on the capability of software to maintain its level of performance under stated conditions for a stated period of time.*
  - Maturity
  - Recoverability
  - Fault Tolerance
- **Usability** - *A set of attributes that bear on the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users.*
  - Learnability
  - Understandability
  - Operability
- **Efficiency** - *A set of attributes that bear on the relationship between the level of performance of the software and the amount of resources used, under stated conditions.*
  - Time Behaviour
  - Resource Behaviour
- **Maintainability** - *A set of attributes that bear on the effort needed to make specified modifications.* Stability
  - Analyzability
  - Changeability
  - Testability
- **Portability** - *A set of attributes that bear on the ability of software to be transferred from one environment to another.*
  - Installability
  - Replaceability
  - Adaptability
  - Conformance (similar to compliance, above, but here related specifically to portability, e.g. conformance to a particular database standard)

Standard ISO/IEC 25010 je evoluirao iz standarda ISO 9126.

1.1.2 Kvalitet softvera

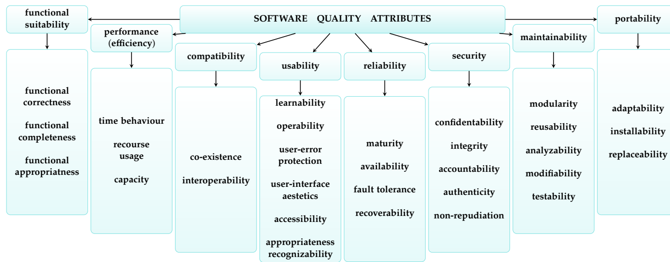
Visok kvalitet softvera je od suštinskog značaja za softver koji treba da bude konkurentan na tržištu. Standardi kvaliteta softvera daju definicije i propise koji omogućavaju sistematski pristup kvalitetu softvera. Pored toga, za dobijanje visokokvalitetnog softvera neophodna je automatizovana podrška i neophodni su sofisticirani alati.

Osnovni procesi koji su vezani za kvalitet softvera uključuju planiranje kvaliteta softvera, obezbeđivanje (eng. *assurance*) kvaliteta softvera i kontrolu (eng. *control*) kvaliteta softvera. Planiranje kvaliteta softvera je potrebno da bi se definisali potrebni nivoi kvaliteta i pristupi razvoju softvera koji su potrebni da bi se željeni nivo kvaliteta postigao. Na primer, različit nivo kvaliteta softvera se očekuje za softver aviona i igricu na mobilnom telefonu. Obezbeđivanje kvaliteta softvera podrazumeva uključivanje aspekata kvaliteta u svakodnevni rad, dok kontrola treba da obezbedi da dobijeni rezultati budu željenog kvaliteta.

Željeni kvalitet softvera je definisan softverskim zahtevima, ali može biti nametnut i različitim međunarodnim standardima. Serija standarda ISO/IEC 25000 sadrži okvir za procenu kvaliteta softvera. Najvažniji standard je ISO/IEC 25010. Ovaj standard definiše osam karakteristika kvaliteta softvera, koje se obično nazivaju atributi kvaliteta softvera. Standard ISO/IEC 25023 opisuje kako se ove karakteristike kvaliteta koriste za merenje ukupnog kvaliteta proizvoda.

Postoje i važni IEEE standardi koji se široko koriste. Na primer, IEEE 730 standard daje smernice za pokretanje, planiranje, kontrolu i izvršavanje procesa obezbeđenja kvaliteta softvera, dok IEEE 1012-2016 standard definiše procese verifikacije i validacije za razvoj sistema, softvera i hardvera.

U zavisnosti od svrhe i ciljeva softvera, svaki atribut kvaliteta softvera može imati različit nivo važnosti. Atributi kvaliteta softvera, definisani standardom ISO 25010, prikazani su na



Slika 1.2: Atributi softvera su skladu sa kategorizacijom standarda ISO/IEC 25010.

slici 1.2. Ovaj standard definiše hijerarhiju sledećih osam atributa kvaliteta (koji takođe uključuju podatribute):

Funkcionalna podobnost odgovara stepenu u kojem softver odgovara funkcionalnim zahtevima. Ovo uključuje funkcionalnu ispravnost (tačni rezultati), funkcionalnu potpunost (dostupnost funkcija očekivanih specifikacijom) i funkcionalnu prikladnost (ispunjavanje njene funkcionalnosti). To je dinamičko svojstvo softvera i može se proceniti kroz procese verifikacije i validacije softvera.

Performanse (efikasnost) odgovara stepenu u kojem softver odgovara na zahteve performansi. Uključuje vremensko ponašanje (vreme odgovora i obrade ili stope protoka), korišćenje resursa (količine i vrste korišćenih resursa) i kapacitet (maksimalna ograničenja). To je dinamičko svojstvo i može se proceniti kroz procese verifikacije softvera. Za to se obično koriste profajleri koji mogu pronaći probleme sa performansama.

Kompatibilnost odgovara stepenu u kojem softver može da radi na različitim platformama ili da deli podatke sa drugim proizvodima, sistemima i komponentama. Uključuje koegzistenciju (sposobnost deljenja okruženja i resursa sa drugim softverskim proizvodima) i interoperabilnost (sposobnost promene i korišćenja informacija sa/iz drugih aplikacija). To je dinamičko svojstvo i može se proceniti kroz procese verifikacije softvera.

Upotrebljivost odgovara stepenu u kojem određeni korisnici mogu koristiti softver u određenim uslovima. Uključuje mogućnost učenja (lakoća učenja kako softver funkcioniše), operabilnost (lakoća rada i kontrole softvera), zaštitu od grešaka korisnika (zaštita od pravljenja grešaka u korišćenju), estetiku korisničkog interfejsa (prijatnost i prihvatljivost korisničkog interfejsa), pristupačnost (upotrebljivost od strane ljudi sa

različitim mogućnostima), prepoznatljivost prikladnosti (sposobnost korisnika da lako prepoznaju da je softver odgovarajući za njihove potrebe). Upotrebljivost se procenjuje aktivnostima validacije softvera.

Pouzdanost odgovara stepenu u kome je softver pouzdan. Uključuje zrelost (stabilnost tokom svakodnevne upotrebe), dostupnost (mogućnost da budete uvek dostupni), toleranciju na greške (sposobnost rada čak i u prisustvu nekih hardverskih ili softverskih kvarova) i povratljivost (sposobnost oporavka podataka u slučaju kvara sistema). To je dinamičko svojstvo i može se proceniti kroz procese verifikacije softvera.

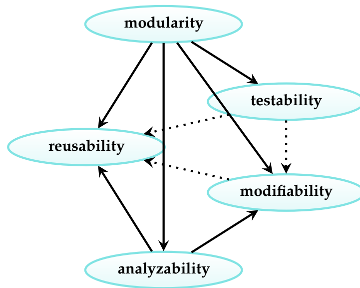
Bezbednost odgovara stepenu u kojem softver štiti informacije i podatke. Uključuje poverljivost (dostupnost podataka samo ovlašćenim korisnicima), integritet (sprečavanje neovlašćenog pristupa i modifikacije podataka), odgovornost (mogućnost praćenja radnji), autentičnost (mogućnost dokazivanja identiteta), neporicanje (mogućnost prikupljanja informacija o određenim radnje i događaje). To je dinamičko svojstvo i može se proceniti kroz procese verifikacije softvera.

Održavanje odgovara sposobnosti softvera da efikasno inkorporira promene. Održivost je statička osobina softvera i uključuje modularnost (stepen u kojem je logičko particionisanje na nezavisne i zamenljive module prisutno unutar softvera), ponovnu upotrebu (stepen u kojem komponente jednog sistema mogu da se koriste u okviru drugih sistema), analitičnost (lakoća analiziranja i razumevanja softvera), promenljivosti (lakoća implementacije željenih promena unutar softvera, bez uvođenje novih grešaka i problema) i mogućnost testiranja (lakoća provere da li promene nisu oštetile postojeći kod). Smatra se jednim od ključnih atributa kvaliteta i može se proceniti kroz pregled koda i alate za statičku analizu.

Prenosivost odgovara stepenu u kome se softver može koristiti u različitim okruženjima. Uključuje prilagodljivost (mogućnost korišćenja sa različitim hardverom, softverom ili okruženjem), instalabilnost (mogućnost instaliranja/deinstaliranja softvera u različitim okruženjima) i zamenljivost (mogućnost zamene drugog softverskog proizvoda za istu svrhu). To je dinamičko svojstvo i može se proceniti kroz procese verifikacije softvera.

Atributi kvaliteta softvera nisu nezavisni, već su međusobno

povezani i isprepleteni. Na primer, razmotrite atribute održavanja. Modularnost, koja se obično postavlja kao jedan od glavnih ciljeva faze projektovanja softvera, direktno utiče na sva ostala četiri atributa, jer je dobra modularnost preduslov za ponovnu upotrebu, mogućnost analize, promenljivost i testiranje. Slično tome, mogućnost analize utiče na ponovnu upotrebu i modifikaciju, dok mogućnost testiranja indirektno utiče na ponovnu upotrebu i promenljivost. Sve ove veze su prikazane na slici 1.3.



Slika 1.3: Veze između atributa koji utiču na održavanje softvera: strelice sa punim linijama predstavljaju jak uticaj jednog atributa na drugi, dok isprekidane strelice odgovaraju indirektnom uticaju.

1.1.3 (Ne)ispravanost softvera

Greške su svuda oko nas, tj one su sastavni deo ljudskih života. Ljudi prave greške u razmišljanju, postupcima i u svojim proizvodima. Greške se pojavljuju svuda gde ljudi sprovode akcije i prave odluke.

Na koji sve način neispravan softver utiče na svet oko nas?

- Neprijatnosti — Mobilni telefoni, Internet pregledači, muzički uređaji...
- Materijalni gubici — Poslovni softver, banke, gubici podataka (virusi)...
- Fatalne posledice — Avioni, automobili, vozovi, aparati u zdravstvu, svemirske letilice, nuklearne elektrane

Alan Džej Perlis (engl. Alan Jay Perlis, Pitsburg, 1. april 1922 — Nju Hejven, 7. februar 1990) je bio američki naučnik koji se bavio računarstvom, poznat po svom pionirskom bavljenju razvojem programskih jezika i kao prvi dobitnik Tjuringove nagrade:

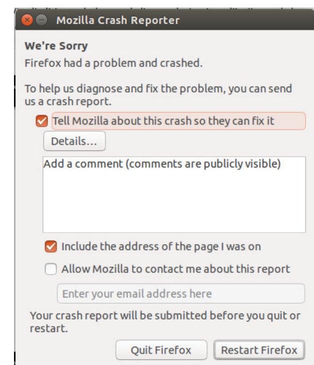
> *"It is easier to change the specification to fit the program than vice versa."*

Primeri

Apple Maps gives us directions to nowhere (2012)

Love virus, 2000 Desetine miliona zaraženih računara, značajan gubitak podataka, šteta od oko 10 milijardi dolara

Knight Capital Group, 2012 Izgubili su $460 miliona dolara za samo 45 minuta usled greške u softveru kojom je pokrenuta pogrešna verzija softvera.



Greška u radu internet pregledača.

Ariane 5, 1996 Ariane 5, 1996 raspala se 37s posle lansiranja usled greške u konverziji (64bit u 16bit), šteta 370 000 000 $

Dhahran raketa, 1991 Raketa je pogodila cilj i ubila 28 vojnika: zbog greške u softveru nije ispaljena protivodbrana

Therac 25, 1986 Više ljudi je umrlo kao posledica predoziranja radijacijom usled neispravnosti softvera uređaja za terapijsku radijaciju.

Greška u sistemu za izgradnju alata.

Fiat Chrysler *Fiat Chrysler recalls 1.25 million trucks over software error* (link) Kompanija Fijat Krajsler (eng. *Fiat Chrysler*) u maju 2017. godine imala je grešku u softveru koja je dovela do tragičnih saobraćajnih nesreća. Greška je prouzrokovala to da bočni vazdušni jastuci kao i pojas u slučaju naglog kočenja ne rade. Zbog toga su opozvali 1.25 miliona kamiona iz upotrebe!

General motors Vest: *GM recalls 4.3 million vehicles over air bag-related defect* (link) Sličan problem je imala i kompanija Dženeral Motors (eng. *General Motors*) koja je grešku otkrila tek nakon saobraćajne nesreće koja je ubila jednog i ranila tri čoveka. Zbog toga su opozvali 4.3 miliona vozila!

Greška $\xrightarrow{proizvodi}$ Nedostatak $\xrightarrow{uzrokuje}$ Pad $\xrightarrow{pravi}$ Incident $\xrightarrow{posledice}$ ...
Therac 25

- Incident — Pacijent je umro
- Pad — Prekoračena je bezbedna doza zračenja
- Nedostatak — U softveru je nedostajala provera opsega frekvencije zračenja koja se primenjuje
- Greška — (1) Programer je zaboravio da stavi proveru opsega frekvencije zračenja, (2) tehničar je uneo vrednost van opsega

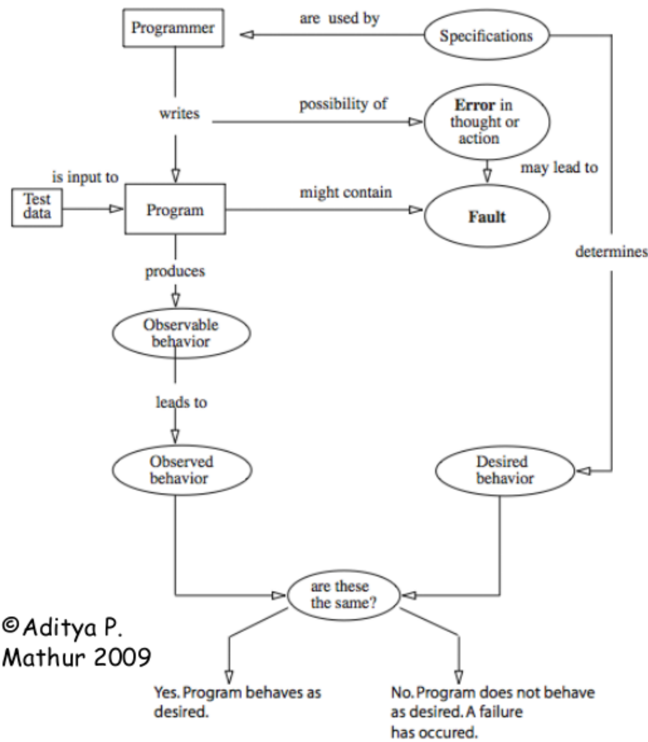
Koliko koštaju greške?

Postoji veliki izbor, vreme tolerancije grešaka je prošlo:

- 50% korisnika obriše mobilnu aplikaciju zbog samo jedne greške (propust u verifikaciji softvera)
- 50% aplikacija se skine i koristi samo jednom (propust u validaciji softvera)

Cena neispravnosti softvera — 2002. godine

> Neispravan softver košta američku ekonomiju 59.5 milijardi dolara godišnje.*



> Ranije otkrivanje grešaka moglo bi da uštedi 22 milijarde dolara godišnje.*
> *_The Economic Impacts of Inadequate Infrastructure for Software Testing_, US National Institute of Standards and Technology (NIST), 2002.

Cena neispravnosti softvera — sada? Neuporedivo je veća — broj linija koda se značajno povećao dok je broj defakata po liniji koda relativno stabilan.

1.2 Tehnike verifikacije softvera

Tehnike verifikacije softvera obuhvataju dinamičku i statičku verifikaciju softvera.

1.2.1 Dinamička verifikacija softvera

Dinamička verifikacija softvera obuhvata tehnike ispitivanja ispravnosti koda u toku njegovog izvršavanja.

Najčešći vid verifikacije softvera je **testiranje**. Testiranje se često koristi kao sinonim za verifikaciju softvera. Testiranje se često koristi i kao sinonim za validaciju i verifikaciju softvera.

Testiranje je tehnika izvršavanja programa sa namerom da se pronađe što više mogućih defekata ili da se stekne dovoljno poverenja u sistem koji se testira. Pravilnim i sistematičnim testiranjem podižemo nivo pouzdanosti i smanjujemo verovatnoću da greške promaknu.

Testiranje se ne radi nasumično, već je važno poznavati metodologiju, procese i principe testiranja.

U okviru testiranja, koriste se i alati za debagovanje i razne vrste profajliranja.

1.2.2 Statička verifikacija softvera

Statička verifikacija softvera predstavlja analizu ispravnosti programa bez njegovog izvršavanja — analizu izvornog koda programa. Postoje različite vrste statičke verifikacije:

- Ručne provere i pregledi koda
- Formalne metode verifikacije softvera — uslovi ispravnosti softvera iskazuju se u terminima matematičkih tvrđenja na striktno definisanom formalnom jeziku izabrane matematičke teorije.

Ručne provere i pregledi koda su veoma važni i svakodnevno se primenjuju u okviru procesa razvoja softvera. Formalne metode se sve više koriste na najrazličitije načine.

Formalne metode verifikacije softvera

Idealno rešenje za veririkaciju softvera bi bio alat koji automatski analizira kôd i daje precizne informacije o njegovoj ispravnosti. Međutim, postoji fundamentalno ograničenje zbog kojeg tako nešto nije moguće napraviti. Naime, **halting problem je neodlučiv.*** Ne postoji opšti automatizovan način za proveravanje da li je neka naredba programa dostižna, pa sami tim ni da li je ispravna, odnosno da li je sam program ispravan.

Posledica teorijskog ograničenja je da nije moguće napraviti program koji bi potpuno automatski u konačnom vremenu,

* Alan Turing, *On Computable Numbers With an Application to the Entscheidungsproblem*, Proceedings of the London Mathematical Society, **1936**.

koristeći konačne resurse mogao da utvrdi ispravnost proizvoljnog programa poptuno precizno. Međutim, ukoliko se odreknemo potpune preciznosti, možemo da napravimo program koji poptuno automatski, u konačnom vremenu, koristeći konačne resurse može da da veoma korisne informacije o ispravnosti programa, iako ne u potpunosti precizne. Preciznost može da bude narušena na dva načina

- Lažna upozorenja
- Propuštene greške

Automatski pristupi obično teže da imaju ili samo lažna upozorenja, ili samo propuštene greške — kombinovanje lažnih upozorenja sa propuštenim greškama čini alat praktično nepoželjnim. Takođe, prilikom izgradnje automatskog alata za ispitivanje ispravnosti često se pravi kompromis između preciznosti i efikasnosti. Efikasni alati nisu precizni, dok precizni alati nisu efikasni.

U automatsku statičku analizu se ubrajaju:

- Simboličko izvršavanje
- Proveravanje modela
- Apstraktna interpretacija
- Analiza vođena kontra-primerima

Pored alata za automatsko otkirvanje grešaka u programiram, formalne metode verifikacije softvera obuhvataju i tehnike razvoja ispravnog softvera. Razvoj direktno iz specifikacije i formalno dokazivanje ispravnosti softvera koji se razvija predstavljaju najviši nivo sigurnosti u ispravnost softvera. Ovo je ujedno i najskuplji razvoj softvera i zahteva visoko stručne ljude i upotrebu posebnih alata, kao što su npr *Coq* i *Isabelle*. Iako ovakav razvoj softvera vodi najpouzdanijem softveru, ovakav razvoj je najskuplji i najsporiji pa se u industriji koristi samo za razvoj kritičnih delova koda.

Rezime

- Uticaj neispravnog softvera: neprijatnosti, materijalni troškovi, materijalno nemerljive posledice...
- Verifikacija softvera: skup metoda, alata i procesa za utvrđivanje ispravnosti softvera
- Osnovni pristup: testiranje, ali testiranje ne može da garantuje ispravnost softvera

- Automatske statička verifikacija: neophodni su kompromisi između preciznosti i efikasnosti
- Formalne metode: za razvoj kritičnih delova aplikacija
- Trendovi: razvoj algoritama i integracija alata automatske statičke verifikacije softvera, primena ovih tehnika u sintezi koda i bioinformatici, poboljšavanje i ubrzavanje metodama mašinskog učenja

Dinamička verifikacija softvera

Testiranje 2

2.1 Testiranje i razvoj softvera 13
2.2 Vrste testiranja . . . 19
2.3 Tehnike testiranja . 23
2.4 Načini testiranja . . 36

Dinamička verifikacija softvera obuhvata tehnike ispitivanja ispravnosti softvera u toku njegovog izvršavanja. Najčešći vid dinamičke verifikacije softvera, a i verifikacije softvera uopšte, je **testiranje**. Pravilnim i sistematičnim testiranjem podižemo nivo pouzdanosti i smanjujemo verovatnoću da greške promaknu. Zbog toga je važno poznavati metodologiju, procese i principe testiranja.

Pregled

- Testiranje i razvoj softvera — koja je uloga i gde je mesto testiranja u procesu razvoja softvera?
- Vrste testiranja — šta tačno proveravamo?
- Tehnike testiranja — gde pronaći dobre test primere?
- Načini testiranja — manuelno ili automatsko testiranje?

2.1 Testiranje i razvoj softvera

Softver se implementira prema zahtevima korisnika. Svako ponašanje softvera koje se ne slaže sa zahtevima korisnika uzrokovano je nekakvom ljudskom greškom u razvoju softvera i predstavlja defekt koji čini softver manje kvalitetnim. U zavisnosti od namene softvera, smanjen kvalitet softvera može da ima najrazličitije posledice. Testiranje predstavlja važan deo životnog ciklusa razvoja softvera.

2.1.1 Cena grešake u kontekstu vremena otkrivanja

Testiranje se često koristi i kao sinonim za verifikaciju softvera. Testiranje se često koristi i kao sinonim za validaciju i verifikaciju softvera. Testiranje se često koristi i kao sinonim za brigu o kvalitetu softvera. Međutim, testiranje je ipak samo važan deo verifikacije softvera, važan deo validacije softvera, a verifikacija i validacija su važan deo brige o kvalitetu softvera.

U okviru razvoja softvera, cilj je da se maksimizuje profit pravljenjem prozvoda visokog kvaliteta ali u vremenskim i budžetskim granicama. Najzastupljenije i trenutno najpopularnije metodologije razvoja softvera promovišu paralelnu implementaciju i pisanje testova za svaku od celina koja se razvija u okviru softverskog sistema.

Sa porastom složenosti projekta, raste i značaj testiranja i provera celokupnog softverskog sistema kako bi se izbegli

ishodi koji mogu da unište ceo projekat. Poželjno je sve greške detektovati što ranije u fazi razvoja softvera jer je ispravljanje grešaka jeftinije i brže u ranijim fazama razvoja softvera.

Edsger Wybe Dijkstra (dobitnik Tjuringove nagrade 1972. godine)

"Program testing can show the presence of bugs, never their absence"

Testiranje može da potvrdi prisustvo grešaka u softveru — testiranje ne može da dokaže ispravnost softvera.

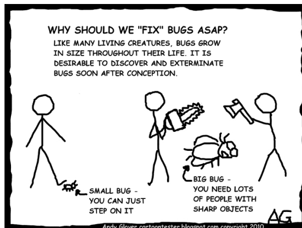
Faza analize zahteva. Ukoliko se greška otkrije u fazi analize zahteva, cena greške obuhvata vreme potrebno da se utvrde i zapišu novi zahtevi.

Faza dizajna softvera. Ukoliko se greška otkrije u fazi dizajna softvera, cena greške obuhvata vreme potrebno da se ispravi dizajn.

Faza implementacije softvera. Ukoliko se greška otkrije u fazi kodiranja, cena greške obuhvata dodatno vreme programera. Vreme može da varira u zavisnosti od kompleksnosti greške, ali je značajno manje nego kada se ispravlja greška koju pronađe neko drugi. Kada programer pronađe sam svoju grešku, on obično razume problem i zna kako da ga reši.

Faza integracije komponenti softvera. Ukoliko se greška otkrije u fazi integracije različitih delova softverskog sistema, onda cena greške obično obuhvata dodatni rad programera ali i drugih učesnika u razvoju softvera. U ovoj fazi vreme za ispravljanje greške je obično dva puta duže jer kada se problem desi na višem nivou potrebno je da se najpre pronađe koji tačno deo koda ili konfiguracija su bili pogrešni.

Faza sistemskog testiranja. Ukoliko se greška otkrije u fazi testiranja celokupnog sistema, onda cena greške dodatno obuhvata i rad program menadžera i QA tima. U ovoj fazi greška zahteva da QA tester bude u stanju da reprodukuje i dokumentuje sve korake koji su potrebni da se opiše greška, QA tester treba da prijavi grešku, da joj da prioritet, da se sastane sa programerma i da to prodiskutuje. Kada programeri isprave grešku, kôd mora ponovo da se integriše u testno okruženje, druge vrste testiranja treba da se ponovo odrade, i za tu grešku mora da se utvrdi da je stvarno popravljena. Defekat takođe mora da se isprati u okviru sistema za praćenje defekata.



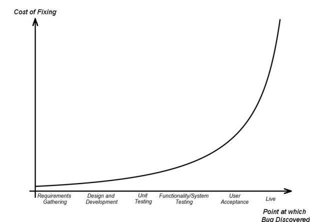
Slika 2.1: Težina ispravljanja greške u kontekstu vremena otkrivanja.

Faza testiranja prihvatljivosti softvera. Ukoliko u procesu razvoja softvera postoji korak testiranja prihvatljivosti softvera, cena greške dodatno obuhvata i rad kupca odnosno korisnika koji vrši testiranje prihvatljivosti. Ovo zahteva komunikaciju između testera za proveru prihvatljivosti sa testerima sistemskog testiranja. Te-

ster sistemskog testiranja će pokušati da reprodukuje grešku i da utvrdi da li je to greška ili sistem radi u skladu sa dizajnom. Ako reprodukovanje greške nije moguće, onda se nastavlja komunikacija sa testerom provere prihvatljivosti. Ako reprodukovanje greške jeste moguće, onda je potrebno da se dokumentuju koraci reprodukovanja greške i da se prođu sve faze kao i kod pronalaženja greške u prethodnoj fazi. Kada se greška ispravi, kôd mora ponovo da se uvede u okruženje za testiranje prihvatljivosti tako da korisnik može da nastavi testiranje.

Faza upotrebe. Ukoliko se greška otkrije kada je program već u upotrebi, cena greške obuhvata i dodatni rad šireg kruga korisnika koji treba da prijave grešku. Prijava greške prati sličan postupak kao i u prethodnom slučaju, ali su posledice značajno veće, a ispravljen kôd treba da se isporuči svima. Takođe, u ovom slučaju, u zavisnosti od greške i vrste softverskog proizvoda, gubi se i poverenje u softver i firmu koja ga razvija, što je veoma značajan gubitak koji se ne može uvek precizno proceniti.

Eksponencijalni rast cene grešake u kontekstu vremena otkrivanja greške prikazuje Slika 2.2.



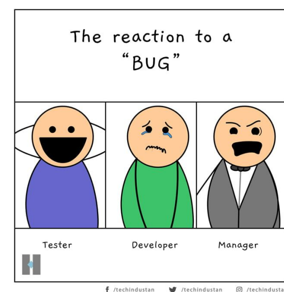
Slika 2.2: Cena grešake u kontekstu vremena otkrivanja.

2.1.2 Uloga testera u razvoju softvera

Projekti se vode na najrazličitije načine. Briga o kvalitetu softvera obuhvata razna pravila i procedure, koji su različiti u zavisnosti od vrste projekata koje firma radi, ali i od zrelosti i veličine firme. U zavisnosti od firme i projekta, QA tim može i ne mora da postoji. Ukoliko postoji zaseban QA tim, on može da bude interni (u okviru firme koja razvija softver) ili eksterni (tim unajmljen u okviru specijalizovane firme koja se bavi testiranjem softvera).

Moguće su različite raspodele obaveza i timova:

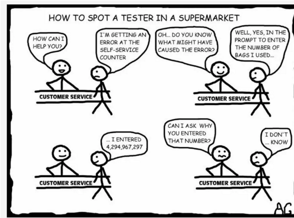
- programeri su istovremeno i testeri,
- programeri i testeri rade zajedno,
- programeri i testeri su potpuno razdvojeni.



Slika 2.3: Osnovni razlog neslaganja testera i programera.

Iako je kvalitetan softver krajnji cilj i programera i testera, često postoji veliki broj problema na relaciji programer – tester. Najčešći problemi su

- loša komunikacija,
- međusobno nerazumevanje,
- netrpeljivost.



Slika 2.4: Tester u supermarketu.

Ovi problemi su najčešće posledica psihološke reakcije na pronalaženje greške u softveru. Pronalaženje greške u softveru testeru označava da je uspešno obavio svoj zadatak, dok programeru to znači da nije uspešno obavio svoj posao i da je potrebno ponovo da radi na nekom delu koda za koji je verovao da je završen.

Primeri neformalne edukacije

- Meetup-ovi — QA Serbia `http://www.qaserbia.rs/`

- Konferencije — Belgrade Test Conference `https://bg-testconference.rs/`

Da bi se osoba bavila testiranjem potrebno je da poseduje osnovno razumevanje programiranja i procesa razvoja softvera. Posebno, mora da detaljno poznaje procedure i proces testiranja kao i alate koji se koriste u testiranju. Za efikasnu implementaciju i automatizaciju testiranja neophodno je i poznavanje skript jezika i skript programiranje. Dobri testeri su kreativni i imaju potrebu za stalnim usavršavanjem, a vremenom upoznaju i česte greške i propuste kao i nesvakidašnje slučajeve upotrebe, što značajno olakšava i ubrzava proces testiranja.

Za poslove testiranja još uvek je dominantna neformalna edukacija kroz kurseve, konferencije i sastanke profesionalnih udruženja.

2.1.3 Faze testiranja softvera

Za početak procesa testiranja, postojanje koda nije neophodno. Dovoljno je imati jasno definisane zahteve korisnika jer priprema za testiranje počinje analizom tih zahteva. Dakle, da bi se započeli procesi vezani za testiranje, potrebno je da postoji specifikacija zahteva sistema (eng. *system requirements specification*) kao i specifikacija zahteva softvera (eng. *software requirements specification*).

Testiranje softvera se u opštem slučaju sastoji od četiri faze pri čemu svaka faza obuhvata veliki broj aktivnosti.

1. Planiranje
2. Analiza, dizajn i implementacija testova
3. Izvršavanje
4. Evaluacija testova

Planiranje

Planiranje (eng. *test planning*) predstavlja pripremu za ceo proces testiranja i uključuje definisanje zadataka koje je potrebno sprovesti kao i način njihovog izvršavanja.

Planiranje definiše:

- potrebne vrste testova,
- tehnike testiranja koje će se koristiti,
- načine testiranja,
- opseg testiranja,
- kriterijum završetka,
- potrebne resurse,
- način komunikacije između članova tima.

Plan testiranja zavisi od primenjene metodologije razvoja softvera i prilagođava se svakom konkretnom projektu.

Rezultat planiranja je skup dokumenata koji sadrže opšti pogled na sistem koji će se testirati, aktivnosti koje će biti izvršene kao i alate koji će biti korišćeni.

Analiza, dizajn i implementacija testova

Analiza, dizajn i implementacija testova (eng. *test analysis, design and implementation*) obuhvata pravljenje detaljne specifikacije načina na koje će se aktivnosti predviđene planom izvršiti. Ispituje se mogućnost testiranja određenih delova koda, prikupljaju potrebni podaci i preciziraju zahtevi korisnika. Kreiraju se i precizna uputstva kako će se vršiti testiranje sistema, i definišu se test slučajevi (engl. *test cases*). Test slučaj je dokument koji definiše skup konkretnih ulaza u sistem i njima odgovarajuće očekivane izlaze.

Rezultat ovih faza je skup test slučajeva i test procedura koja će biti korišćene u fazi izvršavanja testova. Dokument koji prati sve dobijene rezultate naziva se strategija testiranja.

Strategija testiranja je vodič koji se prati za postizanje ciljeva testiranja i izvršavanja testova koji se pominju u planu testiranja. U okviru strategije testiranja, definišu se ciljevi, okruženja, pristupi, automatizacija i tehnike testiranja, nepredviđene situacije i analiza rizika. Definisanje strategije testiranja je najvažniji dokument vezan za testiranje.

Odnos plana i strategije: Ako je plan testiranja destinacija, onda je strategija testiranja uputstvo, mapa, kako da se na tu destinaciju stigne.

Izvršavanje testova

Izvršavanje testova (eng. *test execution*) je proces konkretne primene test slučajeva i test procedura formiranih na osnovu plana, analize, dizajna i implementacije. Samo izvršavanje može biti ručno i automatsko. Ova faza podrazumeva i određivanje prioriteta izvršavanja testova, pripremu testova za automatizovano testiranje (ukoliko je ono deo procesa testiranja) i organizaciju testova za što efikasnije izvršavanje.

Izvršavanje testova se vrši radi provere funkcionalnosti sistema. Ono obuhvata i dodatnu aktivnost praćenja statusa problema. Ova aktivnost podrazumeva eliminaciju prijavljenih problema kao i potvrđivanje da je problem rešen. Neophodno je ponovno izvršavanje testova posle svake popravke grešaka. Ova faza obuhvata aktivnu komunikaciju na relaciji tester — programer.

Evaluacija testova

Evaluacija testova (eng. *test evaluation*) obuhvata procenu kriterijuma završetka testiranja i izveštavanje. Svaka izmena u kodu, čak i koja podrazumeva popravljanje grešaka, može da dovede do novih grešaka. Iz tog razloga se, za različite oblasti testiranja, definiše kriterijum završetka testiranja u odnosu na rezultate izvršavanja test skritpova, procenta nerešenih bagova ili preostalog vremena za testiranje. Proces evaluacija uključuje i pregled rezultata dobijenih analizom izlaza test slučajeva.

Sažeti izveštaj testiranja (eng. *test summary report*) je opis rezultata testiranja. Na osnovu ovog izveštaje se utvrđuje da li je implementirani sistem spreman za korišćenje u skladu sa korisničkim zahtevima.

Izlazni kriterijumi (eng. *exit criteria*) određuju da li je testiranje kompletirano i da li je aplikacija spremna za korišćenje u skladu sa korisničkim zahtevima. Uključuju sažeti izveštaj testiranja, izračunavanje raznih metrika i izveštaj o defektima (eng. *defect analysis report*).

Testiranje se zatvara kada je softver isporučen korisniku, a da pritom ne postoji obaveza održavanja softvera (što je zapravo jako retka pojava). Testiranje se može zaustaviti i u nekim drugim situacijama, na primer, kada je projekat otkazan ili

je neki cilj postignut. Tokom ove faze, test skriptovi i dokumentacija se arhiviraju, dok se primenjeni proces testiranja analizira i diskutuje o tome šta je u procesu bilo dobro, i šta bi trebalo zadržati u narednim projektima. Takođe, analizira se i šta nije bilo dobro kako bi se to izbeglo u budućnosti.

2.2 Vrste testiranja

Postoje različite podele testiranja softvera. Osnovna podela obuhvata

Testiranje funkcionalnih karakteristika — obuhvata proveru ispravnosti aplikacije u odnosu na postavljene funkcionalne zahteve.

Testiranje nefunkionalnih karakteristika — obuhvata proveru ispunjenosti neophodnih tehničkih kvaliteta aplikacije.

Druga bitna podela je podela po **nivoima testiranja**. Mogu se testirati

- pojedinačni moduli,
- grupe modula (vezanih namenom, upotrebom, ponašanjem ili strukturom) ili
- ceo sistem.

U skladu sa pomenutom podelom, prema nivou testiranja, razlikujemo testove jedinice koda, komponentne, integracione i sistemske testove. Na svakom nivou mogu se testirati funkcionalne i nefunkcionalne karakteristike softvera.

2.2.1 Testiranje jedinica koda

Testiranjem jedinica koda (eng. *Unit testing*) proverava se funkcionisanje delova sistema koji se nezavisno mogu testirati. U zavisnosti od konteksta i programske paradigme, to mogu biti podprogrami, klase, manje ili veće celine formirane od tesno povezanih jedinica. Ovom vrstom testiranja prolazi se svaki i najmanji deo sistema. Jedinični testovi definisani su standardom *IEEE Standard for Software Unit Testing*. Testiranje jedinica koda ima dobru podršku u alatima za automatsko izvršavanje i proveravanje ovih testova (sastavni deo razvojnog okruženja).

Cilj jediničnih testova je utvrđivanje da li izolovani delovi koda imaju predviđenu funkcionalnost. Ukoliko kôd komunicira sa mrežom, bazom podataka ili fajl sistemom, to se u okviru testova apstrahuje u nekakve fiksirane vrednosti. Ukoliko kôd komunicira sa drugim klasama, modulima ili komponentama sistema, i to se apstrahuje. Dozvoljena je samo direktna komunikacija sa memorijom. Ove testove najčešće piše programer. Ukoliko postoje greške unutar jedinice koda, one bi trebalo da budu otkrivene u ovoj fazi.

2.2.2 Komponentno i integraciono testiranje

Komponentno testiranje (eng. *component testing*) proverava ispravnost komponente sastavljene od više jedinica koda. Jedinice koda koje su proverene da ispravno rade u izolaciji, sada se testiraju na nivou komponente i proverava se komunikacija između jedinica koda. Neformalno, komponenta je skup povezanih jedinica koda koje imaju zajednički interfejs prema ostalim komponentama. Komponente se proveravaju odmah po njihovom kreiranju pri čemu se testiranje može vršiti izolovano od ostatka sistema, u zavisnosti od izabranog modela razvoja.

Primer: Pretpostavimo da imamo bravu kao na slici ispod. Ukoliko takva brava treba da se integriše sa kliznim vratima (vratima koja se pomeraju levo-desno, a ne napred-nazad), bez obzira što su i brava i vrata ispravni, njihova integracija neće davati željene rezultate.

Komponentno testiranje je veoma slično sa testiranjem jedinica koda, samo što su komponente malo veće. Sama jedinica koda je u prethodnoj fazi testiranja izolovana u potpunosti od spoljašnjeg sistema, dok je u okviru komponentnog testiranja sada napuštena izolacija na nivou same komponente, ali i dalje ostaje izolacija u okviru povezanosti sa drugim komponentama i spoljašnjim sistemom. S obzirom da se u komponentnom testiranju integrišu osnovne jedinice koda, ovo je vrsta integracionog testiranja, tako da se često i ne izdvaja kao posebna vrsta testiranja. Pošto se odnosni na direktno spajanje jedinica koda, može se shvatiti i kao integraciono testiranje na najnižem nivou. Komponentno testiranje može raditi programer, a može i tester, u zavisnosti od vrste projekta.

Primer: Pretpostavimo da imamo dve komponente koje rade sa realnim brojevima. Usled nepreciznosti u specifikaciji, u prvoj komponenti su korišćeni realni brojevi jednostruke tačnosti, a u drugoj realni brojevi dvostruke tačnosti. U okviru integracionog testiranja, trebalo bi da se otkrije ova greška i da se usklade tipovi.

Integraciono testiranje (eng. *integration testing*) proverava saradnju između komponenti koji predstavljaju jednu celinu sistema. Ispituje se da li su veze između komponenti dobro definisane i realizovane, tj. da li komponente komuniciraju na način opisan u specifikaciji projekta. Integracionim testovima proverava se da različite komponente sistema rade ispravno zajedno. Tokom integracionog testiranja mogu se naći

propusti u komunikaciji između komponenti. Integraciono testiranje obično rade testeri.

2.2.3 Sistemsko testiranje

Sistemsko testiranje (eng. *system testing*) obuhvata proveravanje sistema kao celine. Ispituje se da li je ponašanje sistema u skladu sa specifikacijom zadatom od strane klijenta. Ovde se zahteva i potpun pristup bazi i hardverskim delovima sistema. Sistemsko testiranje može da uključuje i **funkcionalne** i **nefunkcionalne** aspekte sistema. U sistemsko testiranje nekada se ubrajaju **istraživačko testiranje** i **testiranje prihvatljivosti**, a nekada se ove dve vrste testiranja izdvajaju nezavisno.

Istraživačko testiranje

Istraživačko testiranje (eng. *exploratory testing*) je testiranje tokom kojeg testeri pronalaze i proveravaju druge (neočekivane) pravce korišćenja softverskog sistema. Ovo podrazumeva istaknutu kreativnost testera. Ova vrsta testiranja obuhvata aktivnosti prepoznavanja, kreiranja i izvršavanja novih test slučajeva (onih koji nisu bili predviđeni test planom). Istraživačko testiranje uglavnom ima smisla kada je aplikacija u svom finalnom obliku, kada tester može videti i druge alternativne pravce korišćenja sistema koji ranije nisu mogli biti predmet testiranja. Ukoliko se ova faza testiranja preskoči, postoji opasnost da neke funkcionalnosti sistema ne budu pokrivene testovima.

Testovi prihvatljivosti

Testovi prihvatljivosti (eng. *acceptance testing*) treba da omoguće klijentima i korisnicima da se sami uvere da je napravljeni softver u skladu sa njihovim potrebama i očekivanjima. Ovu vrstu testiranja izvode i procenjuju korisnici, a razvojni tim im pruža pomoć oko tehničkih pitanja, ukoliko za tim ima potrebe. Testiranje prihvatljivosti obično spada u tehnike validacije softvera.

Klijent može da proceni sistem na tri načina: **referentnim** testiranjem, **pilot** testiranjem i **paralelnim** testiranjem.

Referentno testiranje Kod referentnog testiranja, klijent generiše test slučajeve koji predstavljaju uobičajne uslove u kojima sistem treba da radi. Ove testove izvode korisnici kako bi procenili da li je softver implementiran u skladu sa očekivanjima.

Pilot testiranje Pilot testiranje podrazumeva instalaciju sistema na privremenoj lokaciji i njegovu upotrebu. U ovom slučaju, testiranje se vrši simulacijom svakodnevnog rada na sistemu.

Paralelno testiranje Paralelno testiranje se koristi tokom razvoja, kada jedna verzija softvera zamenjuje drugu ili kada novi sistem treba da zameni stari. Ideja je paralelno funkcionisanje oba sistema (starog i novog) čime se korisnici postepeno privikavaju i prelaze na korišćenje novog sistema.

2.2.4 Nefunkcionalno sistemsko testiranje

Nefunkcionalni zahtevi sistema obuhvataju performanse sistema, efikasnost, otpornost na otkaze, uklapanje u okruženje u kojem će se sistem koristiti.

Tokom testiranja performansi, izvršavaju se testovi **konfiguracije**, **kapaciteta** i **kompatibilnosti**.

Testovima konfiguracije ispituje se ponašanje sistema u različitim hardverskim i softverskim okruženjima. Različite konfiguracije namenjene su različitim korisnicima sistema. Ovim testovima proveravaju se sve konfiguracije sistema.

Testovima kapaciteta proverava se ponašanje sistema pri obradama velikih količina podataka. Proverava se i ponašanje sistema u slučaju kada skupovi podataka postignu svoje maksimalne kapacitete.

Testovima kompatibilnosti proverava se način ostvarivanja komunikacije sistema sa drugim spoljnim sistemima.

Važna krakteristika sistema je i njegova bezbednost. **Testovima bezbednosti** proverava se da li su određene funkcionalnosti dostupne isključivo onim korisnicima kojima su namenjene. Proveravaju se i dostupnost, integritet i poverljivost svih skupova podataka.

Instalaciono testiranje izvodi se instaliranjem softvera na klijentskoj mašini. Prilikom instaliranja, sistem se konfiguriše u skladu sa okruženjem. Ukoliko je potrebno, sistem

se povezuje sa spoljnim uređajima i sa njima uspostavlja komunikaciju. Instalacioni testovi se izvršavaju u saradnji sa korisnicima. Ispituje se da li uslovi na klijentskoj mašini i okruženju negativno utiču na neke funkcionalne ili nefunkcionalne osobine sistema. Kada rezultati testiranja zadovoljavaju potrebe klijenta, testiranje se prekida i sistem se formalno isporučuje.

2.2.5 Regresiono testiranje

Regresiono testiranje se radi nakon izmena u razvoju sistema, da bi se utvrdilo da nisu uvedene greške i da nije došlo do lošeg rada nekih funkcija koje nisu bile obuhvaćene izmenama. Regresioni testovi se sprovode i za testiranje funkcionalnih osobina softvera i za testiranje performaansi programa. Regresioni testovi treba da garantuju da su performanse novog sistema barem jednako dobre kao performanse starog sistema.

2.3 Tehnike testiranja

U kontekstu definisanja tehnika testiranja od suštinske važnosti je metod određivanja reprezentativnog skupa podataka nad kojima će se vršiti testiranje. **Reprezentativni skup testova** treba da ima naredne karakterisitke:

Osnovno pitanje: Kako i gde pronaći dobre test primere?

- Visok potencijal otkrivanja grešaka.
- Relativno mala veličina.
- Relativno velika brzina izvršavanja.
- Visok stepen poverenja u pouzdanost softvera.

Pokrivenost koda (engl. *coverage*) Značenje pojma pokrivenosti zavisi od konteksta u kojem se javlja. Uopšteno, pokrivenost je broj nekih elemenata programa (engl. *items*) ispitanih testovima u odnosu na ukupan broj tih elemenata.

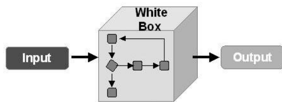
Dobri test primeri se mogu naći na osnovu specifikacije programa i na osnovu koda programa.

Testiranje crne kutije (engl. *black box testing*) — generisanje test primera bez razmatranja interne strukture koda već isključivo na osnovu specifikacije. Ovakav način testiranja se fokusira na ponašanje sistema, posmatrano iz korisničkog ugla. Drugi nazivi su i funkcionalno

Slika 2.5: Testiranje crne kutije: fokus na ulazu i izlazu

testiranje (engl. *functional testing*), testiranje ponašanja (engl. *behavioural testing*), testiranje vođeno podacima (engl. *data driven testing*). Prednost ovog pristupa je mogućnost potpunog razdvajanja programera i testera i zato ovo testiranje obično obavljaju testeri. Zadatak testera je da sistemu pruži ulaze, a zatim da proveri izlaze u odnosu na datu specifikaciju.

Testiranje bele kutije (engl. *white box testing*) — generisanje test primera na osnovu interne strukture koda, npr jedinični testovi. Drugi nazivi su i strukturno testiranje (engl. *structural testing*), testiranje vođeno logikom (engl. *logic driven testing*). Ovo testiranje obično obavljaju programeri.



Slika 2.6: Testiranje bele kutije: fokus na strukturi koda.

Testiranje sive kutije (engl. *gray box testing*) — mešovita strategija. Predstavlja sredinu između tehnika crne i bele kutije. Kod ovih tehnika postoji uvid u unutrašnju strukturu sistema, ali ne u toj meri kao kod tehnika bele kutije. Koristiti se kod komponentnog i integracionog testiranja. Ovo je tehnika koju koriste i programeri i testeri.

Odnos: Testiranje crne kutije je testiranje iz ugla korisnika dok je testiranje bele kutije testiranje iz ugla programera.

2.3.1 Testiranje crne kutije

Testiranje crne kutije teorijski se može uraditi isprobavanjem svih mogućih ulaza (engl. *exhaustive input testing*). Međutim, već za trivijalne programe nije moguće koristiti ovu tehniku.

Primer 2.3.1 Kvadratna jednačina

$$a \cdot x^2 + b \cdot x + c = 0$$

ima rešenje

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4 \cdot a \cdot c}}{2 \cdot a}$$

Ako je svaka promenljiva tipa `int32`, broj različitih test primera za potpuno testiranje je

$$2^{32} \cdot 2^{32} \cdot 2^{32} = 2^{96}$$

Dodatno, pored isprobavanja svih mogućih ispravnih vrednosti ulaza, potrebno je razmotriti i moguće neispravne ulaze kojih takođe može biti mnogo.

Primer 2.3.2 Neispravan ulaz za očekivanu starost osobe može da bude negativan broj ili unos neke proizvoljne reči umesto broja. Očekivano ponašanje softvera u takvim situacijama može da bude signaliziranje greške korisniku sa mogućnošću unosa nove vrednosti ili prekid rada softvera uz odgovarajuću poruku.

Cilj tehnika testiranja crne kutije je da pronađu prihvatljiv broj test slučajeva (tj. kombinacija ulaza) koji odgovara reprezentativnom skupu testova. Test slučajevi mogu se podeliti u dve osnovne kategorije: test slučajevi koji odgovaraju validnim ulazima i test slučajevi koji odgovaraju nevalidnim ulazima.

Primer 2.3.3 (Nastavak primera 2.3.1.) Test slučaj koji odgovara ispravnom ulazu za kvadratnu jednačinu može da sadrži ulazne vrednosti {1, -3, 2} i izlazne vrednosti {1, 2}. Test slučaj koji odgovara neispravnom ulazu može da sadržu ulazne vrednosti {"I", -3, 2} i izlaznu vrednost {"Neispravan unos prvog koeficijenta"}

Pronalaženje reprezentativnog skupa testova metodama crne kutije se ostvaruje postavljanjem odgovarajućih pretpostavki o softveru koji treba da se testira. Najpoznatiji tehnike obuhvataju tehniku klasa ekvivalencije, tehniku graničnih vrednosti, tabele odlučivanja, dijagrame stanja i pogađenje grešaka.

Klase ekvivalencije

Testiranje pomoću klasa ekvivalencije (engl. *equivalence class testing*) je tehnika koja se koristi da smanji broj test slučajeva na prihvatljiv nivo, pritom održavajući razumnu pokrivenost testovima. Ovde se pokrivenost odnosi na procenat svih mogućih ulaza koji će biti ispitan testovima. Ovu jednostavnu tehniku koriste intuitivno skoro svi testeri, iako oni možda nisu svesni da je to formalna metoda oblikovanja testova.

Klasa ekvivalencije predstavlja skup podataka koji se tretiraju jednako od strane modula ili koji treba da proizvedu isti rezultat. Iz tačke gledišta testiranja, sve vrednosti podataka u okviru jedne klase su *ekvivalentne* svim ostalim vrednostima u okviru te klase. Konkretno, očekuje se da:

- Ako jedan test slučaj u jednoj klasi ekvivalencije detektuje grešku, *svi* ostali test slučajevi u okviru iste klase ekvivalencije će verovatno detektovati istu grešku.
- Ako jedan test slučaj u jednoj klasi ekvivalencije ne detektuje grešku, *nijedan* drugi test slučaj u okviru iste klase ekvivalencije verovatno neće detektovati grešku.

Osnovni koraci testiranja pomoću klasa ekvivalencije su

1. Identifikovati validne klase ekvivalencije.
2. Napraviti test slučaj za svaku validnu klasu ekvivalencije.
3. Identifikovati nevalidne klase ekvivalencije.
4. Napraviti test slučaj za svaku nevalidnu klasu ekvivalencije.

Napomena: Često je lakše identifikovati ispravne klase ekvivalencije od neispravnih. Na primer, pored neispravnih unosa -5 i 105, potrebno je testirati i neispravne unose kao što je unos izraza koji vodi neispravnom broju (npr. 66+35), unos slova i nenumeričkih karaktera i slično.

Primer 2.3.4 Pravila organizacije pri zapošljavanju

Godine	Pravilo
0-16	Ne zaposliti
16-18	Može se zaposliti samo sa pola radnog vremena
18-55	Može se zaposliti sa punim radnim vremenom
55-99	Ne zaposliti

Broj validnih test slučajeva može se smanjiti sa 100 (testiranje za svaku godinu starosti) na 4 (testiranje jedne godine starosti za svaku klasu ekvivalencije, npr 10, 17, 30, 70). Nevalidni test slučajevi -5, 105.

Granične vrednosti

Testiranje pomoću klasa ekvivalencije je najosnovnija tehnika oblikovanja testova i ona nas vodi do ideje o testiranju graničnih vrednosti. Testiranje graničnih vrednosti (engl. *boundary value testing*) se fokusira na granice zato što se tu krije mnogo grešaka. Greška koju programeri često čine je pogrešno kodiranje testova nejednakosti. Primer toga je pisanje $>$ znaka umesto $\geq$ znaka.

Koraci za korišćenje testiranja graničnih vrednosti su:

1. Identifikujemo klase ekvivalencije.
2. Identifikujemo granice svake klase ekvivalencije.

3. Pravimo test slučaj za svaku graničnu vrednost birajući jednu tačku na granici, jednu tačku ispod granice i jednu tačku iznad granice.

Ispod i *iznad* su relativni termini i zavise od jedinica vrednosti podataka. Tačke ispod i iznad granice mogu biti u drugim klasama ekvivalencije i treba voditi računa da se testovi ne dupliraju. Problem nastaja kada postoji više dimenzija u klasama ekvivalencije, ili npr kada su u pitanju realni brojevi.

Primer 2.3.5 (Nastavak primera 2.3.4.) Primećuje se problem na granicama svake klase. Starost *16* je uključena u dve različite klase ekvivalencije (a takođe to važi i za starosti 18 i 55). Prvo pravilo kaže da se ne zapošljavaju osobe sa 16 godina. Drugo pravilo kaže da se osobe sa 16 godina mogu zaposliti sa pola radnog vremena. Ovo je greška u specifikaciji sistema. Ispravna tabela koja ne sadrži preklapajuće vrednosti je:

Godine	**Pravilo**
0-15	Ne zaposliti
16-17	Može se zaposliti samo sa pola radnog vremena
18-54	Može se zaposliti sa punim radnim vremenom
55-99	Ne zaposliti

Validni test slučajevi u ovom primeru su naredne vrednosti na granici ili blizu granice: (za granicu 0) {0, 1}; (za granicu 15) {14, 15, 16}; (za granicu 16) {15, 16, 17}, (za granicu 17) {16, 17, 18}; (za granicu 18) {17, 18, 19}; (za granicu 54) {53, 54, 55}; (za granicu 55) {54, 55, 56} i (za granicu 99) {98, 99, 100}. Ako se razmatra unija ovih skupova, to su vrednosti {0, 1, 14, 15, 16, 17, 18, 19, 53, 54, 55, 56, 98, 99, 100}. Međutim, neke ovako određene vrednosti dupliraju provere u okviru iste klase ekvivalencije i po potrebi se mogu eliminisati. Na primer, 1 i 14 pripadaju istoj klasi ekvivalencije, 19 i 53 i slično.

Tabele odlučivanja

Izrada tabela odlučivanja (engl. *decision table*) je tehnika za prikaz složenih poslovnih pravila u lako čitljivom obliku

pomoću koje se mogu napraviti i test slučajevi. Prvu grupu redova tabele čine uslovi nad ulazom, a drugu moguće akcije. Kolone tabele predstavljaju pravila koja jedinstvenoj kombinaciji uslova dodeljuju odgovarajuće akcije. Uslovi pravila mogu biti binarni ili sa više od dve vrednosti. Iz prvih se može direktno izvesti tačno jedan test slučaj, dok iz drugih može više njih. Izbor različitih test slučajeva iz jednog pravila može se vršiti u kombinaciji sa drugim tehnikama, kao što su klase ekvivalencije ili granične vrednosti. Kada imamo više pravila kod kojih akcija ne zavisi od vrednosti nekog uslova možemo ih spojiti u jedno pravilo (engl. *table collapsing*). Takav uslov u novom pravilu označavamo sa '-' i nazivamo nebitnim (engl. *don't care*).

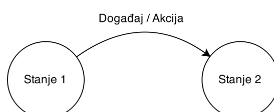
Primer 2.3.6 Banka
Klijent zahteva isplatu gotovine na bankomatu. Sistem treba da odluči da li će da odobri isplatu. Odlučivanje vrši pomoću podataka o sredstvima na računu i o dozvoljenom minusu. Način odlučivanja je prikazan nardnom tabelom.

	Pravilo 1	Pravilo 2	Pravilo 3
Uslovi			
Dovoljno sredstava na računu	Da	Ne	Ne
Dozvoljen minus	-	Da	Ne
Akcije			
Isplata odobrena	Da	Da	Ne

Prvo pravilo je dobijeno spajanjem dva pravila: kada ima dovoljno sredstava na računu, nezavisno od toga da li je minus dozvoljen ili ne, isplata se odobrava. Iz drugog pravila može se izvesti test slučaj tako što se za ulaz uzme da korisnik nema dovoljno sredstava na računu i da mu je dozvoljen minus. Zatim se izlaz iz programa poredi sa očekivanom akcijom, a to je da je isplata odobrena.

Dijagrami stanja

Dijagram stanja (engl. *state-transition diagram*) kompaktno opisuje kompleksne zahteve sistema i njegov način interakcije sa spoljašnjim svetom. Primenjuje se kod sistema čije akcije zavise od akcija izvršenih u prošlosti i koji reaguju na spoljašnje događaje.



Slika 2.7: Prikaz prelaza u okviru dijagrama stanja.

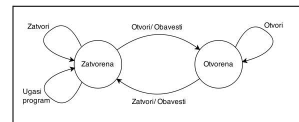
Osnovnu strukturu dijagrama čine:

Stanje Čuva znanje o prošlim događajima i definiše reakciju na buduće.

Prelaz Promena iz jednog stanja u drugo.
Događaj Nešto izvan sistema što preko interfejsa izaziva prelaz.
Akcija Operacija sistema izazvana prelazom.

U svakom trenutku sistem se nalazi u nekom od konačno mnogo stanja i čeka na neki događaj. Kombinacija stanja i događaja određuje stanje u koje sistem prelazi. Pri prelasku sistem može da izvrši još neku akciju, obično pravljenje nekih izlaza. Ovakav sistem se može modelovati konačnim automatom (engl. *finite state machine*). Dijagram stanja je jedan od načina prikaza takvog modela.

Test slučajeve možemo da generišemo obilaskom dijagrama, jer dijagram stanja predstavlja vrstu usmerenog grafa. Pri pravljenju skupova test slučajeva možemo zahtevati različite nivoe pokrivenosti, pri čemu se pravi kompromis između pokrivenosti i količine testova. Primer dobrog komprimisa je skup testova koji omogućava da se svaki prelaz ispita bar jednom. Pored toga, možemo zahtevati da se svako stanje ili svaka putanja kroz dijagram obiđu bar jednom.



Slika 2.8: Slika uz primer sa ugrađenom kasom.

Primer 2.3.7 (Ugrađena kasa) Posmatramo softverski sistem za maloprodaju koji ima ugrađenu opciju za otvaranje i zatvaranje (fioke) kase prikazan na slici 2.8. Primeri test slučajeva

- Otvori, zatvori, ugasi program
- Otvori, otvori, zatvori, zatvori, ugasi program. Ovaj test slučaj aktivira svaki prelaz datog dijagrama bar jednom.

Pri izvršavanju navedenih naredbi proverava se da li sistem reaguje u skladu sa zadatim zahtevima.

Tabele stanja

Konačni automat koji modeluje sistem se može prikazati i tabelama stanja (engl. *state transition tables*). Osnovna prednost tabela stanja jeste njihov sistematični pristup, prikazuju sve moguće kombinacije stanja i događaja. Takvim pristupom mogu da se uoče situacije u kojima ponašanje sistema nije definisano, što može da spreči pojavu grešaka. Kod tabela stanja, iz svakog reda se može direktno izvesti jedan test slučaj.

Tabele stanja postaju nepraktične ukoliko postoji veliki broj mogućih stanja i događaja.

Tabela 2.1: Tabela stanja koja odgovara dijagramu stanja sa slike 2.8.

Trenutno stanje	Događaj	Akcija	Naredno stanje
Zatvorena	otvori	obavesti	Otvorena
Zatvorena	zatvori	-	Zatvorena
Zatvorena	ugasi program	-	Zatvorena
Otvorena	otvori	-	Otvorena
Otvorena	zatvori	obavesti	Zatvorena
Otvorena	ugasi program	-	Nedefinisano

Primer 2.3.8 (Nastavak primera 2.3.6.) Tabela stanja prikazana je na slici 2.1. Test slučaj na osnovu tabele stanja: Ukoliko korisnik sistema želi da ugasi program, a kasa nije zatvorena, ponašanje je nedefinisano. Opasnost od ostavljanja otvorene kase nije navedena u dijagramu stanja 2.8, ali jeste u tabeli stanja 2.1. Moguća rešenja su da sistem upozori korisnika i spreči zatvaranje programa ili da automatski zatvori kasu.

Pogađanje grešaka

Pogađanje grešaka (engl. *error guessing*) je tehnika koja se oslanja na iskustvo i procenu testera. To je umetnost pogađanja gde bi greška mogla da bude skrivena. Za ovu tehniku ne postoje specifični alati niti uputstva.

Kako izračunati nivo pokrivenosti koda?
Postoje različiti alati za izračunavanje nivoa pokrivenosti testovima `https://stackify.com/code-coverage-tools/` Alati mogu da budu sastavni deo alata za razvoj softvera ili se mogu pokretati nezavisno:

- gcov, Cobertura, CodeCover, Coverage.py, Emma, Gretel, Hansel,JaCoCO, JCov ...
- Visual Studio Testing Tools `https://docs.microsoft.com/en-us/visualstudio/test/improve-code-quality` (opcija Analyze Code Coverage)
- EclEmma (JaCoCO za Eclipse `http://www.eclemma.org/`, ranije Emma za Eclipse)
- ...

2.3.2 Testiranje bele kutije

Testiranje bele kutije (eng. *white box testing*) podrazumeva znanje o unutrašnjoj strukturi softvera. Testove najčešće piše programer, ali može i tester. Programer/tester kreira test slučajeve na osnovu izučavanja implementacije. Ova vrsta testiranja je skupa i sprovodi se obično za sisteme kod kojih su greške skupe.

Testiranje metodama bele kutije ispituju se različite putanje kroz program. Testiranje bele kutije koristi se najčešće za pisanje jediničnih testova, ali može i za integraciono i sistemsko testiranje. Mogu se testirati putanje kroz jedinicu koda, putanje između različitih jedinica koda za vreme integracije, i putanje između podsistema za vreme sistemskog testiranja.

Osnovni koraci su

1. Razumevanje koda.
2. Pisanje testova i njihovo izvršavanje.

Zbog prvog koraka, najčešće ovu vrstu testiranja sprovode sami programeri.

Analogno ispitivanju svih kombinacija ulaza kod tehnika zasnovanih na modelu crne kutije, može se zahtevati ispitivanje svih putanja kroz program. Međutim, takav pristup je nepraktičan, a često i nemoguć zbog prevelikog broja mogućih putanja kroz program Zbog toga tehnike nastoje da omoguće kreiranje praktično prihvatljivog broja test slučajeva, ali i da obezbede visok nivo pokrivenosti. Pre početka testiranja, treba izabrati odgovarajući nivo pokrivenosti.

Pokrivenost

Pokrivenost putanja (Path Coverage) Mera prolaska kroz moguće putanje (potpuna pokrivenost: sve moguće putanje programa su izvršene bar jednom)

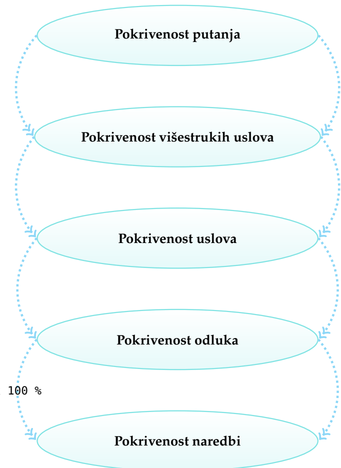
```
Path Coverage = (Number of paths exercised / Total Number of paths in the program) x 100 %
```

Pokrivenost naredbi (Statement Coverage) Mera izvršavanja naredbi programa (potpuna pokrivenost: svaka naredba programa je izvršena bar jednom)

```
Statement Coverage = (Number of Statements Exercised / Total Number of Statements) x 100 %
```

Pokrivenost grana/odluka (Branch/Decision Coverage) Mera prolaska kroz grane programa (potpuna pokrivenost: svaka odluka u programu je doneta bar jednom)

```
Decision Coverage = (Number of decisions outcomes tested / Total Number of decision Outcomes) x 100 %
```



Slika 2.9: Hijerarhija pokrivenosti

Pokrivenost uslova (Condition Coverage) Mera ispitivanja uslova programa (potpuna pokrivenost: svaki uslov u svakoj odluci je uzeo sve moguće vrednosti bar jednom)

Pokrivenost višestrukih uslova (Multiple Condition Coverage) Mera ispitivanja višestrukih uslova programa (potpuna pokrivenost: svaka moguća kombinacija uslova u svakoj odluci je ispitana bar jednom)

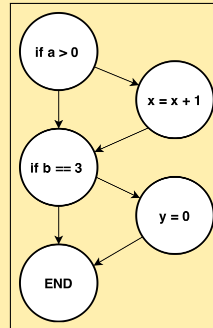
Pokrivenost funkcija (Function Coverage) Mera poziva svih funkcija programa

Primer 2.3.9 Razmorimo naredni kod5

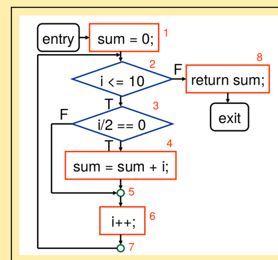
```
1 if (a > 0) {
2    x = x + 1;
3 }
4 if (b == 3) {
5    y = 0;
```

```
6 }
```

Test a=6, b=3 pokriva sve naredbe, ali ne i sve putanje



Primer 2.3.10 Razmotrimo naredni dijagram toka programa



Putanja 1 2 3 4 5 6 7 2 8 pokriva sve naredbe
Putanja 1 2 3 5 6 7 2 3 4 5 6 7 2 8 pokriva sve naredbe i svaka odluka je doneta na različite načine bar jednom

Kako izabrati putanje?

Bolje je imati više manjih putanja kroz više test primera nego jednu komplikovanu putanju. Najbolje je kada su putanje male varijacije drugih putanji. Petlja može da ima beskonačno mnogo putanja:

- Preskoči petlju
- Prođi jednom kroz petlju
- Prođi dva dva puta kroz petlju
- Ako postoji maksimalan broj prolaza n, prođi n-1 i n puta kroz petlju

Testiranje osnovnih putanja (engl. *basis path testing*) je sačinjeno od sledećih koraka:

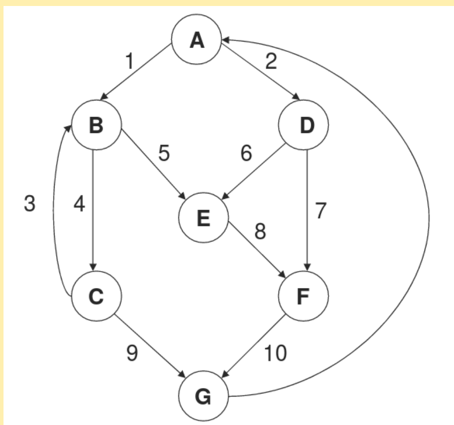
1. Izvođenje grafa toka upravljanja iz softverskog modula
2. Izračunavanje ciklomatične kompleksnosti grafa (**C**)
3. Odabir skupa **C** osnovnih putanja
4. Pravljenje test slučaja za svaku osnovnu putanju
5. Izvršavanje ovih testova

Ciklomatična kompleksnost (engl. *cyclomatic complexity*) je metrika koja se koristi da se izračuna kompleksnost softvera. To je kvantitativna mera broja linearno nezavisnih putanja kroz kôd programa. Izračunava se pomoću jednačine:

$$C = grane - čvorovi + 2 \cdot broj_povezanih_komponenti$$

Kreiranjem i izvršavanjem svih osnovnih test slučajeva, garantuje se i pokrivenost grana i pokrivenost naredbi zato što skup osnovnih putanja pokriva sve grane i čvorove grafa kontrole toka. Osnovni problem: pretpostavka o mogućnosti izbora ovih putanja i kako odabrati putanje koje su dostižne.

Primer 2.3.11 Posmatrajmo naredni graf:



C = 10 - 7 + 2 = 5, algoritam: u svakom čvoru odluke promeniti odluku, počevši od najdonjeg

Path 1: A, B, C, G.
Path 2: A, B, C, B, C, G.
Path 3: A, B, E, F, G.
Path 4: A, D, E, F, G.
Path 5: A, D, F, G.

Testiranje grafa toka podatka (engl. *data flow graph*) koristi graf kontrole toka da istraži ispravnost upotrebe podataka. Anomalije upotrebe podataka uključuju:

- Promenljive koje se koriste a nisu inicijalizovane
- Inicijalizovane promenljive koje se ne koriste
- Promenljive koje su više puta definisane pre upotrebe
- Dealokacija promenljive pre prve upotrebe
- ...

2.3.3 Metamorfno testiranje

Kako primeniti tehnike testiranja ukoliko ne znamo koji je odgovarajući izlaz za neki konkretan ulaz? Kako generisati odgovarajuće parove ulaz-izlaz?

Jedno rešenje može da bude da se koristi više implementacija algoritma čiji se izlazi nad istim test primerima porede: ako su rezultati različiti, onda bar jedan od algoritama ima grešku. Ova tehnika se često ne može primeniti pošto često ne postoji više implementacija istog algoritma ili su te implementacije previše skupe i zahtevaju puno vremena Takođe, ako se različite implementacije kreiraju od strane istih ljudi, moguće je da oni prave iste greške.

Primer 2.3.12 Računanje standardne devijacije niza brojeva

- Permutacija reda elemenata ne utiče na konačni rezultat.
- Množenje svake vrednosti sa -1, ne utiče na konačni rezultat.
- Ako se svaki broj pomnoži sa nekom konstantom, standardna devijacija tog novog niza brojeva bi trebalo da je srazmerna standardnoj devijaciji originalnog niza.
- ...

Problem prorčišta — kako odrediti da li je izlaz iz testiranog programa ispravan? U nekim situacijama proročište nije dostupno ili ga je previše teško iskoristiti. Problem proročišta je izražen u oblastima kao što su računarska grafika, konstrukcija kompilatora, mašinsko učenje. Metamorfno testiranje je metod koji testira programe bez korišćenja proročišta. Umesto toga koriste se osobine algoritma koji se testira kako bi se

generisali dodatni test primeri i automatski verifikovali njihove izlazi. Potrebno je dobro domensko poznavanje problema kako bi primena ovog vida testiranja bila efikasna.

Metamorfne relacije

Većina aplikacija ima neka svojstva takva da za određene promene ulaza mogu da se predvide neke karakteristike novog izlaza, uz poznavanje prvobitnog izlaza. Na primer, "ukoliko se ulaz poveća za n onda će izlaz da se poveća za n^2" ili "ukoliko je ulaz po modulu k isti, onda je i odgovarajući izlaz isti".

Primer 2.3.13 Konstrukcija kompilatora
Teško je verifikovati ekvivalenciju između izvornog koda i objektnog koda. Za dati izvorni kôd, mogu se dodati određene linije koje mu ne menjaju semantiku (dodavanje nule izrazu ne menja vrednost izraz, uslov `if (true) ...` takođe ne menja semantiku...). Za tako generisane programe, trebalo bi da bude generisan izvršni kôd koji se ponaša na isti način.

"Izvorni" i "naknadni" test primeri mogu se generisati na osnovu metamorfnih relacija. Opisana tehnika je bazirana na jednostavnom konceptu, nije teška za implementaciju, može se automatizovati i nezavisna je od određenog programskog jezika. Ključni korak pri ovoj tehnici je identifikacija metamorfnih relacija koje daju neku vezu između više ulaza i njihovih izlaza za dati program.

Primer 2.3.14 Mašinsko učenje
Intuitivne metamorfne relacije koje se mogu koristiti za algoritme klasifikacije:

- Konzistentnost sa afinim transformacijama
- Permutacija labela klase
- Permutacija atributa
- Dodatak neinformativnih atributa
- ...

Kako izabrati odgovarajuće metamorfne relacije? Postoje principi koji se mogu poštovati, iz perspektive testiranja metodama bele kutije ali takođe i iz testiranja metodama crne kutije.

- Dobro je izabrati metamorfne relacije koje rezultuju najvećim razlikama u izvršnim putevima između izvornih i naknadnih test primera: ako imamo veliku razliku izmedu izvršnih puteva, imamo i veliki prostor u kojem može da se izrazi propust u softveru. Međutim, često nije očigledno koje metamorfne relacije će rezultovati većoj razlici između izvršnih puteva, tako da je u tim slučajevima neophodno pokrenuti program i analizirati pokrivenost koda za tako generisane test primere.
- Relacija ekvivalentnosti kao relacija između relevantnih izlaza se pokazala kao bolja od ostalih relacija pošto je ekvivalentnost uži uslov od ostalih neekvivalentnih uslova. Metamorfne relacije sa relacijom ekvivalentnosti se lakše mogu prekršiti od ostalih relacija, čime bi trebalo da se detektuje veći broj grešaka.

2.4 Načini testiranja

Po načinu testiranja testiranje možemo podeliti na manuelno i automatsko. Svi poslovi koje obavljaju ljudi su podložni greškama. Zato je automatizacija testiranja od ključne važnosti.

Automatizaciju možemo podeliti na:

1. automatizaciju generisanja test primera,
2. automatizaciju izvršavanja test primera.

2.4.1 Automatsko generisanje test primera

Generisanje test primera se može automatizovati samo za neke vrste testiranja, na primer, testiranje robusnosti (glava **??**) ili za testiranje odsustva nekih konkretnih grešaka u kodu (glava 6).

Za većinu funkcionalnog testiranja neophodno je manuelno generisati test primere.

2.4.2 Automatsko izvršavanje test primera

S druge strane, u mnogim slučajevima je moguće automatizovati izvršavanje testiranja. Tehnike automatizacije procesa

testiranja — sastavni deo alata za razvoj softvera, alati za kontinuiranu integraciju softvera, alata za testiranje specifičnih vrsta softvera Validacija softvera testiranjem najčešće zahteva ručno izvršavanje testova.

Automatsko izvršavanje test primera u okviru alata za razvoj sofvtera najčešće je vezano za automatsko pokretanje testova jedinica koda, što je najčešće povezano i sa automatskim računanjem pokrivensti koda testovima. Postoje različita okruženja za automatsko izvršavanje test primera, i obično se ona nazivaju *xxxUnit*, pri čemo ovo *xxx* odgovara nekom programskom jeziku. Na primer, postoje CppUnit, JUnit...

Testiranje veb aplikacija
Selenium je softver otvorenog koda koji je postao skoro standard za automatizovano testiranje veb aplikacija. Selenium podržava testiranje veb aplikacija u gotovo svim dostupnim pretraživačima. Test skriptovi mogu biti pisani u različitim programskim jezicima, kao što su C#, Java, Ruby, Python i Perl i pokretani na Windows, Macintosh ili Linux platformama.

Alati za kontinuiranu integraciju softvera

Konitunirana integracija je neophodna za razvoj softvera gde članovi time unose izmene na dnevnom nivou. Svaku izmenu je potrebno objediniti sa tekućim stanjem koda, potvrditi automatskom izgradnjom koda i testiranjem kako bi se detektovale potencijalne greške u najkraćem vremenskom roku. Prilikom svakog objedinjavanja, sistem je integrisan (sve promene do tog trenutka su objedinjene u projekat), izgrađen (kôd je kompajliran u paket ili u izvršni fajl), testiran (pokreću se automatski testovi), arhiviran (verzionisan i sačuvan) i primenjen (učitan na sistem gde se može izvršavati). Ovakvim pristupom se smanjuje cena projekta, vreme razvoja i rizik pri objavljivanju novih verzija.

Postoje razni alati za kontinuiranu integraciju softvera:

- Jenkins `https://jenkins.io/`
- Buildbot
- Travis CI
- GitLab CI
- Circle CI
- TeamCity by Jetbrains
- Bamboo by Atlassian
- ....

2.4.3 Manuelno testiranje

Međutim, postoje neke vrste testiranja koje je nemoguće automatizovati. U to spadaju razni testovi vezani za validaciju softvera, na primer ispitivanje prihvatljivosti, naučivosti ili lakoće upotrebljavanja softvera.

Dodatna literatura

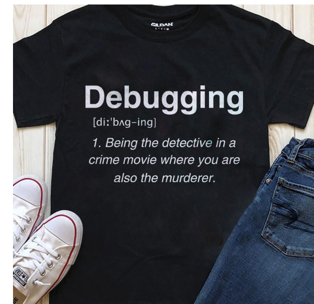
- A Practitioner's Guide to Software Test Design, Lee Copeland
-
-

Debagovanje | 3

— If Broken it is, Fix it You Should —
(Yoda, Star Wars)

3.1 Debagovanje 39

3.1 Debagovanje



Debager (engl. *Debugger*) je program koji se koristi za praćenje rada drugog programa sa ciljem pronalaženja uzroka greške. Da bi informacije koje debager daje bile razumljivije, potrebna je podrška kompajlera/linkera. Da bi debager mogao da radi, potrebna je podrška operativnog sistema i/ili hardvera.

Debageri se često koriste kroz razvojna okruženja koja im daju grafički korisnički interfejs. Udobnost GUIa je važna za lakšu upotrebljivost, ali kao i za sve ostale alate, važi da je to samo korisnički interfejs i ne treba poistovećivati GUI sa samim alatom.

Poznati debageri

- GDB — the GNU debugger
- LLDB — LLVM based debugger
- WinDBG — Microsoft
- Microsoft Visual Studio Debugger

Primer
GDB se može koristiti kroz QtCreator, Visual Studio Code, Eclipse, NetBeans... (`https://sourceware.org/gdb/wiki/GDBFrontEnds`)

3.1.1 Prevođenje sa ciljem debagovanja

Prilikom kompilacije projekta postoje razna podešavanja i opcije koje se mogu koristiti a koje služe za preciznije navođenje načina prevođenja projekta. Jedna od osnovnih opcija je mogućnost prevođenja u *release* ili *debug* modu.

Prevođenje za korisnika — *Release mode*

Release mode je prevođenje programa u izvršnu verziju namenjenu krajnjem korisniku. Ovakvo prevođenje uključuje optimizacije koje omogućavaju efikasno izvršavanje koda. Nekada je ova efikasnost vidljiva, nekada se ne može jednostavno opaziti. Optimizacijama se gubi veza sa originalnim kodom, neki delovi koda se usled optimizacija obrišu, neki se pomere, neki se prezapišu... Veličina izvršnog fajla je manja nego izvršnog fajla prevedenog u *debug* modu.

Prevođenje za programera — *Debug mode*

Debug mode je prevođenje programa u izvršnu verziju namenjenu programeru. Ovakvo prevođenje obično isključuje optimizacije sa ciljem lakšeg uparivanja izvornog i izvršnog koda. Izvršavanje ovakvog programa može da bude manje efikasno. U izvršnu verziju se umeću podaci koji su potrebni za povezivanje izvornog i izvršnog koda, tj omogućavaju debageru da precizno utvrdi koji deo koda se izvršava u datom trenutku. Veličina izvršnog fajla je veća nego izvršnog fajla prevedenog u *release* modu, zbog manjka optimizacija i viška dodatnih informacija.

DWARF je format za predstavljanje pomoćnih informacija za debagovanje, koji se koristi od strane programskih prevodioca (kao npr. GCC ili LLVM/Clang) i debagera (kao npr. GNU GDB) da bi se omogućilo debagovanje na nivou izvornog koda. Omogućava podršku za razne programske jezike kao što su C/C++ i Fortran, ali je dizajniran tako da se lako može proširiti na ostale jezike. Arhitekturalno je nezavisan i predstavlja „most” između izvornog koda i izvršne datoteke. Format DWARF se primenjuje na programe UNIX-olikih operativnih sistema, kao što su Linux i MacOS. Debageri i programski prevodioci u okviru operativnog sistema Windows ne prate standard DWARF prilikom baratanja pomoćnim informacijama za debagovanje, već prate standard Microsoft CodeView.

DWARF
Pogledati tekst o formatu DWARF u okviru master rada Đorđa Todorovića
http://www.verifikacijasoftvera.matf.bg.ac.rs/vs/predavanja/03_dinamicka_analiza/MasterRadDjordjeTodorovic.pdf

Različite izvršne verzije mogu imati različita ponašanja

Moguća su različita ponašanja *debug* i *release* izvršavanja. Na primer, može se desiti da se neke greške ispoljavaju samo u *release* verziji koda. To je situacija koja može da nastane iz različitih razloga

- *Debug* verzija koda, usled dodatnih podataka ili inicijalizacija memorije maskira grešku (dakle greška je ipak u originalnom kodu, ali je ne možemo naći na ovaj način).
- Greška se pojavljuje kao rezultat greške kompajlera prilikom neke optimizacije (greška je u kompajleru, manja verovatnoća, ali i dalje naš problem jer nam treba izvršna verzija bez greške — treba naći način kako zaobići takvu grešku).

Ometanje debagovanja

Debagovanje omogućava da se isprati logika izvršavanja programa i razume kako program funkcioniše. Mogućnost debagovanja izvršne verzije programa nije uvek poželjna osobina. Anti-debagovanje je implementacija jedne ili više tehnika koje ometaju pokušaje obrnutog inženjeringa (engl. *reverse engineering*) ili debagovanja ciljanog procesa. Poznati izdavači u sferi zaštite od kopiranja aktivno koriste anti-debagovanje. Takođe, tehnike anti-debagovanja koriste se i u malicioznim programima kako bi ih antivirusni programi teže detektifikovali i eliminisali.

3.1.2 Kako rade debageri?

Debager može da započne proces i da ga prati i debaguje, ili može da se nakači na proces koji se već izvršava. Debager omogućava izvršavanje programa instrukciju po instrukciju, omogućava postavljanje prekidnih tačaka (engl. *break point*) i izvršavanje programa do tih tačaka prekida kao i praćenje promenljivih i stanja na steku prilikom izvršavanja. Savremeni debageri omogućavaju i izmenu koda koji se izvršava i posmatranje efekta takvih izmena, debagovanje unazad, uslovne prekidne tačke i *watchpoints*.

Debager se može koristiti za svaku izvršnu verziju programa, dakle i za *release* i za *debug* verziju. Mogućnosti i informacije koje se dobijaju za debag verziju su povezane sa izvornim kodom i olakšavaju programeru da poveže stanje procesora sa izvornim kodom. Za *release* verziju *debag* informacije su često samo uvid u asemblerski kôd, na isti način kao što ih vidi i procesor. Međutim, aktivno se radi na poboljšavanju korisničkog iskustva prilikom debagovanja koda sa prisutnim optimizacijama.

Primer rada na poboljšavanju korisničkog iskustva Vaših kolega, Nikola Prica & Đorđe Todorović: `https://www.youtube.com/watch?v=1cWAmLMF1eI`
Debug info in optimized code - how far can we go? Improving LLVM debug info with function entry values
`https://www.youtube.com/watch?v=ih5v65K10M8`
Improving Debug Information in LLVM to Recover Optimized-out Function Parameters

Kada se postavi prekidna tačka u programu sa željom da se na tom mestu zaustavi program, debager umetne na to mesto u softveru instrukciju prekida ili neku nevalidnu instrukciju. Kada se prilikom izvršavanja programa naiđe na ovu instrukciju desi se hardverski izuzetak koji uzrokuje prekid. Najpre se proveri da li je prekid u listi očekivanih debager prekida (tj da li je u pitanju namerno zaustavljanje ili greška u originalnom kodu). Ukoliko je greška u originalnom kodu, onda se dopusti da se ta greška i izvrši i da program pukne. Ukoliko je u pitanju tačka prekida, prekid se prosledi na

obradu debageru koji ga onda obradi tako što na tom mestu omogući uvid u sve vrednosti fizičkih registara procesa kao i u stanje memorije. Debager prikazuje pročitane informacije o procesu povezane sa informacijama o izvornom kodu koje su nalaze u programu umetnute od strane kompajlera/linkera prilikom prevođenja programa. Ukoliko je u pitanju uslovna prekidna tačka, debager proverava uslov i u slučaju da uslov nije ispunjen, preskače se obrada prekida i samo se nastavlja dalje sa izvršavanjem procesa.

Kada korisnik poželi da nastavi sa izvršavanjem,

1. debager zameni instrukciju prekida sa originalnom instrukcijom,
2. izvrši je,
3. zameni ponovo originalnu instrukciju instrukcijom prekida,
4. prepusti ponovo dalje kontrolu programu.

3.1.3 Potrebna podrška

Pošto debageri hvataju sistemske prekide, oni su sistemski zavisni alati i za razumevanje njihovog funkcionisanja potrebno je razumeti procese i sistem prekida na odgovarajućem operativnom sistemu. Na primer, pod Linux-om za rad debagera od suštinske važnosti je sistemska funkcija **ptrace**.

Operativni sistemi — primer *ptrace*

ptrace je sistemski poziv u Unix i njemu sličnim operativnim sistemima. Ime je skraćenica od „proces tragač” (engl. *process trace*). Korišćenjem *ptrace* jedan proces može da kontroliše drugi. Time upravljački proces ima mogućnost da upravlja unutrašnjim stanjem ciljanog procesa. *ptrace* najviše koriste debageri kako bi mogli da zaustavljaju program, da posmatraju memoriju i menjaju je.

```
long ptrace
(enum __ptrace_request request, pid_t pid, void *addr, void *data);
```

Pogledati primere upotrebe ove funkcije u master radu Đorđa Todorovića ili seminarskom radu Nikole Dimitrijevića (dostupni u okviru strane kursa).

Hardver — specijalni registri za debagovanje

Za efikasno funkcionisanje, debager može da koristi i direktno neke funkcionalnosti hardvera, ukoliko su dostupne. Na primer, *watchpoints* omogućavaju praćenje vrednosti neke

promenljive u memoriji, tj da li se neka vrednost u memoriji menja ili se sa nje nešto čita. Ukoliko postoji podrška hardvera (specijalni registri koji pamte i proveravaju odgovarajuće adrese), biće podignut izuzetak koji će debager da obradi. Ukoliko to nije dostupno ili se zahteva praćenje većeg broja vrednosti nego što je to dostupno podrškom hardvera, onda debager mora da izvršava instrukciju po instrukciju i da za svaku proverava šta se dešava na traženim memorijskim lokacijama, što značajno usporava izvršavanje.

3.1.4 Vrste debagovanja

Debagovanje možemo podeliti na osnovno degabovanje i na napredne tehnike debagovanja. U osnovno debagovanje spada interaktivno debagovanje, dok napredne tehnike debagovanja obuhvataju udaljeno debagovanje i *post-mortem* debagovanje.

Interaktivno debagovanje

Interaktivno debagovanje obuhvata komunikaciju sa debagerom, tačke prekida, koračanje... Sistem za interaktivno debagovanje zahteva mogućnost kontrole toka izvršavanja programa: postavljanjem tačaka prekida (engl. *brakepoints*) izvršavanje programa se pauzira, korišćenjem komandi debagera analizira se progres programa a zatim se ponovo nastavlja sa izvršavanjem programa. Moguće je postaviti i uslovne izraze koji se proveravaju tokom izvršavanja programa i ukoliko se uslovi ispune, izvršavanje programa se zaustavlja i vrše se analize.

Udaljeno debagovanje

Pored interaktivnog debagovanja, postoji i udaljeno debagovanje (engl. *remote debugging*). Daljinsko debagovanje je postupak debagovanja programa koji se izvršava na sistemu udaljenom od debagera. Da bi proces započeo, debager mora da se poveže sa udaljenim sistemom preko mreže. Tada debager može da kontroliše izvršavanje programa na udaljenom sistemu i da sakuplja informacije o njegovom stanju. Ovo je moguće ukoliko je udaljeni sistem iste arhitekture kao i arhitektura na kojoj se debager pokreće, ili ukoliko

debager poseduje podršku za arhitekturu udaljenog sistema. Daljinsko debagovanje je posebno često za uređaje sa ugrađenim računarom kod kojih instaliranje i pokretanje debagera najčešće nije opcija, zbog ograničenja performansi ovih sistema.

Post-mortem debagovanje

Post-mortem debagovanje je postupak debagovanja programa nakon njegovog prekida. Za ovu vrstu debagovanja često se koriste datoteke jezgra koje sistem generiše. Povezane tehnike često uključuju razne tehnike praćenja i/ili analizu snimljenog stanja radne memorije programa u momentu kada je došlo do prekida. Tačan momenat prekida procesa može se ustanoviti automatski od strane sistema (npr. kada je proces završen zbog nekog odstupanja), preko instrukcija napisanih od strane programera ili eksplicitno od strane korisnika.

3.1.5 Primeri debagera

GNU GDB

GNU GDB je alat koji omogućava uvid u događanja unutar drugog programa koji se nalazi u fazi izvršavanja, ili u slučaju neregularnog prekida izvršavanja programa uvid u to šta se dešavalo pa je došlo do neregularnog prekida. GNU GDB debager takođe omogućava uvid u to šta se dešavalo sa programima i na platformama koje imaju različitu arhitekturu od arhitekture domaćina (engl. *host architecture*). Da bi se to realizovalo koristi se GNU GDB server, što se naziva udaljeno debagovanje, jer se udaljenom uređaju pristupa preko posebnih protokola.

U svrhe debagovanja programa drugih procesorskih arhitektura na ličnim računarima se takođe upotrebljava Multiarch GNU GDB koji koristi biblioteke namenjene drugačijim arhitekturama.

GDB se razvija u programskom jeziku C.

Programi koji mogu biti analizirani mogu biti napisani u raznim programskim jezicima, kao što su Ada, C, C++, Objective-C, Pascal, Fortran, Go, Java. GNU GDB debager se može pokrenuti na najpopularnijim operativnim sistemima UNIX i Microsoft Windows varijanti.

LLDB

LLDB je debager koji se aktivno razvija u okviru LLVM projekta. Ima podršku za interaktivno debagovanje.

LLDB se razvija u programskom jeziku C++.

Operativni sistemi Unix-oliki (ali i dalje manja podrška nego kod GDBa), od nedavno i za Windows, ali posebno bitan za MacOS.

Programi koji mogu biti analizirani mogu biti napisani u raznim programskim jezicima, kao što su C, C++, Objective-C, Swift.

Moguća integracija u različita razvojna okruženja.

WinDbg i Visual Studio Debugger

WinDbg je višenamenski debager za Windows operativne sisteme. Manje je poznat od debagera *Visual Studio Debugger*. Oba omogućavaju interaktivno i post-mortem debagovanje, ali WinDbg ima naprednije mogućnosti. Podržavaju jezike koji su podržani *.Net* platformom. Za *Visual Studio Debugger* se podrazume integracija u grafičko okruženje *Visual Studio*.

3.1.6 Debagovanje drugih programskih jezika

Debagovanje se obično odnosi na tradicionalne programske jezike. Na primer, za Javu postoji debager *jdb* (`https://docs.oracle.com/javase/7/docs/technotes/tools/windows/jdb.html`). Može se koristiti iz komandne linije na sličan način kao i *gdb*, može se integrisati u neko razvojno okruženje, npr Eclipse. Za javu se mogu koristiti i *gdb* i drugi debageri.

Postoje moduli koji su podrška debagovanju i u skript jezicima, na primer u Python-u modul pdb `https://docs.python.org/2.0/lib/module-pdb.html`.

Upotreba svih ovih debagera je slična, podrazumevaju se tačke prekida, koračanje i slično.

Protivnici debagera
Rob Pike (jedan od autora jezika Go)
If you dive into the bug, you tend to fix the local issue in the code, but if you think about the bug first, how the bug came to be, you often find and correct a higher-level problem in the code that will improve the design and prevent further bugs.

3.1.7 Otvoreni problemi

Pronalaženje grešaka u višenitnim aplikacijama je suštinski teško. Da bi debager omogućio debagovanje višenitnih aplikacija potrebno je da za to obezbedi dodatnu podršku. Iako u najvažnijim debagerima postoji podrška za debagovanje višenitnih aplikacija, često je ta podrška loša i nepraktična, a kada postoji komunikacija između niti to često može da zbuni debager. Pokretanje aplikacije kroz debager može da poremeti redosled izvršavanja u okviru aplikacije i na taj način zamaskira postojeće probleme.

Protivnici debagera
Postoji izvesan broj značajnih programera koji ne vole debagere

- Linus Torvalds, the creator of Linux, does not use a debugger.
- Robert C. Martin, one of the inventors of agile programming, thinks that debuggers are a wasteful timesink.

`Print` umesto debagera

- *Brian W. Kernighan, Unix for Beginners (1979)* The most effective debugging tool is still careful thought, coupled with judiciously placed print statements.
- *The author of Python, Guido van Rossum* uses print statements for 90% of his debugging.

Debagovanje je primenljivo nad aplikacijama srednje veličina, tj ne skalira uvek. Za kompleksniji i veći softver proces debagovanja može da bude previše spor.

3.1.8 `Print` umesto debagera

`Print` umesto debagera se i dalje često koristi. Jedan od osnovnih razloga je navika uz činjenicu da debageri nisu oduvek bili moćni i uspešni alati kao što su sada.

Print umesto debagera je najčešće prevaziđen stav iz više razloga. Sa printom je potrebno prevesti ponovo program (build može da traje dosta dugo). Umetanje poziva funkcije štampanja remeti memoriju i može da maskira grešku. Štampanjem se ne može videti vrednost svih interesantnih aspekata programa niti se može zaustaviti program. Štampanje je statička aktivnost (za bilo kakvu izmenu ili dodatak potrebno je ponovo prevesti i pokrenuti program), a tačke prekida se mogu postavljati i uklanjati u fazi debagovanja.

Iako prevaziđen stav, i dalje se koristi. Nekada je u pitanju neznanje. Nekada i nema alternativu: ukoliko ne postoji debager za određenu platformu ili sistem. Nekada debager ne uspeva da odradi posao jer maskira grešku pa su neophodne alternative.

Profajliranje 4

Važan deo testiranja performansi obuhvata merenje vremenske i memorijske efikasnosti programa. Ukoliko program ne zadovoljava postavljene kriterijume, potrebno je naći uzrok i izvršiti optimizaciju. Proces optimizacije je neizostavan deo razvoja softvera. Da bi se uočili delovi koda koji treba da se poboljšaju i optimizuju, koriste se pomoćni alati, profajleri, koji generišu informacije na osnovu kojih se donose odluke o optimizacijama.

4.1 Osnovni pojmovi . 47
4.2 Profajliranje uzimanjem uzoraka 49
4.3 Instrumentalizacija 49
4.4 Kompajlerski zasnovana dinamička analiza — sanitajzeri 54
4.5 Napredna analiza izvršnog programa . 55
4.6 Alat Perf 66

4.1 Osnovni pojmovi

Profajliranje predstavlja vid dinamičke analize koda čiji je rezultat skup podataka dobijen izvršavanjem programa sa određenim ulaznim podacima. Profajliranje može da se zasniva na intrumentalizaciji, tj na ubacivanju dodatnih instrukcija u program kako bi se prikupili precizni podaci o programu za vreme njegovog izvršavanja. Takođe, profajliranje može da se zasniva na uzimanju uzoraka (engl. *sampling*).

Podaci dobijeni profajliranjem predstavljaju **profil** programa. Željeni podaci o programu obuhvataju, na primer, frekvencije izvršavanja funkcija ili blokova koda, procenat utrošenog vremena u bloku koda, podatke o alokaciji memorije... Ovi podaci pomažu u otkrivanju dela koda koji se često izvršava, određivanju pokrivenosti koda datim ulazima, proširivanju skupa testova i u rešavanju još mnogih drugih problema.

Izvršnim tokom programa upravljaju konkretni ulazi i različiti ulazi daju različite rezultate profajliranja. Da bi se donosile odluke o optimizaciji na osnovu profajliranja, **važno je da izvršavanje u okviru kojeg se vrši profajliranje reflektuje realnu upotrebu programa**, odnosno da su skupljeni podaci na osnovu relevantnih ulaznih podataka ili da su skupljeni podaci na osnovu više različitih skupova ulaznih podataka. U suprotonom, mogu se propustiti prilike za optimizaciju programa ili se mogu doneti pogrešne odluke.

Primer — propuštena prilika
Razmotrimo funkciju koja vrši obradu elemenata niza algoritmom kubne složenosti i pretpostavimo da se ta funkcija u praksi koristi nad velikim nizovima. Ukoliko se program profajlira sa ulazom koji funkciji prosleđuje niz sa malim brojem elemenata, onda se profajliranjem ne može uočiti problem sa performansama koji će nastati u praksi.

Primer 2 — pogrešne odluke
Razmotrimo funkciju f koja ima naredbu grananja u okviru koje se u `then` grani odlazi na funkciju f_t, a u `else` grani na funkciju f_e. Pretpostavimo da se `then` grana izvršava 20 puta češće nego `else` grana ali da su ulazni podaci za profajliranje izabrani tako da nas vode u `else` granu. Na osnovu takvog profajliranja, može da se donese pogrešan zaključak, tj. da je potrebno optimizovati funkciju f_e, ali optimizacija te funkcije u praksi neće davati vidljive rezultate, s obzirom da se ona retko izvršava.

4.1.1 Vrste optimizacija na osnvou rezultata profajliranja

Optimizaciju na osnovu rezultata profajliranja može vršiti čovek, a može se sprovoditi i automatski. Automatska optimizacija može da napravi važna poboljšanja u efikasnosti koda, ali samo čovek može da suštinski izmeni algoritam koji se koristi u implementaciji. Obe vrste optimizacija su važne i zapravo imaju različite domete.

Automatska optimizacija može da se sprovodi u fazi izvršavanja koda ili u fazi kompilacije. Optimizacija u fazi kompilacije koristi informacije dobijene profajliranjem da bi se generisala nova efikasnija izvršna verzija. Optimizacija u fazi izvršavanja korisiti informacije koje se dobijaju profajliranjem da bi se donele odluke o tome da se neki delovi izvršnog koda optimizuju u fazi izvršavanja.

4.1.2 Podrška profajliranju

Profajliranje može da bude samo softversko, može da ima podršku u okviru hardvera, ili da bude kombinacija hardverskog i softverskog profajliranja. Podrška hardvera u profajliranju omogućava veću efikasnost profajliranja kao i veći opseg podataka dobijenih na osnovu profajliranja.

Za neka merenja neophodna je hardverska podrška, na primer, za merenje broja promašaja u keš memoriji ili za merenje utrošenog vremena zbog čekanja protočne obrade (engl. *pipline stall*) neke instrukcije.

4.1.3 Implementacija profajliranja

Postoje dve osnovne implementacije profajliranja: profajliranje uzimanjem uzoraka i profajliranje instrumentalizacijom izvršnog koda. Postizanje pravog poretka između vremena utrošenog prilikom profajliranja i same dobiti na osnovu dobijenih profila je od izuzetne važnosti. Profajliranjem se u prvim koracima dobijaju informacije na osnovu kojih je relativno jednostavno napraviti optimizacije. Međutim, vremenom, kako se sve više optimizuje program to ga je teže dodatno optimizovati. Zato se u prvim koracima profajliranja i optimizacije mogu koristiti i manje precizne tehnike, a kasnije je potrebno koristiti sve preciznije tehnike.

4.2 Profajliranje uzimanjem uzoraka

Jedan od efikasnih načina koji znatno smanjuje opterećivanje programa profajliranjem je **profajliranje uzimanjem uzoraka** (engl. *sample based profiling*).

Uzimanje uzoraka se jako često koristi. Na primer, to je podrazumevan metod profajliranja za *Visual Studio Profiling Tools*

U okviru profajliranja uzimanjem uzoraka uzimaju se „slike" programa u određenim vremenskim intervalima i od njih pravi profil programa. Za dobro izabrane intervale merenja preciznost ove metode može da bude veoma visoka. Međutim, ukoliko vremenski intervali nisu dobro odabrani može se desiti da se ne zabeleži određeni događaj.

Profajliranje uzimanjem uzoraka se najčešće koristi za pronalaženje najfrekventnijih događaja. Na primer, uzimanjem slika stanja na steku može se proceniti koja se funkcija najčešće poziva.

4.3 Instrumentalizacija

Profajliranje može da sa zasniva na **instrumentalizaciji** (engl. *instrumentation*), tj. na ubacivanju novog koda u postojeći kôd. Novi kôd ima za cilj da omogući prebrojavanje određenih događaja odnosno prikupljanje podataka koji su od interesa.

Najvažnije osobine koje dobra instrumentalizacija treba da zadovolji su:

1. Prikuplja samo **potrebne** podatke.
2. Ne utiče na funkcionalnost programa.
3. Ne usporava previše rad programa.

Prvi uslov je važan jer zahtev za previše podataka dodatno usporava program i samu njihovu obradu, dok premalo informacija može biti beznačajno. Drugi uslov ističe da ukoliko dodata instrumentalizacija utiče na funkcionalnost programa onda prikupljeni podaci neće oslikavati pravi način njegovog rada. Poslednji uslov zavisi od tipa aplikacije i njega možemo kontrolisati izborom delova programa koji se instrumentalizuju.

Usporavanje rada programa koje je posledica instrumentalizacije je posebno važno u kontekstu sistema koji treba da rade u realnom vremenu. Ovi sistemi često imaju vrlo stroga vremenska ograničenja koja se profajliranjem mogu poremetiti (prekršiti) i time izazvati štetu. Dodatno, problem mogu da budu i memorijska ograničenja uređaja jer dodatni kôd za instrumentalizaciju i njeno rukovanje može uvećati program tako da on ne može da stane na uređaj.

Instrumetalizacija se može podeliti na osnovu toga kako se nove instrukcije ubacuju u program:

- Može je izvršiti sam programer, manuelnim dodavanjem dodatnih linija na željena mesta u kodu.

- Može se izvršiti automatski u različitim fazama.

U okviru automatske instrumentalizacije:

- Istrumentalizaciju može da sprovede kompajler i/ili linker.
- Istrumentalizacija može da se ubaci u već preveden program.
- Istrumentalizacija može da se ubacuje za vreme izvršavanja programa.

4.3.1 Osnovne vrste profajliranja

Najčešće se prikupljaju informacije o količini izvršavanja određenih delova koda kako bi se prilikom optimizacije obratila pažnja baš na te delove. Ukoliko optimizujemo deo koda koji se retko izvršava, to neće značajno uticati na celokupne performanse. Najbitnije informacije su

1. Sekvence blokova koje se najčešće izvršavaju
2. Instrukcije (blokovi) koje se najčešće izvršavaju

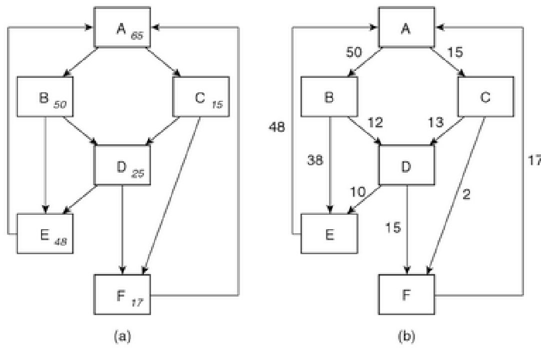
Na osnovu željenih podataka, vrste profajliranja se mogu podeliti na

1. Profajliranje putanje (engl. *path profiling*).
2. Profajliranje grana (engl. *edge profiling*).
3. Profajliranje blokova (engl. *basic-block profiling*).

U okviru profajliranja postavljamo brojače kojima utvrđujemo koliko se puta neki događaj desio prilikom izvršavanja programa. Dobijeni profili se mogu koristiti za kompajlerski zasnovane optimizacije i za utvrđivanje pokrivenosti koda testovima.

Profajliranje putanja

Profajliranje putanja je složen vid profajliranja kojim se dobijaju informacije o najčešće korišćenim putanjama kroz program. Ova vrsta profila u sebi sadrži i informacije o profilima grana i blokova. Zahteva kompleksne algoritme i najviše utiče na performanse izvršavanja prilikom profajliranja.



Slika 4.1: Profil dobijen profajliranjem blokova (a) i profajliranjem grana (b)

Profajliranje blokova

Blokovi mogu biti funkcije ili deo koda u kome se ne nalaze instrukcije grananja ili skokova. Profajliranjem blokova broji se ukupan broj izvršavanja svakog bloka. Naivni algoritam: može se ostvariti tako što se u svaki blok umeće brojač, čime se dobijaju precizne informacije o broju izvršavanja blokova, ali se i prilično usporava sistem. Ovaj način profajliranja ne daje informacije o tome koje su putanje kroz program najčešće kao ni koji su prelazi između blokova česti.

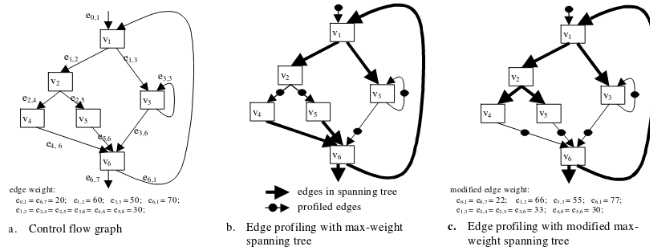
Profajliranje grana

Grana predstavlja prelazak koji se ostvaruje instrukcijom grananja ili skoka kojom se prebacuje tok izvršavanja programa iz jednog bloka u drugi. Profajliranjem grana mogu se dobiti i podaci koji se dobijaju profajliranjem blokova. Broj izvršavanja svakog bloka može se sračunati pomoću brojača grana tako što sumira sve grane koje ulaze u blok.

Naivni algoritam podrazumeva da se za svaku naredbu skoka umeće po jedan brojač. Međutim, takvo rešenje previše opterećuje program. Profajliranje grana može da se implementira značajno efikasnije, pod pretpostavkom da se radi u fazi kompilacije programa. Jedno takvo rešenje prvi je teoretski uveo Donald Knut (engl. *Donald Knuth*) i on je pokazao da je broj umetnutih brojača u njegovom rešenju minimalan.

Knutov algoritam profajliranja grana:

1. Najpre je neophodno napraviti graf kontrole toka (engl. *control flow graph*), u kojem svaki čvor predstavlja blok instrukcija, a grana naredbu skoka ili grananja.



Slika 4.2: Profajliranje grana

2. Za ovaj graf potrebno je napraviti razapinjuće stablo (engl. *spanning tree*). Broj grana u razapinjućem stablu je $v - 1$, gde je v broj čvorova grafa.
3. Granama koje ne pripadaju dobijenom stablu treba dodati brojač.
4. Broj izvršavanja grana koje ne sadrže brojač se može izračunati na osnovu sračunatih vrednosti

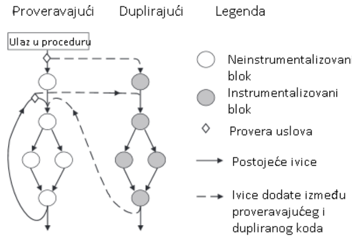
Podsetnik
Razapinjuće stablo je podskup grafa G koji sadrži sve čvorove pokrivene sa minimalnim mogućim brojem grana. Razapinjuće stablo ne sadrži cikluse i ne može biti nepovezano. Po definiciji, svaki povezan neusmeren graf G ima najmanje jedno razapinjuće stablo.

Knutovo rešenje može istrumentalizovati isti broj grana ali na različite načine, jer standardni algoritam računanja razapinjućeg stabla može vratiti različita stabla u zavisnosti od redosleda obrade grana. Optimalno razapinjuće stablo je ono stablo kod kojeg se grane najveći broj puta izvršavaju (ali baš tu informaciju zapravo tražimo!). Tomas Bal (engl. *Tomas Ball*) i Džejms Larus (engl. *James R. Larus*) su osmislili način kako da se proceni koji skup grana je optimalan, tj kako da se korišćenjem jednostavne statičke analize instrumentalizuju one grane za koje se predviđa da će se najmanje puta izvršiti.

4.3.2 Instrumentalizacija u kombinaciji sa uzimanjem uzoraka

Uzimanje uzoraka može da se radi i dodavanjem brojača koji proveravaju da li pratimo instrumentalizovan kod ili ostajemo u originalnom kodu. Na taj način, ne izvršava se stalno instrumentalizovan kod, već se izvršavanje orginalnog i instrumentalizovanog koda smenjuje.

Jedno rešenje je da se pored originalnog koda napravi i duplikat koji će sadržati instrumentalizovani kôd i njega ćemo nazvati duplirani kôd. Orginalni kôd zovemo proveravajući zato što se u njemu ispituje uslov koji, ukoliko je ispunjen, treba kontrolisano da pređe u duplirani kôd. Ovim uslovom kontrolišemo koliko vremena će se izvršavati svaki od ova dva koda. Kada izvršavanje pređe u duplirani kôd ono tu ostaje ograničeno vreme a zatim se vraća u proveravajući. Trenutak



Slika 4.3: Proveravajući i duplirajući kod

prelaska iz proveravajućeg u duplirani kôd se može inicirati hardverski, putem operativnog sistema ili softverski.

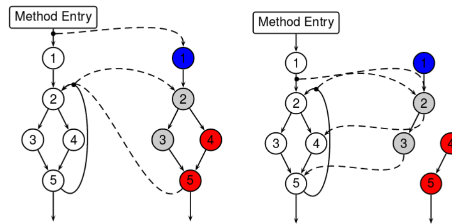
Prelazak iz proveravajućeg u duplirajući kod može da se vrši na osnovu fiksiranih vremenskih intervala (engl. *time based sampling*)između dva uzimanja uzoraka: tajmer postavlja bit, uslov proverava li je taj bit postavljen. Period uzimanja uzoraka je ograničen frekvencijom tajmera što nije praktično za veoma česte događaje. Kada istekne vreme, sledeći uzorak će biti uzet tek kada se ponovo ispita uslov prelaska.

Prelazak iz proveravajućeg u duplirajući kod može da se vrši i na osnovu vrednosti brojača (engl. *counter-based sampling*). Čuva se brojač za prelaske koji se dekrementira, kada brojač dođe do nule, uzima se uzorak i resetuje se brojač. Očekuje se da je cena ovoga mala jer se brojač može držati u kešu. Na ovaj način je obezbeđeno da će se uzorci uzimati proporcionalno broju provera.

Najčešće je profajliranje na osnovu brojača preciznije u odonosu na profajliranje na osnovu fiksiranih vremenskih intervala.

Prikazani algoritam povećava potrebnu memoriju kao i vreme kompilacije. To se može unaprediti instrumentalizacijom samo delova koda i postoje proširene verzije ovog algoritma koje ne prave celu kopiju koda već samo onih delova koji su vezani za instrumentalizaciju, jer, za rekonstrukciju profila nije potrebno instrumentalizovati svaki blok ili svaku granu. Delimičnim dupliranjem može se smanjiti upotreba memorije i vreme kompilacije sa potpunim zadržavanjem preciznosti dobijenih informacija.

Uzimanje uzoraka može se ostvariti i bez dupliranja koda. U tom slučaju, dodaje se provera ispred svakog instrumentalizovanog čvora. Na ovaj način narušava se svojstvo proporcionalnosti (broj provera nije proporcionalan broju



Slika 4.4: Delimično dupliranje

ulazaka u instrumentalizovani kod) i dodaje se veće opterećenje prilikom izvršavanja. Rezultati nisu isti kao kod punog ili delimičnog dupliranja, ali je preciznost slična.

4.4 Kompajlerski zasnovana dinamička analiza — sanitajzeri

Instrumentalizacija može da se vrši i sa drugačijim ciljevima, tj cilj ne mora da bude samo profajliranje i optimizacija koda. Na primer, česta upotreba instrumentalizacije je da se pronađu greške u radu sa memorijom ili u radu sa nitima. Primer toga su santajzeri, koji se koriste u fazi razvoja programa kao pomoć u detektovanju grešaka.

4.4.1 Detektovanje grešaka u radu sa memorijom i nitima

`AddressSanitizer` (ASan) u okviru Clang-a i gcc-a imaju za cilj detektovanje prekoračenja bafera i korišćenje već oslobođene memorije. Instrumentalizacija je za ovakve upotrebe drugačija i obuhvata i izmene u adresnim prostorima izvršavajućeg programa U proseku, instrumetalizacija u ovom slučaju povećava vreme izvršavanja za 73% a upotrebu memorije za 340%.

Memorijski sanitajzer (engl. *MemorySanitizer* — *MSan*) u okviru Clang-a je detektor čitanja neincijalizovane memorije. Neincijalizovane vrednosti javljaju se u trenutku kada se sadržaj memorije koja je dodeljena steku ili hipu pročita pre nego što je u nju upisana ispravna vrednost. Ovaj tip sanitajzera otkriva slučajeve u kojima takve vrednosti utiču na izvršavanje programa.

`ThreadSanitizer` alat u okviru Clang-a pomaže u detketovanju problema u radu sa nitima. Koristi kompajlersku instrumetalizaciju i odgovarajuću run-time biblioteku. Tipično uspori izvršavanje od 5 do 15 puta. Memorijsko dodatno opterećenje je od 5 do 10 puta.

4.5 Napredna analiza izvršnog programa

Postoje različiti alati koji omogućavaju naprednu analizu programa u fazi izvršavanja. Razmotrićemo alate Valgrind i Perf. Valgrind je primer profajlera koji vrši instrumentalizaciju nad izvršnim kodom programa. Valgrind is an instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail. You can also use Valgrind to build new tools. Perf je primer profajlera koji omogućava pristup hardveru i profajliranje uzimanjem uzoraka.

4.5.1 Platforma Valgrind

Dinamička analiza obuhvata analizu korisničkog programa u izvršavanju. Analiza izvršnog programa (binarna analiza, engl. *binary analysis*) obuhvata analizu na nivou mašinskog koda, snimljenog ili kao objektni kôd (nepovezan) ili kao izvršni kôd (povezan). Primer platforme koja omogućava naprednu analizu je Valgrind. Valgrind je besplatan i otvorenog koda.

Usporenje koje Valgrind nameće izvršavanju programa je od 5 do 100 puta, u zavisnosti od konkretnog alata.

Valgrind distribucija sadrži naredne alate:

- detektor memorijskih grešaka *Memcheck*,
- praćenje rada dinamičke memorije *Massif*,
- profajler keš memorije *Cachegrind*,
- profajler funkcija *Callgrind*,
- detektor grešaka niti *Helgrind* i *DRD*.

Ime potiče iz nordijske mitologije, ne od engleske reči *value* (vrednost) + grind (mlevenje). Valgrind, the main entrance to Valhalla in Norse mythology; Valgrind, a programming tool named after this mythical gate. Valgrind = 'fence of the fallen ones' From Old Norse Valgrind, from valr ("dead warrior") + grind ("gate").
Izgovara se Velgrind (između a i e): The "Val" as in the world "value". The "grind" is pronounced with a short 'i'.
Dostupnost Valgrinda
Valgrind radi na sledećim arhitekturama:

Linux - *x86, AMD64, ARM, ARM64, PPC32, PPC64, S390X, MIPS32, MIPS64*
Solaris - *x86, AMD64*
Android - *ARM, ARM64,x86* (4.0 i novije), *MIPS32*
Darwin - *x86, AMD64*(Mac OS X 10.12)

4.5.2 Kako radi Valgrind?

Valgrind se može koristiti i kao alat za pravljenje novih alata. Svi *Valgrind* alati rade na istoj osnovi, iako informacije koje se emituju variraju.

Jezgro Valgrind-a + alat koji se dodaje = Alat Valgrind-a

Alat za dinamičku analizu koda se kreira kao dodatak, pisan u programskom jeziku C, na jezgro *Valgrind*-a. Jezgro *Valgrind*-a omogućava izvršavanje klijentskog programa, kao i snimanje izveštaja koji su nastali prilikom analize samog programa. Informacije koje se emituju mogu se iskoristiti za otklanjanje grešaka, optimizaciju koda ili bilo koju drugu svrhu za koju je alat dizajniran.

Svaki *Valgrind*-ov alat je statički povezana izvršna datoteka koja sadrži kôd alata i kôd jezgra. Izvršna datoteka *valgrind* predstavlja program omotač koji na osnovu --tool opcije bira alat koji treba pokrenuti. *Valgrind*-ovo jezgro prvo inicijalizuje podsistem kao što su menadžer adresnog prostora, i njegov unutrašnji alokator memorije i zatim učitava klijentovu izvršnu datoteku. Potom se inicijalizuju *Valgrind*-ovi podsistemi kao što su translaciona tabela, aparat za obradu signala, raspoređivač niti i učitavaju se informacije za debagovanje klijenta, ukoliko postoje. Od tog trenutka *Valgrind* ima potpunu kontrolu i počinje sa prevođenjem i izvršavanjem klijentskog programa.

Nijedan deo koda klijenta se ne izvršava u svom izvornom obliku. Može se reći da *Valgrind* vrši JIT (*Just In Time*) prevođenje mašinskog koda programa u mašinski kôd programa dopunjen instrumentalizacijom. Alat u originalni kôd umeće operacije u svrhu instrumentalizacije, zatim se takav kôd prevodi. Prevođenje se vrši dinamički.

Proces prevođenja se sastoji iz raščlanjivanja originalnog mašinskog koda u odgovarajuću međureprezentaciju (engl. *intermediate representation*, skraćeno IR) koji se kasnije instrumentalizuje sa alatom i ponovo prevodi u novi mašinski kôd. Rezultat ovog procesa se naziva **translacija**, koja se čuva u memoriji i koja se izvršava po potrebi. Jezgro troši najviše vremena na sam proces pravljenja, pronalaženja i izvršavanja translacije.

Valgrind deli originalni kôd u sekvence koje se nazivaju **osnovni blokovi**. Osnovni blok je pravolinijska sekvenca mašinskog

koda, na čiji se početak skače, a koja se završava sa skokom, pozivom funkcije ili povratkom u funkciju pozivaoca. Svaki kôd programa koji se analizira ponovo se prevodi na zahtev, pojedinačno po osnovnim blokovima, neposredno pre izvršavanja samog bloka. Veličina osnovnog bloka je ograničena na maksimalno šesdeset mašinskih instrukcija.

Izazovi uspešnog rada Valgrinda sastoje se, pre svega, u sklapanju dva procesa u jedan: klijentski program i program alata se sklapaju u jedan proces. Mnogi resursi se dele između ova dva programa, kao što su registri ili memorija. Sistemski pozivi se izvršavaju bez posredstva *Valgrind*-a, zato što jezgro *Valgrind*-a ne može da prati njihovo izvršavanje u samom jezgru operativnog sistema.

4.5.3 Translacija

U nastavku su opisani koraci koje *Valgrind* izvršava prilikom analize programa. Postoji osam faza translacije. Sve faze obavlja jezgro *Valgrind*-a, osim instrumentalizacije koju obavlja alat *Valgrind*-a.

Koraci translacije:

Disasembliranje - Proces prevođenja mašinskog koda programa u ekvivalentni asemblerski kôd. *Valgrind* vrši prevođenje mašinskog koda programa u interni skup instrukcija koja se nazivaju međukod (engl. *intermediate representation*, skr. IR). U ovoj fazi međukod je predstavljen stablom. Ova faza je zavisna od arhitekture na kojoj se izvršava.

Optimizacija 1 - Prva faza optimizacije linearizuje *IR* reprezentaciju. Primenjuju se neke standardne optimizacije programskih prevodilaca kao što su uklanjanje redudantnog koda, eliminacija podizraza...

Instrumentalizacija - Blok koda u *IR* reprezentaciji se prosleđuje alatu, koji može proizvoljno da ga transformiše. Prilikom instrumentalizacije alat u zadati blok dodaje dodatne *IR* operacije, kojima proverava ispravnost rada programa.

Optimizacija 2 - Druga faza optimizacije je jednostavnija od prve, uključuje izračunavanje matematičkih izraza koji se mogu izvršiti pre faze izvršavanja i uklanjanje mrtvog koda.

Izgradnja stabla - Linearizovana *IR* reprezentacija se konvertuje natrag u stablo radi lakšeg izbora instrukcija.

Odabir instrukcija - Stablo *IR* reprezentacije se konvertuje u listu instrukcija koje koriste virtualne registre. Ova faza se takođe razlikuje u zavisnosti od arhitetkure na kojoj se izvršava.

Alokacija registara - Virtualni registri se zamenjuju stvarnim. Po potrebi se uvode prebacivanja u memoriju. Ne zavisi od platforme, koristi poziv funkcija koje nalaze iz kojih se registara vrši čitanje i u koje se vrši upis.

Asembliranje - Izabrane instrukcije se kodiraju na odgovarajući način i smeštaju u blok memorije. Ova faza se takođe razlikuje u zavisnosti od arhitekture na koji se izršava.

4.5.4 Memcheck

Memcheck je najpoznatiji alat Valgrinda. *Memcheck* je alat koji detektuje memorijske greške korisničkog programa. Kako ne vrši analizu izvornog koda već mašinskog, *Memcheck* ima mogućnost analize programa pisanog u bilo kom jeziku. Program koji radi pod kontrolom *Memcheck*-a je obično dvadeset do sto puta sporiji nego kada se izvršava samostalno, zbog translacije koda. Izlaz iz programa je dopunjen izlazom koji dodaje sam alat *Memcheck*, koji se ispisuje na standardnom izlazu za greške.

Pokretanje Memcheck-a ostvaruje se poziv sledeće linije u terminalu:

```
valgrind --tool=memcheck ./main
```

Za programe pisane u jezicima C i C++, detektuje uobičajne probleme u radu sa memorijom, na primer:

- upisivanje podataka van opsega hipa i steka
- pristupanje memoriji koja je već oslobođena
- neispravno oslobađanje hip memorije, kao što je duplo oslobađanje hip blokova ili neuparenog korišćenja funkcija `malloc/new/new[]` i `free/delete/delete[]`
- curenje memorije
- korišćenje vrednosti koje nisu inicijalizovane ili koje su izvedene od drugih neinicijalizovanih vrednosti
- preklapanje parametara prosleđenih funkcijama (npr. preklapanje `src` i `dst` pokazivača kod funkcije `memcpy`).

Primer koda sa upotrebom neinicijalizovane memorije

```
int main {
  int x;
  printf("%d\n",x);
  return 0;
}
```

```
==7070== Memcheck, a memory error detector
==7070== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==7070== Using Valgrind-3.14.0.GIT and LibVEX; rerun with -h for copyright info
==7070== Command: ../main
==7070==
==7070== Conditional jump or move depends on uninitialised value(s)
==7070==    at 0x4E814CE: vfprintf (vfprintf.c:1660)
==7070==    by 0x4E8B3D8: printf (printf.c:33)
==7070==    by 0x400548: main (in /export/main)
==7070==
```

...

```
==7070== Conditional jump or move depends on uninitialised value(s)
==7070==    at 0x4E8161C: vfprintf (vfprintf.c:1660)
==7070==    by 0x4E8B3D8: printf (printf.c:33)
==7070==    by 0x400548: main (in /export/main)
==7070==
x = 0
==7070==
==7070== HEAP SUMMARY:
==7070==     in use at exit: 0 bytes in 0 blocks
==7070==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==7070==
==7070== All heap blocks were freed -- no leaks are possible
==7070==
==7070== For counts of detected and suppressed errors, rerun with: -v
==7070== Use --track-origins=yes to see where uninitialised values come from
==7070== ERROR SUMMARY: 6 errors from 6 contexts (suppressed: 0 from 0)
```

- Prve tri linije se štampaju prilikom pokretanja bilo kog alata koji je u sklopu *Valgrind*-a
- Sledeći deo nam pokazuje poruke o greškama koje je *Memcheck* pronašao u programu.
- Poslednja linija prikazuje rezime svih grešaka koje je alat pronašao i štampa se po završetku rada.

U programu neinicijalizovana promenljiva može više puta da se kopira, *Memcheck* prati sve to, beleži podatke o tome, ali ne prijavljuje grešku. U slučaju da se neinicijalizovane vrednosti koriste na način da od te vrednosti zavisi dalji tok programa ili ako je potrebno prikaziti vrednosti neinicijalizovane promenljive, *Memcheck* prijavljuje grešku.

Primer koda sa upotrebom sistemskog poziva

```
#include <unistd.h>
int main()
{
  char* arr  = malloc(10);
  int*  arr2 = malloc(sizeof(int));
  write( 1 /* stdout */, arr, 10 );
  exit(arr2[0]);
}
```

```
rtrk@rtrkw579-lin:/export/valgrind$ ./vg-in-place --leak-check=yes --show-reachable=yes --track-origins=yes ../main
==8038== Memcheck, a memory error detector
==8038== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==8038== Using Valgrind-3.14.0.GIT and LibVEX; rerun with -h for copyright info
==8038== Command: ../main
==8038==
==8038== Syscall param write(buf) points to uninitialised byte(s)
==8038==    at 0x4F26390: __write_nocancel (syscall-template.S:81)
==8038==    by 0x4005F6: main (in /export/main)
==8038==  Address 0x5200040 is 0 bytes inside a block of size 10 alloc'd
==8038==    at 0x4C2AC23: malloc (vg_replace_malloc.c:299)
==8038==    by 0x4005CE: main (in /export/main)
==8038==  Uninitialised value was created by a heap allocation
==8038==    at 0x4C2AC23: malloc (vg_replace_malloc.c:299)
==8038==    by 0x4005CE: main (in /export/main)
==8038==
==8038== Syscall param exit_group(status) contains uninitialised byte(s)
==8038==    at 0x4EFC109: _Exit (_exit.c:32)
==8038==    by 0x4E7316A: __run_exit_handlers (exit.c:97)
==8038==    by 0x4E731F4: exit (exit.c:104)
==8038==    by 0x400603: main (in /export/main)
==8038==  Uninitialised value was created by a heap allocation
==8038==    at 0x4C2AC23: malloc (vg_replace_malloc.c:299)
==8038==    by 0x4005DC: main (in /export/main)
==8038==
==8038==
==8038== HEAP SUMMARY:
==8038==     in use at exit: 14 bytes in 2 blocks
==8038==   total heap usage: 2 allocs, 0 frees, 14 bytes allocated
==8038==
==8038== 4 bytes in 1 blocks are still reachable in loss record 1 of 2
==8038==    at 0x4C2AC23: malloc (vg_replace_malloc.c:299)
==8038==    by 0x4005DC: main (in /export/main)
==8038==
==8038== 10 bytes in 1 blocks are still reachable in loss record 2 of 2
==8038==    at 0x4C2AC23: malloc (vg_replace_malloc.c:299)
==8038==    by 0x4005CE: main (in /export/main)
==8038==
==8038== LEAK SUMMARY:
==8038==    definitely lost: 0 bytes in 0 blocks
==8038==    indirectly lost: 0 bytes in 0 blocks
==8038==      possibly lost: 0 bytes in 0 blocks
==8038==    still reachable: 14 bytes in 2 blocks
==8038==         suppressed: 0 bytes in 0 blocks
==8038==
==8038== For counts of detected and suppressed errors, rerun with: -v
==8038== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

- Prva greška prikazuje da parametar *arr* sistemskog poziva *write()* pokazuje na neinicijalizovanu vrednost.
- Druga greška prikazuje da je podatak koji se prosleđuje sistemskom pozivu *exit()* nedefinisan.
- Takođe, prikazane su i linije u samom programu gde se ove vrednosti koriste.
- U okviru LEEK SUMMARY dato je da postoje „still reachable" bajtovi

Memcheck se može koristiti za detektovanje curenja memorije. *Memcheck* beleži podatke o svim dinamičkim blokovima koji su alocirani tokom izvršavanja programa pozivom funkcija `malloc()`, `new()` i dr. Kada program prekine sa radom, *Memcheck* tačno zna koliko memorijskih blokova nije

```
(1)  RRR ------------> BBB
(2)  RRR ---> AAA ---> BBB
(3)  RRR               BBB
(4)  RRR      AAA ---> BBB
(5)  RRR ------?-----> BBB
(6)  RRR ---> AAA -?-> BBB
(7)  RRR -?-> AAA ---> BBB
(8)  RRR -?-> AAA -?-> BBB
(9)  RRR      AAA -?-> BBB

- RRR: skup pokazivaca
- AAA, BBB: memorijski blokovi u dinamičkoj memoriji
- --->: pokazivač
- -?->: unutrašnji pokazivač (eng. interior-pointer)
```

Slika 4.5: Primer pokazivača na memorijski blok

oslobođeno. Ako je opcija *--leak-check* uključena, za svaki neoslobođeni blok *Memcheck* određuje da li je moguće pristupiti tom bloku preko pokazivača.

Postoje dva načina da pristupimo sadržaju nekog memorijskog bloka preko pokazivača:

- Preko pokazivača koji pokazuje na početak memorijskog bloka.
- Preko pokazivača koji pokazuje na sadržaj unutar memorijskog bloka.

Drugi način je često problematičan.

Postoji nekoliko načina da nastanu pokazivači koji pokazuje na unutrašnjost nekog memorijskog bloka.

- Postojao je pokazivač koji je pokazivao na početak bloka, ali je namerno (ili nenamerno) pomeren da pokazuje na unutrašnjost bloka.
- Postoji odgovarajuća vrednost u memoriji, koja je u potpunosti nepovezana i slučajna.

Na slici 4.5 je prikazano devet mogućih slučajeva kada pokazivači pokazuju na neke memorijske blokove. *Memcheck* objedinjuje neke od ovih slučajeva, tako da dobijamo naredne četiri kategorije.

Još uvek dostupan (engl. *Still reachable*) Ovo pokriva primere 1 i 2. Pokazivač koji pokazuje na početak bloka ili više pokazivača koji pokazuju na početak bloka su pronađeni. Zato što postoje pokazivači koji pokazuju na memorijsku lokaciju koja nije oslobođena, programer može da oslobodi memorijsku lokaciju neposredno pre završetka izvršavanja programa.

```
LEAK SUMMARY:
   definitely lost: 47 bytes in 3 blocks
   indirectly lost: 0 bytes in 0 blocks
     possibly lost: 0 bytes in 0 blocks
   still reachable: 0 bytes in 0 blocks
        suppressed: 0 bytes in 0 blocks
```

Slika 4.6: Rezime curenja memorije

```
16 bytes in 1 blocks are definitely lost in loss record 2 of 3
   at 0x4847838: malloc (vg_replace_malloc.c:299)
   by 0x4007B4: main (in /home/aleksandrak/main)

19 bytes in 1 blocks are definitely lost in loss record 3 of 3
   at 0x4847838: malloc (vg_replace_malloc.c:299)
   by 0x40073C: main (in /home/aleksandrak/main)
```

Slika 4.7: Izveštaj o curenju memorije

Defininitivno izgubljen (engl. *Definitely lost*) Ovo se odnosi na slučaj 3. Ovo znači da je nemoguće pronaći pokazivač koji pokazuje na memorijski blok. Blok je proglašen izgubljenim, zauzeta memorija ne može da se oslobodi pre završetka programa, jer ne postoji pokazivač na nju.

Indirektno izgubljen (engl. *Indirectly lost*) Ovo pokriva slučajeve 4 i 9. Memorijski blok je izgubljen, ne zato što ne postoji pokazivač koji pokazuje na njega, nego zato što su svi blokovi koji ukazuju na njega izgubljeni. Na primer, ako imamo binarno stablo i koren je izgubljen, sva deca čvorovi su indirektno izgubljeni. S obzirom na to da će problem nestati ako se popravi pokazivač na definitivno izgubljen blok koji je uzrokovao indirektno gubljenje bloka, *Memcheck* neće prijaviti ovu grešku ukoliko nije uključena opcija *--show-reachable=yes*.

Moguće izgubljen (engl. *Possibly lost*) Ovo su slučajevi od 5 do 8. Pronađen je jedan ili više više pokazivača na memorijski blok, ali najmanje jedan od njih pokazuje na unutrašnjost memorijskog bloka. To može biti samo slučajna vrednost u memoriji koja pokazuje na unutrašnjost bloka, ali ovo ne treba smatrati u redu dok se ne razreši slučaj pokazivača koji pokazuje na unutrašnjost bloka.

Ako je uključena opcija *--leak-check=yes*, *Memcheck* će prikazati detaljan izveštaj o svakom definitivno ili moguće izgubljenom bloku, kao i o tome gde je on alociran. *Memcheck* nam ne može reći kada, kako ili zašto je neki memorijski blok izgubljen. Generano, program ne treba da ima nijedan definitvno ili moguće izgubljen blok na izlazu.

4.5.5 Massif

Massif je alat za analizu hip memorije korisničkog programa. Postoje određeni scenariji curenja memorije koji ne spadaju u klasične, i takve propuste ne može detektovati *Memcheck*. Ovo se dešava zato što memorija nije zapravo izgubljena, pokazivač na nju i dalje postoji, ali ona se više ne koristi. Programi sa ovakvim tipom curenja memorije dovode do nepotrebnog korišćenja određene količine memorije tokom svog izvršavanja. *Massif* pomaže u otkrivanju baš ovakvih curenja memorije. *Massif* ne daje samo informaciju o tome koliko hip memorije se koristi, već i detaljne informacije koje upućuju na to koji deo programa je odgovoran za alociranje te memorije

Program koji se izvršava pod alatom *Massif* radi veoma sporo. Nakon završetka rada, sve statistike su ispisane u fajl. Podrazumevani fajl u koji se piše je *massif.out.<pid>*, gde *<pid>* predstavlja ID procesa. Može se promeniti fajl u kome će se ispisivati komandom *--massif-out-file*. Da bi informacije koje je *Massif* sakupio mogli da vidimo u čitljivom formatu, koristimo *ms_print*. Na primer, ako imamo fajl *massif.out.1234*:

ms_print massif.out.1234

ms_print prozvodi graf koji pokazuje trošenje memorije tokom izvršavanja programa, kao i detaljne informacije o različitim tačkama programa koje su odgovorne o alokaciji memoriji. Korišćenje različitih skripti za prezentaciju rezultata je namerno, jer odvaja sakupljanje podataka od prezentacije, kao i da je moguće dodati nov način prikaza podataka u svakom trenutku.

Alat *Massif* meri samo hip memoriju, odnosno memoriju koja je alocirana `malloc`, `calloc`, `realloc`, `memalign`, `new`, `new[]` i nekoliko drugih sličnih funkcija. To znači da *Massif* ne meri memoriju koja je alocirana sistemskim pozivima kao što su `mmap`, `mremap` i `brk`.

4.5.6 Cachegrind

Cachegrind je alat koji omogućava softversko profajliranje keš memorije tako što simulira i prati pristup keš memoriji mašine na kojoj se program, koji se analizira, izvršava. Takođe, može se koristiti i za profajliranje izvršavanja grana. On simulira

memoriju mašine, koja ima prvi nivo keš memorije podeljene u dve odvojene nezavisne sekcije: *I1* - sekcija keš memorije u koju se smeštaju instrukcije i *D1* - sekcija keš memorije u koju se smeštaju podaci. Drugi nivo keš memorije koju *Cachegrind* simulira je objedinjen - *L2*. Ovaj način konfiguracije odgovara mnogim modernim mašinama.

Postoje mašine koje imaju i tri ili četiri nivoa keš memorije. U tom slučaju, *Cachegrind* simulira pristup prvom i poslednjem nivou. Generalno gledano, *Cachegrind* simulira *I1*, *D1* i *LL* (poslednji nivo keš memorije). *Cachegrind* prikuplja naredne statističke podatke o programu koji analizira (skraćenice koje se koriste dalje u tekstu su date u zagradama):

Ir - ukupan broj izvršenih instrukcija
I1mr - broj promašaja čitanja instrukcija iz keš memorije nivoa *I1*
ILmr - broj promašaja čitanja instrukcija iz keš memorije nivoa *LL*
Dr - ukupan broj čitanja memorije
D1mr - broj promašaja čitanja nivoa keš memorije *D1*
DLmr - broj promašaja čitanja nivoa keš memorije *LL*
Dw - ukupan broj pisanja u memoriji
D1mw - broj promašaja pisanja u nivo keš memorije *D1*
DLmw - broj promašaja pisanja u nivo keš memorije *LL*

Statistika se prikuplja na nivou celog programa, kao i pojedinačno na nivou funkcija. Na modernim mašinama *L1* promašaj košta oko 10 procesorskih ciklusa, *LL* promašaj košta oko 200 procesorskih ciklusa.

Program koji želimo da analiziramo pokrećemo *Cachegrind*-om. Na standardni izlaz se ispisuju sumarne informacije, dok se detaljne informacije upisuju u `cachegrind.out.<pid>`, gde *pid* predstavlja jedinsteveni identifikator procesa koji se izvršio. Alat grupiše sve troškove po fajlovima i funkcijama kojima ti troškovi pripadaju. Globalna statistika se računa prilikom prikaza rezultata. Na ovaj način se štedi vreme. Funkcije koje simuliraju pristup keš memoriji se pozivaju jako često, tako da bi dodavanje još nekoliko instrukcija koje sabiraju, znatno usporilo i ovako sporo izvršavanje alata.

Generisani fajl je čitljiv, ali je bolje je koristiti alat *cg_annotate* koji nam lepše prikazuje detaljan izveštaj. Alat *cg_merge* sumira u jednu datoteku više izlaza dobijenih višestrukim pokretanjem *Cachegrind*-a nad istim programom. Tu datotetku koristimo kao ulaz u *cg_annotate*. Alat *cg_diff* pravi razliku

```
==28165== Cachegrind, a cache and branch-prediction profiler
==28165== Copyright (C) 2002-2015, and GNU GPL'd, by Nicholas Nethercote et al.
==28165== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==28165== Command: trunk/cachegrind/tests/clreq
==28165==
==28165==
==28165== I   refs:      102,190
==28165== I1  misses:        728
==28165== LLi misses:        721
==28165== I1  miss rate:    0.71%
==28165== LLi miss rate:    0.71%
==28165==
==28165== D   refs:       39,582  (26,533 rd   + 13,049 wr)
==28165== D1  misses:      1,853  ( 1,358 rd   +    495 wr)
==28165== LLd misses:      1,498  ( 1,040 rd   +    458 wr)
==28165== D1  miss rate:     4.7% (   5.1%     +    3.8%  )
==28165== LLd miss rate:     3.8% (   3.9%     +    3.5%  )
==28165==
==28165== LL refs:         2,581  ( 2,086 rd   +    495 wr)
==28165== LL misses:       2,219  ( 1,761 rd   +    458 wr)
==28165== LL miss rate:      1.6% (   1.4%     +    3.5%  )
```

Slika 4.8: Izveštaj alata *Cachegrind*

između više izlaza iz alata *Cachegrind*, koje takođe možemo da koristimo kao ulaz u alat *cg_annotate*.

4.5.7 Callgrind

Callgrind je alat koji generiše listu poziva funkcija korisničkog programa u vidu grafa. U osnovnim podešavanjima sakupljeni podaci sastoje se od broja izvršenih instrukcija, njihov odnos sa linijom u izvršnom kodu, odnos pozivaoc/pozvan između funkcija, kao i broj takvih poziva. Dodatno može da vrši analizu upotrebe keš memorije i profajliranje grana programa slično kao kod Cachegrinda.

Kada se koristi Valgrind, vrši se kompilacija programa za debug režim, kako bi bile prisutne debag informacije koje olakšavaju praćenje stanja programa. Ali, za razliku od normalnog korišćenja debagera ili Valgrinda, za pozve Callgrinda (kao i Cachegrinda) kompilacija treba da bude sa optimizacijom, jer nema smisla profajlirati kôd koji je drugačiji od onoga koji će se normalno izvršavati.

Podaci koji se analiziraju se zapisuju u fajl nakon završetka rada programa i alata. Podržane komande su:

callgrind_annotate - na osnovu generisanog fajla prikazuje listu funkcija. Za grafičku vizuelizaciju preporučuju se dodatni alati *KCachegrind*, koji olakšava navigaciju ukoliko *Callgrind* napravi veliku količinu podataka.

callgrind_control - ova komanda omogućava interaktivnu kontrolu i nadgledanje programa prilikom izvršavanja. Mogu se dobiti informacije o stanju na steku, može se takođe u svakom trenutku generisati profil.

Alat *Cachegrind* sakuplja podatke, odnosno broji događaje koji se dešavaju direktno u jednoj funkciji. Ovaj mehanizam sakupljanja podataka se naziva **ekskluzivnim**. Alat *Callgrind* proširuje ovu funkcionalnost tako što propagira cenu funkcije preko njenih granica. Na primer, ako funkcija `foo` poziva funkciju `bar`, cena funkcije `bar` se dodaje funkciji `foo`. Kada se ovaj mehanizam primeni na celu funkciju, dobija se slika takozvanih **inkluzivnih** poziva, gde cena svake funkcije uključuje i cene svih funkcija koje ona poziva, direktno ili indirektno.

Zahvaljujući grafu poziva, može da se odredi, počevši od `main` funkcije, koja funkcija ima najveću cenu poziva. Pozivaoc/pozvan cena je izuzetno korisna za profilisanje funkcija koje imaju više poziva iz raznih funkcija, i gde imamo priliku optimizacije našeg programa menjajući kôd u funkciji koja je pozivaoc, na primer redukovanjem broja poziva.

4.5.8 Helgrind i DRD

Helgrind otkriva greške sinhronizacije prilikom upotrebe modela niti *POSIX*. *DRD* je alat za detekciju grešaka u *C* i *C++* programima koji koriste više niti, radi za svaki program koji koristi niti *POSIX* standarda ili koji koriste koncepte koji su nadograđeni na ovaj standard. Alati *DRD* i *Helgrind* ne koriste iste algoritme za otkrivanje grešaka, samim tim ne otkrivaju iste tipove greška, iako imaju veliki broj poklapanja.

Greške koje alati otkrivaju u korišćenju API-ja POSIX niti:

- greške u otključavanju muteksa - kada je muteks nevažeći, nije zaključan ili je zaključan od strane druge niti
- greške u radu sa zaključanim muteksom - uništavanje nevažećeg ili zaključanog muteksa, dealokacija memorije koja sadrži zaključan muteks
- greške prilikom korišćenja funkcije `pthread_cond_wait` - prosleđivanje nezaključanog, nevažećeg ili muteksa koga je zaključala druga nit
- greške sa `pthread barrier` - nevažeća ili dupla inicijalizacija, čekanje na objekat koji nije nikada inicijalizovan...

Greške koje alati otkrivaju

- Mrtvo blokiranje kao posledica problema u redosledu zaključavanja
- Pristup memoriji bez adekvatnog zaključavanja i sinhornizacije
- DRD — zadržavanje katanca i lažno deljenje

4.6 Alat Perf

...

Literatura

- Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design) by Jim Smith and Ravi Nair
- Thomas Ball and James R. Larus. 1994. Optimally profiling and tracing programs. ACM Trans. Program. Lang. Syst. 16, 4 (July 1994), 1319-1360. DOI=http://dx.doi.org/10.1145/183432.183527
  https://www.classes.cs.uchicago.edu/archive/2017/fall/32001-1/papers/ball-larus-profiling.pdf
  http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.27.5085&rep=rep1&type=pdf
  ftp://ftp.cs.wisc.edu/wwt/micro96.pdf
- Thomas Ball, Peter Mataga, and Mooly Sagiv. 1998. Edge profiling versus path profiling: the showdown. In Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '98). ACM, New York, NY, USA, 134-148. DOI=http://dx.doi.org/10.1145/268946.268958
- http://valgrind.org/
- http://valgrind.org/docs/manual/manual.html
- https://clang.llvm.org/docs/AddressSanitizer.html
- Literatura na srpskom
  - Đorđe Todorović, master rad, Podrška za naprednu analizu promenljivih lokalnih za niti pomoću alata GNU GDB http://www.racunarstvo.matf.bg.ac.rs/MasterRadovi/2018_09_01_DjordjeTodorovic/rad.pdf
  - Nikola Prica, master rad, Podrška za profajliranje softvera za uređaje sa ugrađenim računa-

rom `http://www.racunarstvo.matf.bg.ac.rs/MasterRadovi/2016_06_01_Nikola_Prica/rad.pdf`

- Aleksndra Kardžić, master rad, Alat Valgrind - implementacije konvencije FPXX za arhitekturu MIPS `http://www.racunarstvo.matf.bg.ac.rs/MasterRadovi/2017_03_30_Aleksandra_Karadzic/rad.pdf`
- Seminarski radovi, dostupni na strani kursa

Statička verifikacija softvera

Pregledi koda | 5

5.1 Šta se pregleda? . . 71
5.2 Vrste pregleda koda 73

Statička analiza je analiza koda bez njegovog izvršavanja sa ciljem pronalaženja defekata u kodu. Statička analiza može biti u obliku pregleda ili automatizovana.

Pregledi koda (engl. *code review*) obuhvataju ljudske kontrole koda najčešće pre nego što kôd uđe u glavni repozitorijum. Pregledima se otkrivaju razne vrste grešaka. Cilj pregleda je povećanje kvaliteta koda, kako smanjivanjem broja grešaka, tako i po pitanju poštovanja pravila kodiranja i dokumentovanja koda. Pregledima se ostvaruju i razni drugi ciljevi. Pregledi ne mogu da garantuju da greške neće promaći.

5.1 Šta se pregleda?

Pregledi daju odgovore na razna pitanja:

- Da li postoje neke očigledne logičke greške u kodu?
- Ako se posmatraju zahtevi koji su postavljeni, da li su svi slučajevi pokriveni i u potpunosti implementirani?
- Da li su novi testovi koji se dodaju dovoljni za nov kôd?
- Da li postojeći testovi treba da se promene da bi se uzele u obzir nove promene?
- Da li novi kôd prati opšti stil programiranja na projektu?
- Da li je moguće bolje dizajnirano ili kvalitetnije implementirano rešenje (npr polimorfizam umesto if-ova, postojeće implementacije funkcija ili bibliotečke funkcije umesto novo-napisanih funkcija...)?

Primer 5.1.1 Razmotrimo naredni kôd.

```
int x, y, Q, p, A, b, c, D, d;
scanf("%d%d%d%d",&x,&y,&p,&d);
Q = 2*(x + y);
c = x*y;
A = 2*(p+b);
D = p*b;
```

Pregledom se mogu dati sledeće preporuke:

- Potrebno je preimenovati sve promenljive. Imenovanje je nekonzistentno (velika i mala slova) i nedeskriptivno (nije jasan smisao promenljivih).
- Kôd treba da bude konzistentno formatiran (u nekim izrazima postoji a u nekim ne postoji blanko između operatora).
- Koristi se neinicijalizovana promenljiva `b` — verovatno je u pitanju greška.
- Izračunavanje je ponovljeno, odgovara obimu i površini pravougaonika — bilo bi dobro izračunavanje izdvojiti u odgovarajuće funkcije.

Uporedite kvalitet narednog koda sa kvalitetom prethodnog koda. Pored modularnosti, čitljivosti, razumljivosti i moguće ponovne upotrebe izdvojenih funkcija narednog koda, on omogućava i produbljivanje modularizacije i uvođenje apstrakcija ukoliko su potrebne.

```
1  int obim_pravougaonika(int a, int b) {
2    return 2*(a+b);
3  }
4
5  int povrsina_pravougaonika(int a, int b) {
6    return a*b;
7  }
8  ...
9  int a1, b1, obim1, povrsina1;
10 int a2, b2, obim2, povrsina2;
11 scanf("%d%d", &a1, &b1);
12 obim1 = obim_pravougaonika(a1, b1);
13 povrsina1 = povrsina_pravougaonika(a1, b1);
14 scanf("%d%d", &a2, &b2);
15 obim2 = obim_pravougaonika(a2, b2);
16 povrsina2 = povrsina_pravougaonika(a2, b2);
```

Primer 5.1.2 Razmotrimo naredni kôd.

```
if(a>b) if(b>c) return a;
else if(a>c) return a; else return c;
else if(b>c) return b; else return c;
```

- Formatiranje je jako nepregledno, potrebno je propustiti kôd kroz odgovarajući alat za formatiranje.
- Srandardi kodiranja najčešće nameću i upotrebu zagrada prilikom korišćenja if naredbe, čak i kada if sadrži samo jednu naredbu. Jedan od ciljeva te kn-

vencije je da se izbegnu greške kôd višestrukih if-ova koje nastaju neispravnim uparivanjem if i else grana.

- Kako je u pitanju kôd koji računa maksimum tri broja, bolje koristiti funkciju `max`, npr `max(a,max(b,c))` ili standardni algoritam za računanje maksimuma.

Primer 5.1.3 Razmotrimo naredni kôd.

```
int strCmp(char* s1, char* s2)
{
    while(*s1 && (*s1 == *s2))
    {
        s1++;
        s2++;
    }
    return *s1 - *s2;
}
```

Sama implementacija bi mogla da se poboljša na razne načine (`const` u argumentima, provera pre prvog dereferenciranja kako ne bi došlo do dereferenciranja null pokazivača, kastovanje da rezultat funkcije bude tipa `int`). Ipak, najbolje bi bilo ne koristiti tu funkciju već koristiti bibliotečku funkciju `strcmp`.

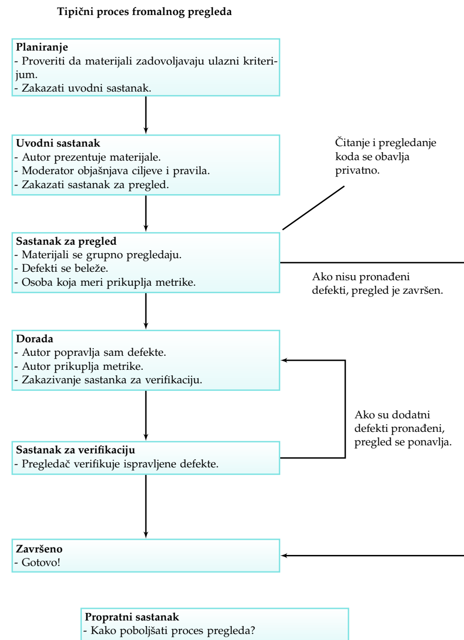
5.2 Vrste pregleda koda

U skladu sa nivoom formalnosti pregledi mogu da budu više ili manje formalni.

5.2.1 Formalni pregledi

Formalni pregledi (engl. *formal inspections*) obuhvataju grupne sastanke (3-6 osoba) na kojima se diskutuje o kodu i rade pregledi (često odštampanog koda). To je dosta skupo i vremenski zahtevno, sve manje se koristi jer iako se na ovaj način može pronaći najveći broj defekata u kodu, ovo zahteva previše vremena i angažovanja, a većina firmi to ne može da priušti. Proces koji je potrebno sprovesti da bi se uradio formalni pregled koda prikazan je na slici 5.1.

Važnost pregleda
Case study
One of our customers set out to test exactly how much money the company would have saved had they used peer review in a certain three-month, 10,000-line project with 10 developers. They tracked how many bugs were found by QA and customers in the subsequent six months. Then they went back and had another group of developers peer-review the code in question. Using metrics from previous releases of this project they knew the average cost of fixing a defect at each phase of development, so they were able to measure directly how much money they would have saved.
The result: Code review would have saved half the cost of fixing the bugs. Plus they would have found 162 additional bugs.



Slika 5.1: Tipičan proces rada za „formalni“ pregled. Artefakti dobijeni tokom pregleda nisu prikazani. To su log defekata, beleške sa sastanaka i log metrika. Neki pregledi takođe imaju završnu anketu na propratnom sastanku.

5.2.2 Neformalni pregledi

Neformalni pregledi mogu da budu više i manje neformalni. Osnovne vrste neformalnih pregleda:

- pregled preko ramena (engl. *over-the-shoulder reviews*),
- pregled preko mejla (engl. *e-mail pass-around*),
- pregled preko alata za pregled koda (engl. *tool-assisted reviews*),
- programiranje u paru (engl. *pair-programming*).

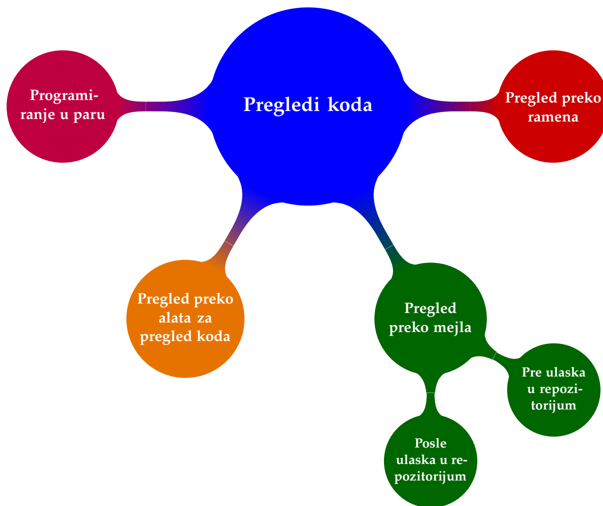
Osnovne vrste neformalnih pregleda prikazane su na slici 5.2.

Pregled preko ramena

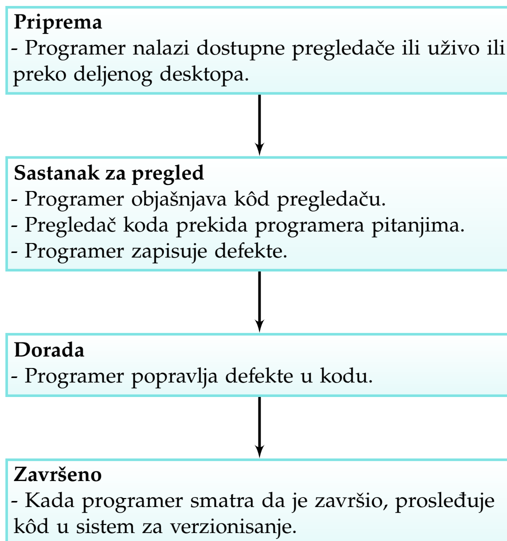
Najčešći i najneformalniji vid pregleda: programer objašnjava pregledaču šta je u kodu i zašto. Može da se obavi i preko interneta i deljenog desktopa, tj ne mora uživo.

Pregled preko ramena je najjednostavniji vid pregleda, najmanje organizaciono zahtevan, dodatno omogućava i razmenu ideja koje ne bi bile razmenenjene pisanim putem.

Problemi: nemoguće je ispratiti šta je pregledano, a šta nije, moguće je propustiti neku izmenu ikao je primećeno da



Slika 5.2: Osnovne vrste neformalnih pregleda



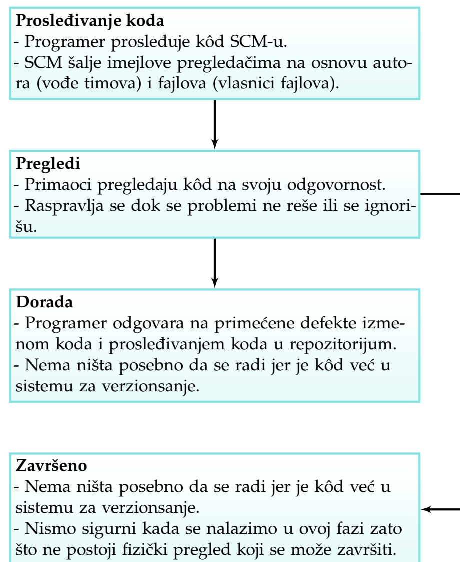
Slika 5.3: Tipičan proces rada pregleda preko ramena.

treba da se uradi, i iako se konstatuju neki defekti i ako se sprovede akcija da se ti defekti isprave, moguće je da se to uradi pogrešno ili da se uvedu novi defekti (nakon pregleda nema ponovnih provera)

Pregled preko mejla

U ovom scenariju izmena se pošalje mejlom pregledačima. Ova vrsta pregleda je lako primenljiva i kada je pregledač na drugoj lokaciji. Pregled preko mejla može da se odigra bude pre nego što kôd uđe u repozitorijum ili automatski da se pošalje nakon što kôd uđe u repozitorijum. I jedna i druga varijanta ima svoje prednosti i mane.

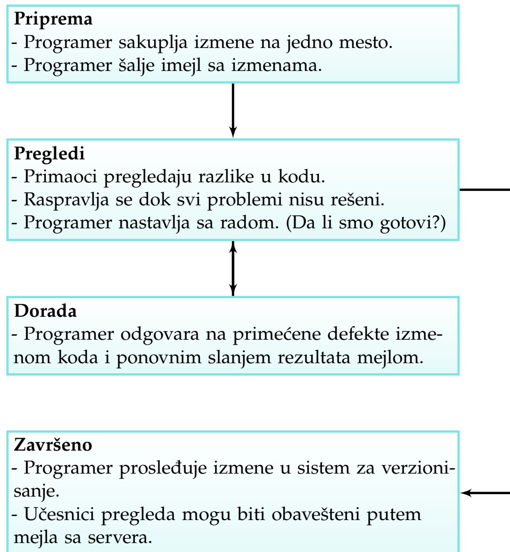
Dugo je ovo bio jedan od najčešćih vrsta pregleda. Prevaziđen, jer se sada koriste različiti alati koji prevazilaze probleme koji na ovaj način nastaju.



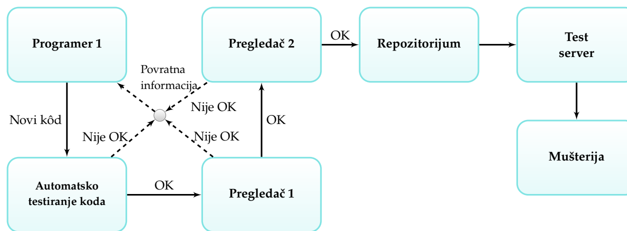
Slika 5.4: Tipičan proces pregleda preko mejla za kôd koji je ubačen u sistem za verzionisanje. Ove faze u stvarnosti nisu tako jasno razdvojene zato što ne postoji opipljiv predmet pregledanja.

Pregled preko alata za pregled koda

Pregled preko alata za pregled koda može biti različitog nivoa formalnosti. Sastavni deo većine agilnih metodologija razvoja softvera. Pregledi obuhvataju provere koda od strane jednog ili više programera pre nego što kôd uđe u repozitorijum. Za preglede su obično zaduženi iskusniji programeri. Veliki broj alata za podršku: Phabricator, Gerrit, Collaborator, GitLab ... Primer pregleda preko alata Phabricator dat je na slici 5.8.



Slika 5.5: Tipičan proces pregleda preko mejla za kôd koji nije ubačen u sistem za verzionisanje. Ove faze u stvarnosti nisu tako jasno razdvojene zato što ne postoji opipljiv predmet pregledanja.



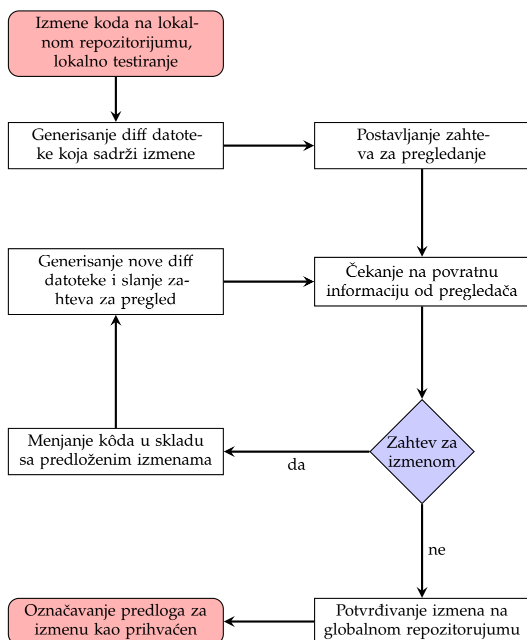
Slika 5.6: Pregledanje koda pomoću alata za automatski pregled i od strane programera u timu.

Programiranje u paru

Programiranje u paru se može smatrati specijalnom vrstom pregleda. Programiranje u paru nije uvek prisutno pa se ova vrsta pregleda ne može uvek koristiti. Programiranje u paru vodi ka kvalitetnijem kodu, ali nekada programeri koji rade zajedno isto razmišljaju i zajedno previđaju greške pa su zato eksterni pregledi ipak neophodni.

5.2.3 Uticaj pregleda

Pravilo (zahtev) da se kôd pregleda pre nego što se ubaci u repozitorijum garantuje da kôd ne može da uđe u repozitorijum nepregledan. Znanje da će kôd biti pregledan od strane nekoga u timu, utiče pozitivno i na programere tako da se učini dodatan trud da se sve proveri, dobro dizajnira i da se



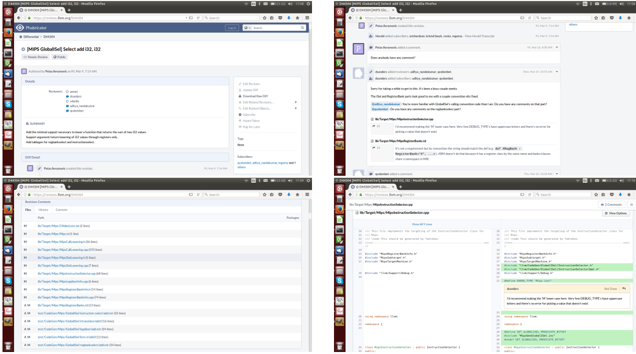
Slika 5.7: Proces pregleda

poštuju pravila kodiranja. Pregledi se rade da se provere one stvari koje mašina (automatsko testiranje) ne može da proveri i sprečava loše odluke i loša rešenja da zagađuju osnovnu liniju razvoja (štiti vas od drugih i druge od vas!).

Pregledi koda omogućavaju razmenu znanja u oba pravca, pre svega kroz mentorisanje novih programera. Kada se pridruže novi članovi timu potrebno je da ih neko iskusniji poduči. Pregledi pomažu konverzaciju o kodu. Često timovi imaju svoje skriveno znanje o kodu koje se ispolji tek za vreme pregleda. Pregledi koda pomažu programeru da savlada celokupan kôd projekta, kao i da brže usvoji nove tehnologije i tehnike. Kako pregledi izlažu programera novim idejama i tehnologijama, na taj način programer uči da stvara sve bolji kôd.

Iako su pregledi prvenstveno od strane iskusnijih programera koji pregledaju posao mlađih programera, pregledi treba da se vrše i u drugom smeru. Na primer, novi članovi tima, sa svežim pogledom na stvari i novom perspektivnom, mogu da otkriju neke stare propuste i loše delove koda na koje su se svi navikli i zbog navike ih ne primećuju.

Svi članovi tima imaju koristi od pregleda bez obzira na razvojnu metodologiju. Agilni timovi, dodatno, mogu da imaju



Slika 5.8: Fabrikator

dodatne koristi jer njihov posao se na taj način decentralizuje u okviru tima. Suština je da ne postoji jedinstvena ličnost koja zna specifičnosti nekog dela koda, već su svi u sve upućeni. Pregledi omogućavaju i pomažu širenje znanja o kodu u okviru tima.

Zašto je decentralizacija važna? Niko ne voli da bude jedinstveni kontakt za deo koda (npr, ne može da ode na duži odmor zbog toga). Takođe, niko ne voli da treba da preuzme rad na tuđem kodu, posebno ako je u pitanju nekakva hitna situacija. Pregedi deljenjem znanja kroz tim omogućavaju da svaki član tima može da preuzme i nastavi svaki posao.

5.2.4 Pravila efikasnog pregledanja

- Pregledaj između 200 i 400 linija koda od jednom, nikada više od 400.
- Imaj za cilj brzinu pregledanja koja je manja od 300-500 linija po satu. Ako se pregleda više od toga, kvalitet pregleda verovatno neće biti zadovoljavajući.
- Planiraj dovoljno vremena za odgovarajuće, sporo pregledanje, u trajanju od 60-90 minuta, ali nikako više od 90 minuta.
- Postaraj se da autori obeleže izmene pre nego što pregled počne.
- Napravi ciljeve pregledanja koda koji se mogu kvantifikovati i prati metrike kako bi mogao da unaprediš svoj proces pregledanja.
- Koristi liste provera koje treba da uradiš, jer se na taj način značajno popravlja rezultat pregleda i za autora i za pregledača. Na primer, u toj listi mogu da budu naredne provere: da li je kôd ispravno formatiran, da li je imenovanje promenljivih adekvatno, da li su u kodu mogle da budu korišćene bibliotečke funkcije...
- Proveri da su uočeni defekti stvarno i popravljeni.
- Neguj dobru kulturu pregleda koda u kojoj se pronalaženje defekata gleda pozitivno.

- Budi svesan efekta „Velikog brata” (Programer može steći utisak da ga neko stalno posmatra, pogotovo ako radi sa alatima za pregledanje. Može da misli da će statistike biti iskorišćene protiv njega i može se fokusirati na poboljšanje statistika umesto na poboljšanje koda, što svakako nije dobro. U tom smislu, vodi računa o tonu i načinu izražavanja prilikom pisanja komentara, komentariši samo kôd i moguće izmene u kodu i nikada nemoj komentarisati lične osobine programera koji je pisao kôd.)
- Ukoliko ne možeš da postigneš pregled celog koda, pogledaj bar njegov deo (zbog benefita koji donosi „ego” efekat — svako ulaže dodatni trud kada zna da će njegov kôd neko pregledati).
- Usvoji proces pregleda koda koji koristi alate za pregled koda.

Literatura

Linkovi na literaturu

- Metode za pregled koda i njihov značaj
- Best Kept Secrets of Peer Code Review
- Why code reviews matter (and actually save time!)
- 11 proven practices for more effective, efficient peer code review
- Overcoming Pre-Commit Code Review Challenges

Simboličko izvršavanje | 6

6.1 Simboličko izvršavanje kroz primer 81
6.2 Istorija, alati, stablo izvršavanja 82
6.3 Izazovi simboličkog izvršavanja . . 84
6.4 Principi dizajna i konkoličko izvršavanje 86
6.5 Strategije obilaska puteva 90
6.6 Modelovanje memorije 94
6.7 Eksplozija broja stanja i putanja . . 101
6.8 Rešavači 104
6.9 Binarni kôd 104

Simboličko izvršavanje je tehnika statičke verifikacije softvera koja ima za cilj automatsko generisanje test primera i automatsko pronalaženje grešaka u programu. U okviru simboličkog izvršavanja umesto praćenja konkretnih vrednosti promenljivih prilikom izvršavanja programa, prate se simboličke vrednosti i grade se simboličke formule koje odgovaraju vrednostima promenljivih u programu. Takođe, prate se uslovi putanja kroz koje se prolazi.

6.1 Simboličko izvršavanje kroz primer

Simboličko izvršavanje predstavlja izvršavanje program sa simbolima, tj pratimo simbolička stanja umesto konkretnih ulaza. Izvršavamo puno putanja simultano, kada izvršavanje neke putanje može da se nastavi na više načina, pravimo nove putanje i dodajemo uslove nad simboličkim vrednostima. Kada izvršavamo jednu putanju zapravo simuliramo veliki broj testova s obzirom da razmatramo sve ulaze koji prolaze kroz tu putanju. Primer konkretnog i simboličkog izvršavanja dat je na slici 6.1.

```
1 int uporedi(int a, int b) {
2   int o = 2*a + 2*b;
3   int p = a*b;
4   int r;
5   if (o > p)
6     r = o;
7   else r = p;
8   return r;
9 }
```

```
1 a = 5, b = 10
2 o = 30
3 p = 50
4 r = ?
5 30 > 50 ?
6
7 r = 50
8
9 ...
```

```
1 a = a_in, b = b_in
2 o = 2*a_in + 2*b_in
3 p = a_in * b_n
4
5
6 o>p and r = o OR
7 not(o>p) and r = p
8
9 ...
```

Slika 6.1: Kod (levo), konkretno izvršavanje (sredina) i simboličko izvršavanje (desno).

6.2 Istorija, alati, stablo izvršavanja

Naučni rad koji je najviše citiran: James C. King. 1976. **Symbolic execution and program testing.** Commun. ACM 19, 7 (July 1976), 385-394.
Ima i drugih sličnih i značajnih radova iz tog perioda
Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. 1975. **SELECT — a formal system for testing and debugging programs by symbolic execution**. International conference on Reliable software. ACM, USA, 234-245.
Lori A. Clarke. 1976. **A program testing system**. ACM annual conference (ACM '76). ACM, USA, 488-491.
Leon J. Osterweil and Lloyd D. Fosdick. 1976. **Program testing techniques using simulated execution**. 4th symposium on Simulation of computer systems (ANSS '76), IEEE Press, USA, 171-177.

Primeri alata zasnovanih na simboličkom izvršavanju

- KLEE (Stanford) — Open source, runs on top of LLVM, found lots of problems in open-source software
- SAGE — Microsoft internal tool, symbolic execution to find bugs in file parsers - e.g., JPEG, DOCX, PPT, etc.
- Pex symbolic execution for .NET
- Cloud9 parallel symbolic execution, also supports threads
- CUTE (UC Berkeley) and jCUTE (symbolic execution for Java)
- Java PathFinder (NASA) - symbolic execution
- S^2E (EPFL) — LLVM based platform
- SymDroid - symbolic execution on Dalvik Bytecode
- Kleenet - testing interaction protocols for sensor network

U Majkrosoftu, za vreme razvoja Windows 7, 30% grešaka je bilo otkriveno simboličkim izvršavanjem. Ove greške nisu bile otkrivene drugim analizama koda niti testiranjem. Upotreba simboličkog izvršavanja u testiranju Majkrosoftovih proizvoda jedna je od bitnih prekretnica u kvalitetu njihovih proizvoda.

Simboličko izvršavanje je nastalo još 70tih godina prošlog veka. Međutim, tehnika je nastala pre svog vremena i za praktičnu upotrebljivost simboličkog izvršavanja bilo je potrebno da prođe još 30 godina kako bi se računarstvo dovoljno razvilo i kako bi bili rešeni razni problemi koji su se tada odmah pojavili.

Proboj su napravili alati

- DART — autori Godefroid and Sen, PLDI 2005 — uvođenje dinamičkog izvršavanja u simboličko izvršavanje
- EXE — autori Cadar, Ganesh, Pawlowski, Dill, and Engler, CCS 2006 — STP: podrška za teoriju nizova

Primena simboličkog izvršavanja uključuju

- Pronalaženje grešaka
- Generisanje test primera
- Otkrivanje nedostižnih putanja
- Generisanje invarijnati programa, automatske ispravke programa....

Simbolička stanja preslikavaju promenljive u simboličke vrednosti. Uslov putanje (eng. *path condition*) je formula (bez kvantifikatora) nad simboličkim ulazima koja sadrži sve odluke koje su do te prilike donete. Sve putanje programa zajedno formiraju stablo izvršavanja. Uprošćen primer stabla izvršavanja koje prikazuje uslove putanja dat je na slici 6.2.

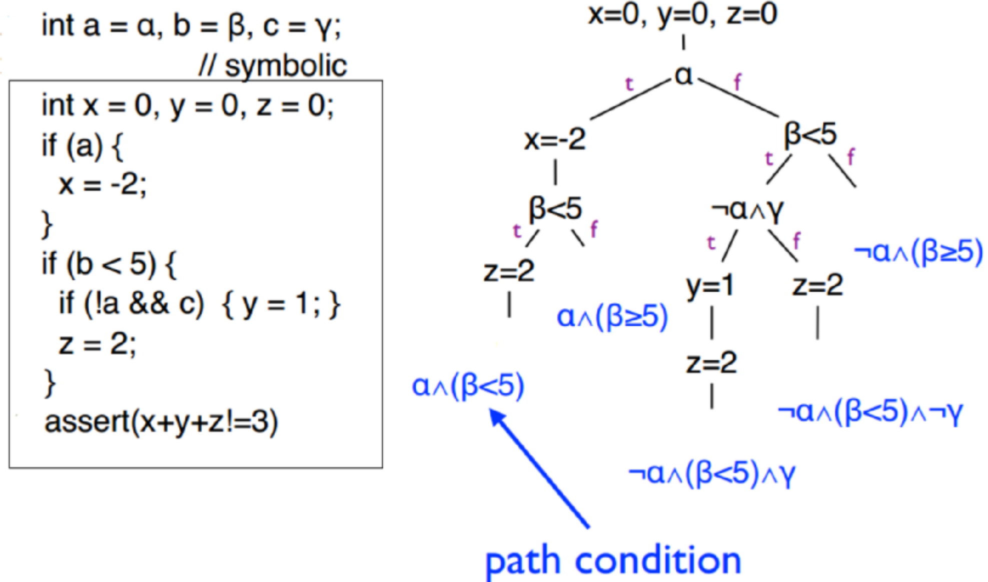
6.2.1 Simboličko stablo izvršavanja

U nastavku teksta biće korišćena sledeća notacija:

- $stmt$ - sledeća instrukcija koju je neophodno izvršiti,
- σ - simboličko skladište,
- π - ograničenja na tekućem putu.

U zavisnosti od vrednosti $stmt$ izvršavanje napreduje po sledećim pravilima:

- $x = e$: ažurira se simboličko skladište σ tako što se vezuje x sa novom simboličkom vrednošću e_s koja se dobija simboličkim evaluiranjem izraza e
- `if` e `then` s_1 `else` s_2: prave se nove dve grane sa novim stanjima i prave se nova ograničenja puta $\pi \wedge e_s$ i $\pi \wedge \neg e_s$
- `assert(`e`)`: vrši se provera vrednosti e:



Slika 6.2: Primer stabla izvršavanja: kôd (leov) i uslovi za svaku putanja izvršavanja (desno)

- Ako je $\neg e \wedge \pi$ nezadovoljivo, uslov je uvek tačan.
- Ako je $\neg e \wedge \pi$ zadovoljivo, traže se vrednosti ulaza.

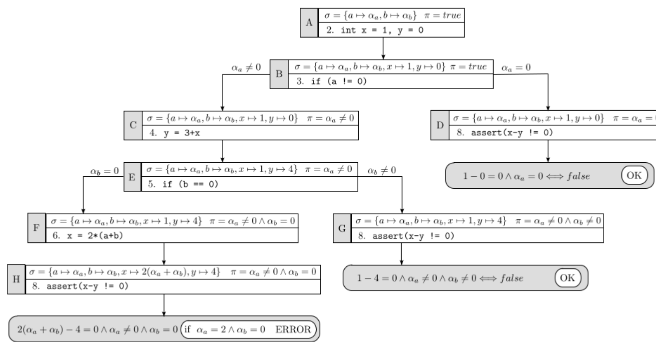
Simboličko izvršavanje je bila ključna tehnika koju je koristila DARPA (*Defense Advanced Research Projects Agency*) u okvriu projekta *Cyber Grand Challenge* — dvogodišnje takmičenje u traženju automatskih sistema za otkrivanje ranjivosti sistema, kao i za eksploataciju i ispravljanje grešaka u realnom vremenu.

Primer 6.2.1 Na slici 6.3 je dat primer funkcije `foobar` koja nameće uslov $x - y \neq 0$ u liniji 9. Funkcija će biti dovedena u stanje greške ukoliko važi negacija uslova koji se nameće u liniji 9, odnosno $x - y = 0$.

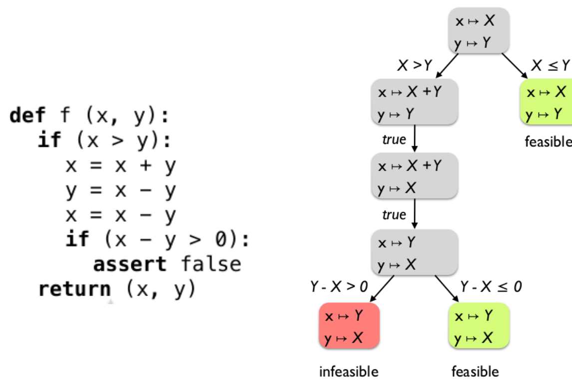
Na slici 6.4 prikazano je simboličko izvršavanje funkcije `foobar`.

```
1  void foobar(int a, int b)
2  {
3      int x = 1, y = 0;
4      if (a != 0) {
5          y = 3 + x;
6          if (b == 0)
7          x = 2 * (a + b);
8      }
9      assert(x - y != 0);
10 }
```

Slika 6.3: Funkcija foobar



Slika 6.4: Simboličko izvršavanje funkcije `foobar`.



Slika 6.5: Primer nedostupne/nedostižne putanje

Primer 6.2.2 Primer nedostupne/nedostižne putanje dat je na slici 6.9.

6.3 Izazovi simboličkog izvršavanja

Teorijski, simboličkim izvršavanjem možemo generisati sve moguće putanje kroz koje program može da prođe za vreme konkretnog izvršvanja programa na nekim konkretnim ulazima (pod pretpostavkom da se svako konkretno izvršavanje zaustavlja). Međutim, iako modelovanje svih mogućih izvršavanja daje mogućnost sprovođenja veoma interesantne analize koda, to je najčešće neizvodivo u praksi, posebno na softveru za realnu upotrebu. Postoje razni problemi koji se javljaju u ovom konktestu, a koji ne mogu da se reše praćenjem čisto simboličkog pristupa. To su problemi sa

- memorijom,
- okruženjem,
- eksplozijom broja putanja i prostora stanja,
- rešavanjem ograničenja,
- nedostupnošću izvornog koda.

Problemi sa memorijom su problemi u kojima je potrebno odrediti kako okruženje za simboličko izvršavanje razrešava rad sa pokazivačima, nizovima i kompleksnim objektima. Kôd koji radi sa pokazivačima i strukturama podataka može da stvori potrebu ne samo za simboličkim podacima veći i za adresama koje se opisuju simboličkim izrazima.

Problemi sa okruženjem se javljaju u interagovanju sistema za simboličko izvršavanje sa pozivima bibliotečkih funkcija i sa sistemskim pozivima koje ne može simbolički da isprati. Takođe, bitno je i pitanje na koji način simboličko izvršavanje modeluje sadržaj datoteka i sadržaj okoline koja utiče na izvršavanje programa.

Eksplozija broja putanja i prostora stanja se dešava da usled postojanja grananja i petlji u okviru programa pa okruženje za simboličko izvršavanje treba da ispita ogroman broj stanja. Grananje i petlje eksponencijalno uvećavaju broj putanja i stanja i postaje vrlo nepraktično ispitati i pratiti sva moguća stanja programa, odnosno malo je verovatno da simboličko izvršavanje može da pretraži sva moguća stanja u nekom razumnom vremenu, slika 6.6.

```
1 int a, b, c; // simbolicke vrednosti
2
3 if (a) ... else ... ;
4 if (b) ... else ... ;
5 if (c) ... else ... ;
```

Slika 6.6: Eksponencijalni broj putanja: tri grananja daje 8 mogućih putanja

Rešavanje ograničenja ima važnu ulogu u simboličkom izračunavanju jer se njime pokazuje da određeni uslovi važe ili pronalaze primeri za koje uslovi ne važe. Ipak i ovde postoji granica koja se ogleda u broju promenljivih i uslova koje rešavač može da reši, domen sa kojim radi (nelinearna aritmetika ume drastično da oteža rešavanje) i da li je poziv rešavača skup u odnosu na željene performanse.

Postoje situacije u kojima **izvorni kôd nije dostupan** te se simboličko izvršavanje vrši nad binarnim kodom (odnosno asemblerskim instrukcijama). Posledice toga su povećavanje

vremena izvršavanja i manjak interpretabilnosti jer je kôd visokog nivoa nedostupan.

Alat daje **lažno upozorenje** (eng. *false alarm*) ukoliko za neku naredbu u kodu prijavi da može da izazove grešku iako ne postoji konkretno izvršavanje koje bi tu grešku izazvalo.

Alat **propušta greške** ukoliko za neku naredbu u kodu ne prijavi da može da izazove grešku iako postoji konkretno izvršavanje koje bi tu grešku izazvalo.

Saglasnost (eng. *soundness*) Analiza je saglasna ako za svaki kôd za koji tvrdi da je ispravan on stvarno i jeste ispravan. To znači da nema neispravnih programa za koje analiza tvrdi da su ispravni, a ako se posmatra vezano za greške — nema propuštenih grešaka.

Kompletnost (eng. *completeness*) Analiza je kompletna ako za svaki ispravan kôd, analiza će i dokazati njegovu ispravnost. To znači da nema ispravnih programa za koje neće biti dokazana ispravnost, a ako se posmatra vezano za greške — nema lažnih upozorenja.

U zavisnosti od konteksta u kojem se simboličko izvršavanje koristi, postoje različiti izbori i pretpostavke koje se prave da bi se adresirala prethodna pitanja. Iako ovi izbori utiču na saglasnost i kompletnost, u mnogim scenarijima je i delimično istraživanje prostora mogućih izvršavanja dovoljno za postizanje željenog cilja (u okviru zadatog budžeta).

6.4 Principi dizajna i konkoličko izvršavanje

Postoje tri ključna aspekta simboličkog izvršavanja koja je potrebno razmatrati prilikom dizajniranja alata. To su napredak, ponavljanje posla i ponovno korišćenje rezultata.

- **Napredak**: izvršavanje bi trebalo da može da napreduje proizvoljno dugo vreme bez prelaženja zadatih granica resursa. Upotreba memorije je obično kritična, zbog mogućeg velikog broja različitih putanja i stanja.
- **Ponavljanje posla**: prilikom izvršavanja posao ne bi trebao da se ponavlja, odnosno treba izbegavati započinjanje programa više puta od samog početka sa ciljem analize različitih putanja koje mogu imati isti prefiks.
- **Ponovno korišćenje rezultata**: rezltati analize iz prethodnih pokretanja simboličkog izvršavanja treba da se ponovo iskoriste svuda gde je to moguće. Konkretno, skupi pozivi SMT rešavačima za ograničenja putanja koje su već ranije rešavana treba da se izbegavaju.

U skladu sa prioritetima dizajna, izdvajaju se *online* i *offline* izvršavanje, a postoji i hibridno izvršavanje (kombinacija prethodna dva).

Online izvršavanje podrazumeva izvršavanje više putanja simultano. Da bi se ovo ostvarilo kloniraju se stanja na svakom mestu gde može da dođe do grananja (npr. alat KLEE). Na ovaj način nikada se ne izvršavaju instrukcije koje su jednom izvršene, odnosno izbegava se ponavljanje posla. Međutim, veliki broj aktivnih stanja je potrebno održavati u memoriji i veoma lako se memorija prepuni i potrebno je na neki način to regulisati kako ne bi bio ugrožen napredak.

S druge strane, posao može da se u velikoj meri ponavlja ako svako izvršavanje započinje od početka programa. Kod *offline* izvršavanja, obično se program najpre izvršava konkretno, prate se instrukcije koje su konkretnim izvršavanjem izvršene i onda se one izvrše i simbolički. Upotreba memorije je značajno manja, a smanjuje se broj poziva solverima. Mešanje konkretnog i simboličkog izvršavanja naziva se **konkoličko** (**konk**retno+simb**oličko**) izvršavanje.

Najbitniji predstavnik konkoličkog izvršavanja je dinamičko simboličko izvršavanje. Postoje i druge vrste konkoličkog izvršavanja, kao što je, na primer, selektivno simboličko izvršavanje.

6.4.1 Dinamičko simboličko izvršavanje

Dinamičko simboličko izvršavanje je simboličko izvršavanje vođeno konkretnim vrednostima. Kao dodatak simboličkom skladištu i uslovima putanje, čuva se i **konkretno skladište**. Nakon što se izabere proizvoljni ulaz sa kojim se počinje izvršavanje, program se izvršava i konkretno i simbolički tako što se simultano ažuriraju oba skladišta i uslovi putanje. Kada god konkretno izvršavanje krene nekom konkretnom granom, simboličko izvršavanje se usmerava prema istoj grani i uslovi putanje se ažuriraju u skladu sa uslovom koji je zadovoljen da bi se tom granom krenulo. Ukratko, **simboličko izvršavanje je vođeno specifičnim konkretnim izvršavanjem**. Kao posledica, simbolčko izvršavanje ne mora da poziva rešavač ograničenja da odluči da li je neka grana dostupna, odnosno da li je neka formula koja odgovara uslovima putanje zadovoljiva: to se direktno proverava konkretnim izvršavanjem.

Da bi se istražile različite putanje, uslovi putanje koji su zadati sa jednim ili više grananja se mogu negirati, i tako dobijena formula se može dati SMT rešavaču da pronađe nove ulaze koji će onda ići tom novom putanjom. Ova strategija se može ponavljati onoliko puta koliko je to potrebno da se dostigne odgovarajuća pokrivenost.

Tehnike pretrage — izbor naredne putanje

Iako dinamičko simboličko izvršavanje koristi konkretne ulaze da navodi simboličko izvršavanje nekom konkretnom putanjom, i dalje postoji problem izbora uslova koji će biti

negiran, odnosno nove putanje koja će svaki naredni put biti istražena. Treba imati u vidu da svako konkretno izvršavanje može da doda nova grananja koja treba da budu posećena. Kako skup grana koje nisu istražene može biti veoma velik, veoma je važno usvojiti efikasnu heuristiku za pretragu.

Kako se prostor svih mogućih stanja samo parcijalno istraži, početni ulaz igra suštinsku ulogu u učinku celokupnog pristupa. Značaj prvog ulaza je sličan onome što se dešava u okviru tehnika rasplinutog testiranja crne kutije (eng. *black-box fuzzing*) i iz tog razloga se alati koji koriste dinamičko simboličko izvršavanje (kao, na primer, alat SAGE), često nazivaju i alatima za rasplinuto testiranje bele kutije (eng. *white-box fuzzing*).

```
void foo(int x, int y) {
   int a = bar(x);
   if (y < 0) ERROR;
}
```
(a)

```
void qux(int x) {
   int a = bar(x);
   if (a > 0) ERROR;
}
```
(b)

```
void baz(int x) {
   abs(&x);
   if (x < 0) ERROR;
}
```
(c)

Figure 4: Concolic execution: (a) testing of function `foo` even when `bar` cannot be symbolically tracked by an engine, (b) example of false negative, and (c) example of a path divergence, where `abs` drops the sign of the integer at `&x`.

Slika 6.7: Kako pratiti funkcijske pozive za koje kôd nije dostupan?

Razmotrimo funkcije `foo`, `qux` i `baz` na slici 6.7. Pretpostavimo da se funkcije `bar` i *abs* ne mogu isprati konkoličkim izvršavanjem. Na primer, to su funkcije koje su obezbeđene kroz spoljnu komponentu za koju nije dostupan kôd ili su napisane u drugom programskom jeziku. Ako pretpostvimo da su vrednosti `x = 1` i `y = 2` slučajno izabrane kao početni ulazni parameteri za konkoličko izvršavanje funkcije *foo*. Konkoličko izvršavanje će ispratiti izvršavanje funkcije `bar` koja će vratiti neku vrednost i preskočiće granu koja bi dovela do greške. U isto vreme, simboličko izvršavanje prati uslove putanje $\alpha_y \geq 0$ u okviru funkcije `foo`. Uslovi grananja u funkciji `bar` nisu poznati konkoličkom izvršavanju. Da bi se istražila alternativna putanja, negira se ograničenje putanje u grananju u okviru funkcije `foo`. To gneriše, na primer, ulaze $x = 1$ i $y = -4$ koji navode konkretno izvršavanje alternativnom putanjom. Sa ovakvim pristupom, obe putanje u `foo` mogu da budu istražene iako funkcija `bar` nije simbolički ispraćena.

Funkcja `qux`, za razliku od funkcije `foo`, uzima jedan ulazni parametar ali proverava rezultat funkcije `bar` u uslovu grananja Iako simbolicko izvršavanje prati uslove putanja u granama u okviru funkcije `qux`, ne postoji garancija da će ulaz moći da navede izvršavanje alternativnim putem: relacija između `a` i `x` je nepoznata konkoličkom izvršavanju jer se

izvršavanje funkcije `bar` ne prati simbolički. U ovom slučaju, izvršavanje koda može da bude pokrenuto puno puta sa različitim random generisanim ulazima, ali na kraju nemamo garanciju da ćemo uspeti tj. veoma je moguće da nećemo uspeti da istražimo interesantne putanje kroz funkciju `foo`.

Funkcija `baz` poziva eksternu funkciju `abs` koja računa apsolutno vrednost broja. Biranjem $x = 1$ za ulazni parametar konkretnog izvršavanja, konkretno izvršavanje ne uzrokuje grešku, ali simboličko izvršavanje beleži uslov putanje $\alpha_x \geq 0$ na osnovu grananja u funkciji `baz` i pokušava da generiše novi izlaz njegovim negiranjem, npr $x = -1$. Međutim, novi ulaz ne prolazi kroz alternativnu putanju zbog uticaja funkcije `abs`. U ovom slučaju, nakon generisanja novog ulaza izvršavanje detektuje **divergenciju putanje**, tj. konkretno izvršavanje koje ne prati predviđeni put. U ovom primeru, ne postoji ulaz koji može da prođe kroz putanju koja uzrokuje grešku, ali konkoličko izvršavanje nije u stanju da detektuje ovu osobinu.

Na osnovu prethodnog primera, možemo da zaključimo da simboličko izvršavanje može da ima lažne negativne rezultate. Lažno negativni rezultati (tj. promašene putanje) kao i divergencija putanja su značajne loše osobine dinamičkog simboličkog izvršavanja. Dinamičko simboličko izvršavanje žrtvuje saglasnost pristupa zarad performansi i lakše implementacije: propuštene greške su moguće jer neka izvršavanja programa, a samim tim i moguća pogrešna ponašanja, mogu da budu propuštena. Analiza ostaje kompletna.

6.4.2 Selektivno simboličko izvršavanje

Drugačiji pristup mešanju simboličkog i konkretnog izvršavanja Konkretno i simboličko izvršavanje se može kombinovati na različite načine. Jedan pristup koji je drugačiji od dinamičkog simboličkog izvršavanja je selektivno simboličko izvršavanje. Osnovna ideja je da neko može da želi da istraži samo neke delove koda u potpunosti, tj. simbolički, a da pritom nije zainteresovan za neke druge delove koda. Selektivno simboličko izvršavanje preplиće konkretno i simboličko izvršavanje održavajući pritom analizu smislenom. Na primer, pretpostavimo da funkcija A poziva funkciju B i da se način izvršavanja prilikom poziva menja.

Od konkretnog prema simboličkom i nazad — *konkretno izvršavanje A, simboličko B.* Argumenti funkcije B postaju simbolički i B se istražuje u potpunosti simbolički. Takođe, B se izvršava i konkretno i konkretan rezultat se vraća u A tako da kada se završi istraživanje funkcije B, funkcija A nastavlja normalno dalje sa radom.

Od simboličkog prema konkretnom i nazad — *simboličko izvršavanje A, konkretno B.* Argumenti funkcije B se konkretizuju i B se izvršava konkretno, a zatim izvršavanje u okviru A se nastavlja simbolički.

Ovakav pristup može da utiče i na saglasnost i na kompletnost analize.

Konkretizacija može da uzrokuje da se u simboličkom izvršavanju preskoče grane koje su dostižne nakon što se vratimo u A, što može da dovede do lažno negativnih rezultata. Da bi to izbegli, skupljena ograničenja se markiraju kao soft ograničenja: kada god neka grana, nakon vraćanja u A bude obeležena kao nedostižna zbog soft ulsova, izvršavanje radi bektreking i bira na drugi način argumente za B. Za vođenje nove konkretizacije argumenata za B, u tom slučaju se koriste uslovi grana koji su skupljeni za B i biraju se one konkretne vrednosti koje omogućavaju drugačije konkretno izvršavanje kroz B.

Da bi bili sigurni da simboličko izvršavanje preskače sve putanje koje nisu dostižne u skladu sa izvršenom konkretizacijom, tj da ne bi imali lažno pozitivne rezultate, potrebno je skupiti sva ograničenja putanje koja je izvršena u skladu sa odgovarajućom konkretizacijom, sve bočne efekte koje je B napravila kao i povratnu vrednost koju B proizvodi.

6.5 Strategije obilaska puteva

Kako obilzak svih mogućih putanja programa može da bude preskup ili vremenski/memorijski nedostižan, u okviru raznih aktivnosti vezanih za testiranje i debagovanje, pretraga se navodi tako što se ispituju najpre putanje za koje deluje da najviše obećavaju.

Postoje razne strategije za izbor naredne putanje. Sve strategije se zasnivaju na heuristikama koje treba da pomognu u ostvarivanju nekih konkretnih ciljeva. Pronalaženje univerzalno optimalne strategije je otvoren istraživački problem.

6.5.1 Naivni pristupi: DFS i BFS

Najčešće strategije i najjednostavnije strategije zasnovane na strukturi koda su pretraga u dubinu (eng. *depth-first search, skraćeno DFS*) i pretraga u širinu (eng. *breadth-first search, skraćeno BFS*).

Pretraga u dubinu prati putanju dok je god to moguće pre nego što se uradi *backtracking* na najdublju neistraženu granu. DFS se primenjuje kada je prioritet smanjena upotreba memorije, ali često biva uhvaćena putanjama koje sadrže petlje ili rekurzivne pozive.

Pretraga u širinu, prati sve putanje paralelno. Uprkos većem zauzeću memorije i velikom vremenu koje je potrebno da se neka putanja završi do kraja, neki alati koriste BFS jer on omogućava da se brzo istraže skroz različite putanje i da se rano detektuju interesantna ponašanja sistema. S druge strane, ako krajnji cilj zahteva da se kompletno istraži jedna ili više putanja, BFS može da zahteva jako puno vremena.

6.5.2 Random strategija

Random strategija podrazumeva upotrebu nasumičnog izbora u situacijama kada se vrši nekakav izbor. Postoje različite moguće ideje:

- Ideja 1: Izaberi sledeću putanju za istraživanje random metodom
- Ideja 2: Random metodom restartuj pretragu ukoliko se ništa novo ne dešava već neko vreme
- Ideja 3: Kada imamo da istražimo dve jednako prioritetne putanje, izaberi sledeću random
- ...

Ukoliko bi bio korišćen pravi random, postojao bi problem sa reprodukovanjem. Zbog toga se koristi pesudo-random i čuva se *seed*.

6.5.3 Izvršavanje vođeno pokrivenošću koda

Izvršavanje vođeno pokrivenošću koda ima za cilj da maksimizuje pokrivenost koda. Algoritam — izaberi putanju koja će najverovatnije da izvrši neku novu instrukciju:

- Pokušaj da posetiš instrukcije koje ranije nisu bile izvršavane.
- Ukoliko takve putanje nema, izaberi onu putanju u kojoj su instrukcije izvršavane najmanji broj puta.

Dobra osobina: greške su često u delovima programa koji se retko izvršavaju, a ova strategija pokušava da dopre svuda.

U okviru izvršavanja vođenog pokrivenošću koda daju se težine stanjima ili putanjama i na osnovu ovih težina se određuje dalja pretraga.

Primer 6.5.1 KLEE dodeljuje težine putanjama koje se zasnivaju na njihovoj dužini i na arnosti njihovog grananja: favorizuje putanje koje su manji broj puta bile istraživane čime sprečava zaglavljivanje u petljama i u drugim uzrocima eksplozije broja putanja. Za svako stanje, izračunava se težina koja se koristi prilikom izbora stanja sa kojim se izvršavanje nastavlja. Težina se izračunava razmatranjem koliko daleko se nalazi instrukcija koja nije pokrivena, da li je novi kôd otkriven iz tog stanja relativno skoro, kao i iz kojih stanja se došlo do datog stanja.

Pretraga vođena pod-putanjama pokušava da istražuje putanje kojima je manji broj puta prolaženo tako što bira pod-putanje grafa kontrole toka koje su obilažene manji broj puta. Ovo se ostvaruje čuvanjem informacija o istraženim pod-putanjama, gde se pod-putanja definiše kao uzastopna sekvenca stanja dužine n. Veličina n igra ključnu ulogu za ostvarivanje dobre pokrivenosti koda ovom heuristikom, ali ne postoji jedna specifična vrednost n koja je univerzalno optimalna.

Primer 6.5.2 Generacijska pretraga u okviru alata *SAGE* predstavlja hibrid DFSa i izvršavanja vođenog pokrivenošću koda:

- Generacija 0: Izvrši random putanju do kraja
- Generacija 1: Uzmi sve putanje iz generacije 0, negiraj jedan uslov tako da vodi do novog prefiksa putanje, nadji rešenje te putanje i onda je izvrši
- ...
- Generacija N: slično, samo što se grananje vrši u odnosu na putanju generacije N-1 (za biranje putanje se koristi heuristika pokrivenosti koda)

6.5.4 Izvršavanje vođeno najkraćim rastojanjem

Izvršavanje vođeno najkraćim rastojanjem (eng. *shortest-distance symbolic execution*) nema za cilj povećanje pokrivenosti koda već pronalaženje ulaznih parametara koji će uzrokovati izvršavanje neke izabrane tačke u programu. Ova heuristka se zasniva, slično kao i izvršavanja vođena pokrivenošću koda, na metrikama za evaluaciju najkraćeg rastojanja do ciljne tačke. Ovo rastojanje se računa kao dužina najkraće putanje u grafu intereproceduralne kontrole toka i putanjama koje imaju najkraću distancu se daje prioritet.

6.5.5 Kombinovana strategija

Kombinovana strategija podrazumeva istovremenu pretragu sa različitim algoritmima. Ova strategija zavisi od uslova koji su potrebni da se pronađe greška u kodu, ponaša se kao najbolji od korišćenih algoritama, sa konstantnim faktorom izgubljenog vremena i memorije sa svim ostalim algoritmima. Mogu se koristiti različiti algoritmi da bi se došlo do različitih delova programa.

6.5.6 Izvršavanje unazad

Simboličko izvršavanje unazad (eng. *Symbolic Backward Execution, skraćeno SBE*) je varijanta simboličkog izvršavanja u kojem izvršavanje počinje od ciljne tačke prema tački ulaza u program, tj analiza se izvršava u obrnutom smeru. Osnovni cilj ovog pristupa je da se napravi test primer koji uzrokuje izvršavanje specifične linije koda (obično nekog *assert*-a ili neke *throw* naredbe. Ovo je takođe može da bude korisno za debagovanje ili regresiono testiranje.

Kako izvršavanje počinje od ciljne linije koda, ograničenja putanje se skupljaju po grananjima unazad. Više putanja se istražuje u jednom trenutku i kao kod običnog simboličkog izvršavanja, putanje se povremeno proveravaju da li su dostižne. Ako putanja nije dostižna, ona se odbacuje i radi se *backtracking*.

Druga vrsta izvršavanja unazad je izvršavanje unazad po lancima poziva (eng. *Call-chain backward symbolic execution, skraćeno CCBSE*). Tehnika započinje utvrđivanjem validne putanje u okviru funkcije gde je ciljna linija locirana. Kada

se putanja pronađe, pomeramo se na funkciju pozivaoca ove funkcije i pokušavamo da rekonstruišemo validnu putanju od njenog ulaza do poziva funkcije u kojoj je ciljna linija koda. Proces se rekurzivno nastavlja dok ne dođemo do *main* funkcije. Osnovna razlika između SBE i CCSBE je što se u okviru svake funkcije za CCSBE izvršava obično simboličko izvršavanje dok se za SBE izvršava unazad.

Da bi izvršavanje unazad moglo da se primeni, potrebno je da postoji na raspolaganju inter-proceduralni graf kontrole toka (eng. *control-flow graph*) koji obezbeđuje tok kontrole za ceo program i omogućava da se odrede mesta poziva svih funkcija koje učestvuju u istraživanju. Nažalost, konstruisanje takvog grafa često je vrlo složen posao u praksi. Dodatno, svaka funkcija može biti pozvana sa više mesta u kodu što dodatno otežava (usporava) pretragu.

6.6 Modelovanje memorije

Primeri koje smo razmatrali su koristili jednostavnu memoriju koja sve podatke čuva kao skalarne vrednosti, tj. bez redirekcije. Važan aspekt simboličkog izvršavanja je modelovanje memorije tako da podrži programe sa pokazivačima i nizovima. To zahteva proširivanje memorije tako da preslikava ne samo promenljive u njihove simboličke vrednosti, već i memorijske adrese u simbličke izraze ili konkretne vrednosti.

Simboličko skladište σ može se posmatrati kao preslikavanje koje adresama u memoriji (umesto samim promenljivama) dodeljuje konkretne ili simboličke vrednosti. Na ovaj način i dalje možemo da podržavamo obične promenljive koristeći njihove adrese umesto imena u tom preslikavanju. Na primer, umesto da se x preslikava u izraz e (u zapisu $x \rightarrow e$) možemo da to razmatramo zapravo kao $\&x \rightarrow e$ gde je $\&x$ konkretna adresa promenljive x. Takođe, ako je v niz i c konstanta, onda za $v[c] \rightarrow e$ zapravo razmatramo $\&v + c' \rightarrow e$ (c' je odgovarajući pomeraj za konstantu c)

Za konkretne vrednosti to ne predstavlja problem (tj. kada je indeks konstanta) ali izazov je obraditi situacija kada je indeks simbolički izraz. To se naziva **problem simboličkih adresa** Izbor modelovanja memorije je važan izbor za dizajn simboličkog izvršavanja jer utiče direktno i na ostvarenu

pokrivenost koda istraživanjem putanja i na skalabilnost rešavača. Postoje različiti pristupi rešavanju ovog problema.

6.6.1 Potpuno simbolička memorija

Kao najopštiji pristup, memorija može da se tretira potpuno simbolički (eng. *fully symbolic memory*):

1. Pravljenje novih stanja
2. Pravljenje *if-then-else* formula
3. Upotreba SMT rešavača i teorije nizova

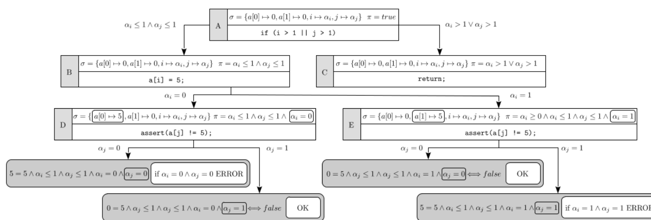
Pravljenje novih stanja

Ako operacija čita ili piše na neku simboličku adresu, prave se nova stanja (eng. *state forking*) razmatranjem svih mogućih stanja koje mogu da rezultuju kao posledica te operacije. Uslovi putanje se dopunjavaju za svako novokreirano stanje.

Primer 6.6.1

```
1  void foobar ( unsigned i, unsigned j ) {
2    int a[2] = { 0, 0 };
3    if (i>1 || j>1) return;
4    a[i] = 5;
5    assert (a[j] != 5);
6  }
```

Stablo izvršavanja za primer iz prethodnog listinga dato je na slici 6.8.



Slika 6.8: Pravljenje novih stanja

if-then-else formule

Alternativni pristup: umesto kreiranja novih stanja, prebaciti posao pretrage na rešavač. Pristup se sastoji u enkodiranju neodređenosti mogućih vrednosti simboličkog pokazivača

u izraze koji se čuvaju u okviru simboličkog skladišta i ograničenja putanja, bez pravljenja novih stanja. Osnovna ideja je da se iskoristi mogućnost rešavača da rezonuju o formulama koje sadrže *if-then-else* izraze, tj. u formi *ite(cond, trueExp, falseExp)*

Pristup radi drugačije za operacije čitanja i pisanja. Neka je α simbolička adresa koja može da ima konkretne vrednosti a_1, a_2 ...

- čitanje sa lokacije α podrazumeva izraz

$$ite(\alpha = a_1, \sigma(a_1), ite(\alpha = a_2, \sigma(a_2), \ldots))$$

- pisanje izraza e na lokaciju α menja simboličko skladište σ tako što svaku vrednost promenljivih na adresama a_1, a_2 ... postavlja na sledeći način:

$$\sigma(a_i) = ite(\alpha = a_i, e, \sigma(a_i))$$

Primer 6.6.2 Na primer, neka je α simbolička adresa koja može imati konkretne vrednosti a_1, a_2, a_3 i a_4

- čitanje sa lokacije α podrazumeva izraz

$$ite(\alpha = a_1, \sigma(a_1), ite(\alpha = a_2, \sigma(a_2),$$

$$ite(\alpha = a_3, \sigma(a_3), \sigma(a_4))))$$

- pisanje izraza e na lokaciju α menja simboličko skladište σ na sledeći način:

$$\sigma(a_1) = ite(\alpha = a_1, e, \sigma(a_1))$$

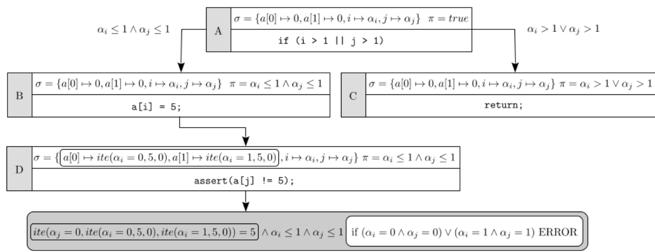
$$\sigma(a_2) = ite(\alpha = a_2, e, \sigma(a_2))$$

$$\sigma(a_3) = ite(\alpha = a_3, e, \sigma(a_3))$$

$$\sigma(a_4) = ite(\alpha = a_4, e, \sigma(a_4))$$

Teorija nizova

Teorija nizova (eng. *theory of arrays, skraćeno ARR*) u okviru SMT rešavača omogućava da se operacije nad nizovima jednostavno izraze tj i oni su *građani prvog reda* u okviru formula. Teorija nizova uvodi ternarni funkcijski simbol *store*



(nekada se naziva i *write*) i binarni funkcijski simbol *select* (nekada se naziva i *read*).

Funkcija *store* vrši izmenu vrednosti na odgovarajućem mestu u nizu i ima tri argumenta:

(i) niz,
(ii) poziciju (indeks) u nizu na koju se smešta vrednost,
(iii) vrednost koja se smešta u niz.

Za zadati niz a, celobrojnu vrednost i i vrednost v tipa nad kojim je niz definisan, term $store(a, i, v)$ označava niz koji je identičan sa nizom a, osim što je vrednost na poziciji i jednaka v. Funkcija *select* vrši čitanje vrednosti sa odgovarajućeg mesta u nizu i ima dva argumenta:

(i) niz,
(ii) poziciju (indeks) u nizu sa koje se čita vrednost.

Za zadati niz a i celobrojnu vrednost i, term $select(a, i)$ označava vrednost na poziciji i niza a.

Teorija nizova je teorija koja ima sledeće dve aksiome:

$$\forall a\ \forall i\ \forall v\ (select(store(a, i, v), i) = v) \tag{A1}$$

$$\forall a\ \forall i\ \forall j\ \forall v\ (i \neq j \Rightarrow select(store(a, i, v), j) = select(a, j)) \tag{A2}$$

Univerzalno kvatnifikovani fragment ove teorije je odlučiv i problem zadovoljivosti u ovom fragmentu teorije je NP-kompletan.

Pored prethodne dve aksiome, ponekad se dodaje i aksioma **proširivanja** (eng. *extensionality*):

$$\forall a\ \forall b\ ((\forall i\ (select(a, i) = select(b, i)) \Rightarrow a = b) \tag{A3}$$

Primer 6.6.3 Naredna formula je primer zadovoljive formule teorije nizova:

$$a' = store(a, 5, select(a, 3)) \land select(a, 3) = select(a', 5)$$

Intuitivno: niz a' nastaje kada se na petom mestu niza a upiše vrednost koja se nalazi na trećem mestu niza a (npr: `a[5] = a[3];`), i vrednost na trećem mestu niza a i vrednost na petom mestu niza a' su jednake.

Primer 6.6.4 Naredna formula je primer nezadovoljive formule teorije nizova:

$$a = b \ \land \ i = j \land \ store(a, i, x) \neq b$$
$$select(b, i) = y \land \ select(store(b, i, x), j) = y$$

Da bi se dokazala nezadovoljivost ove formule, potrebno je koristiti i aksiomu (A3).

Intuitivno, ova formula govori da su a i b nizovi sa svim jednakim vrednostima ($a = b$), i da su i i j jednake pozicije u tim nizovima ($i = j$). Ako na poziciji i niza a upišemo vrednost x, ta vrednost x menja niz a ($store(a, i, x) \neq b$). Takođe, na poziciji i niza b se nalazi vrednost y. Iz ovoga možemo da zaključimo da je $x \neq y$. Zbog toga ne može da važi da kada se izabere vrednost sa pozicije na koju je upisana vrednost x da to bude vrednost y ($select(store(b, i, x), j) = y$).

Primer 6.6.5 Primer modelovanja

```
1 a[i] = 5;
2 b[i] = a[i]+1;
3 c[i] = 100 / (a[i] - b[i]);
```

$$a \longrightarrow store(a, i, 5)$$
$$b \longrightarrow store(b, i, select(store(a, i, 5), i) + 1)$$

ili, sa uvođenjem novih promenljivih a' i b'

$$a \longrightarrow a' \ \land \ a' = store(a, i, 5)$$
$$b \longrightarrow b' \ \land \ b' = store(b, i, select(a', i) + 1)$$

Provera ispravnosti: `a[i] - b[i] != 0` odnosno `a[i] != b[i]`?

$$select(a', i) \neq select(b', i)$$

odnosno
$select(store(a, i, 5), i) \neq select(store(b, i, select(store(a, i, 5), i) + 1), i)$
$5 \neq select(store(b, i, 5 + 1), i)$
$5 \neq 6$

Ograničenja simboličke memorije

Zahvaljujući svojoj opštosti, puna simbolička memorija podržava najpreciznije opis stanja memorije i ponašanja programa i uzima u obzir sve moguće manipulacije sa memorijom. U mnogim praktičnim scenarijima upotrebe, skup mogućih adresa sa kojima memorija operiše je mali, što dozvoljava preciznu analizu korišćenjem razumne količine resursa. Međutim, u opštem slučaju, simbolička adresa može da referencira bilo koju ćeliju memorije, što vodi do velike ekplozije broja stanja koju nije moguće u razumnom vremenu ispratiti. Iz tih razloga, postoji veliki broj tehnika koje su dizajnirane sa ciljem da se poboljša skalabilnost upotrebe teorije nizova i simboličke memorije.

6.6.2 Kompromisi

Kako puna simbolička memorija ne skalira dobro, potrebno je napraviti neki kompromis (eng. *trade-off*). Pravljenje dobrog kompromisa je i dalje otvorenproblem.

Konkretna memorija — Konkretizacija simboličkih adresa

Ukoliko je kombinatorna kompleksnost analize prevelika jer vrednosti pokazivača ne mogu da se ograniče na dovoljno male opsege, često se koristi strategija **konkretizacije simboličke pokazivačke vrednosti** u neku konkretnu vrednost. U ovom slučaju imamo gubitak saglasnosti zarad dobijanja na performansama. Ovo može da smanji broj stanja i kompleksnost formula koje se daju rešavaču i na taj način da poboljša vreme izvršavanja, ali može da uzrokuje i da budu

preskočene neke putanje koje zavise od nekih specifičnih vrednosti pokazivača.

Konkretizacija se prirodno javlja u *offline* simboličkom izvršavanju Konkretizacija pokazivačke vrednosti tipa T* u NULL ili u adresu novo-alociranog objekta veličine sizeof(T). Ovaj izbor može da ide random (DART) ili da imamo različita stanja za različite izbore (CUTE - prvo proba sa NULL, pa zatim sa konkretnom adresom). Ako je T struktura, ista konkretizacija se rekurzivno sprovodi na sva polja na koja pokazuje objekat.

Parcijalno modelovanje memorije

Da bi se prevazišli problemi skaliranja pune simboličke memorije, a da bi se preskočio gubitak informacija koji nastaje konkretizacijom, jedan pravac je parcijalno modelovanje memorije. Ključna ideja je da adrese na koje se piše uvek budu konkretizovane, a da adrese sa kojih se čita se modeluju simbolički ako je neprekidan interval svih mogućih vrednosti koje se mogu pretpostaviti dovoljno mali

Kompromis: korišćenje izražajnih formula u odnosu na konkretizaciju, jer se enkodiraju višestruke vrednosti pokazivača po stanju, ali se ne pokušava enkodiranje svih simboličkih vrednosti kao što je to u punom simboličkom modelovanju memorije. Osnovni pristup vezivanja skupa mogućih vrednosti koje mogu da se pretpostave za adresu je pokušavanje upotrebe različitih konkretnih vrednosti i proveravanje da li one zadovoljavaju tekuće uslove putanje. Na taj način se mogu isključiti veliki delovi adresnog prostora pri svakom pokušaju, sve dok se ne nađe odgovarajući dovoljno uzan prostor. Ovaj algoritam zvuči logično i jednostavno, ali zapravo ima veliki broj dodatnih pretpostavki i posledica koje treba da se uzmu u razmatranje.

6.6.3 Lenja inicijalizacija

Simboličko izvršavanje u prisustvu struktura podataka kao što su liste i drveta je značajno teže. Koristi se okvir verifikacije softvera koji kombinuje simboličko izvršavanje i proveravanje modela. Uopštavanje simboličkog izvršavanja uvođenjem lenje inicijalizacije za efikasno rukovanje sa dinamički alociranim objektima. Kombinovanje lenje inicijalizacije sa korisnički

obezbeđenim preduslovima, tj. uslovima koji se pretpostavlja da su tačni pre izvršavanja metoda.

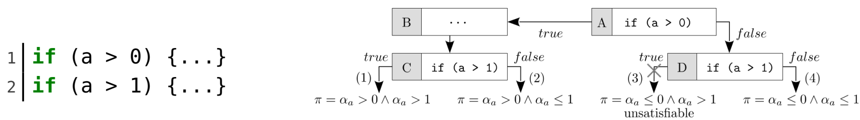
6.7 Eksplozija broja stanja i putanja

Jedna od osnovnih izazova simboličkog izvršavanje je eksplozija broja putanja i broja stanja: simboličkim izvršavanjem mogu se praviti nova stanja na svakom grananju programa i ukupan broj stanja vrlo lako postaje eksponencijalan po broju grananja. Praćenje velikog broja stanja i grana koje treba da budu istražene utiču i na vreme izvršavanja i na ukupnu potrošnju memorije.

6.7.1 Odsecanje nedostižnih putanja

Broj stanja se može redukovati pozivom SMT rešavača da detektuje nedostižna stanja (putanje). Kako su pozivi rešavača skupi, potrebno je napraviti kompromis: poziv rešavača kod svakog grananja (kako se ne bi formirale putanje koje su nedostižne) vs poziv rešavača povremeno (štednja poziva rešavaču jer su pozivi rešavača skupi).

Primer nedostižne putanje dat je na slici 6.9.



Slika 6.9: Za kod sa leve strane može se utvrditi da je jedna putanja nedostižna (slika desno).

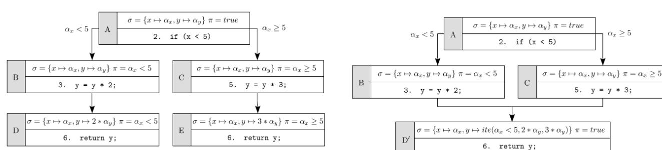
6.7.2 Spajanje stanja

Ako imamo dva stanja $(stmt, \sigma_1, \pi_1)$ i $(stmt, \sigma_2, \pi_2)$ spojeno stanje je

$$(stmt, \sigma', \pi_1 \lor \pi_2)$$

pri čemu σ' spaja stanja σ_1 i σ_2 sa ite izrazima. Spajanjem stanja pravimo kompromis: spajanje smanjuje broj putanja koje treba istražiti ali prebacuje i otežava posao rešavaču. Postoje razne heuristike za određivanje kada spajati stanja.

```
1 int foo(int x, int y) {
2  if(x < 5)
3    y = y * 2;
4  else
5    y = y * 3;
6  return y;
7 }
```



Slika 6.10: Stablo izvršavanja za kôd koji je prikazan sa leve strane: bez spajanja stanja (sredina) i sa spajanjem stanja korišćenjem *ite* formule (desno)

6.7.3 Aproksimacije petlji

Prilikom izvršavanja petlji, broj novih stanja raste u zavisnosti od ograničenja petlje. Ukoliko je petlja konačna, na primer za vrednosti pozitivne promenljive n koja je ograničena sa 10, broj novih stanja je 10: svako stanje odgovara jednoj mogućoj vrednosti za promenljivu n, odnosno prva putanja odgovara jednom prolasku kroz petlju, druga odgovara situaciji kada imamo dva prolaska kroz petlju i tako redom, do deset. Ukoliko n nije ograničeno, onda bi imali broj stanja koji odgovara maksimalnoj celobrojnoj vrednosti koja odgovara datom tipu promenljive n. Ukoliko je petlja beskonačna (npr. `while(1)`) i iz nje se izlazi na primer na osnovu nekog spoljašnjeg uslova (npr. kada učitani broj zadovoljava neki kriterijum) tada je broj mogućih putanja, odnosno narednih stanja beskonačan.

Dakle, svaka petlja, u najboljem slučaju uvodi veliki broj stanja/putanja, a u najgorem slučaju uvodi beskonačan broj stanja/putanja u razmatranje. Zbog toga se sprovode različite aproksimacije petlji.

Razmotavanje petlji

Najjednostavnija aproksimacija je razmotavanje petlje. Razmotavanjem petlje fiksira se broj prolazaka kroz petlju. Najčešće se za petlju razmatraju naredni slučajevi:

- Nijedan prolazak kroz petlju
- Jedan prolazak kroz petlju
- Ako je poznat maksimalni broj porlazaka kroz petlju, onda i maksimalni broj prolazaka kroz petlju

Na ovaj način gubi se saglasnost analize, tj neke greške zbog ovakve aproksimacije mogu promaći zato što ova aproksimacija smanjuje opseg koda koji se razmatra (eng. *under-apprximation*).

Primer 6.7.1 Razmotrimo naredni kod:

```
void f (unsigned int n) {
  i = 0;
  while (i < n) {
    ... // izracunavanje
    i = i + 1;
  }
}
```

Simboličko izvršavanje može da pravi nova stanja za sve moguće vrednosti promenljive `n`. Razmotavanju petlje dva puta odgovara naredni kod:

```
void f (unsigned int n) {
  i = 0;
  if (i < n) {
    ... // izracunavanje
    i = i + 1;
  }
  if (i < n) {
    ... // izracunavanje
    i = i + 1;
  }
}
```

Invarijante petlji

Obezbediti invarijantu petlje koja će dozvoliti simboličkom izvršavanju da preskoči analizu petlje. Ovakva analiza je polu-automatska (ako korisnik obezbeđuje invarijante) ili je u pitanju automatsko generisanje invarijante što ponovo dovodi do aproksimacije koda u smeru uopštavanja rezultata izvršavanja petlje (eng. *over-approximation*).

Primer 6.7.2 Invarijanta naredne petlje je $i \leq n$.

```
void f (unsigned int n) {
  i = 0;
  while (i < n) { // inv: i <= n
    i = i + 1;
  }
}
```

6.8 Rešavači

Ključna komponenta efikasnosti simboličkog izvršavanja su performanse rešavača. Pozivi rešavača su skupi. Ograničenja koja se javljaju u realnom softveru su teška za rešavanje i često uključuju nelinearna ograničenja i ograničenja teorije nizova. Važno je izbeći pozivanje rešavača uvek kada je to moguće.

Tehnike smanjivanja opterećenja rešavača:

- Simplifikovati izraz u hodu
- Inkrementalno rešavanje
- Čuvanje dobijenih rezultata i njihovo ponovno krišćenje.
- Zamena konkretnih vrednosti simboličkim i obratno u okviru kompleksnih uslova

Primer 6.8.1 Primer izbegavanja poziva rešavača korišćenjem dobijenih rezultata.
Na primer, simboličko izvršavanje može da održava preslikavanje iz formula u odgovarajuće vrednosti koje ih zadovoljavaju:

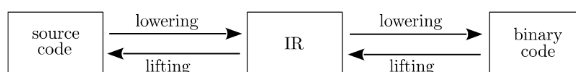
$$x + y < 10 \wedge x > 5 \longrightarrow \{x = 6, y = 3\}$$

- Ako nam je potrebno rešenje za slabiju formulu, npr za $x + y < 10$, možemo da iskoristimo prethodno sračunate vrednosti bez poziva rešavača.
- Ako je formula jača, npr. $x + y < 10 \wedge x > 5 \wedge y \geq 0$ možemo najpre da proverimo da li je sačuvano rešenje ok, ako jeste, iskoristimo ga, ako nije, tek onda ponovo pozivamo rešavač.

6.9 Binarni kôd

Na postojeće probleme i tehnike, kod simboličkog izvršavanja binarnog koda dodaje se:

- Podizanje do međureprezentacije



- Rekonstruisanje grafa kontrole toka
- Problem obfuskacije koda

Literatura

Pregled simboličkog izvršavanja **A Survey of Symbolic Execution Techniques**
autori: Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi
ACM Computing Surveys (CSUR), 51(3), pp.1-39. 2018.
https://dl.acm.org/doi/10.1145/3182657
Rad je slobodno dostupan na:
https://arxiv.org/abs/1610.00502
Dodatni materijali:
http://www.verifikacijasoftvera.matf.bg.ac.rs/

Semantika programskih jezika 7

7.1 Neformalna semantika 108
7.2 Osnovne vrste formalnih semantika 110
7.3 Operaciona semantika 114
7.4 Denotaciona semantika 118
7.5 Aksiomatska semantika 124

Semantika programskog jezika određuje značenje jezika, odnosno šta se dešava kada se program izvršava. Semantika pridružuje značanje samo ispravnim konstrukcijama na nekom programskom jeziku.

Semantika se može zadati *neformalno* i *formalno*. Neformalno opisivanje značenja jezika je najčešći vid zadavanja semantike programskog jezika. Formalno zadavanje semantike je obično značajno teže nego formalno zadavanje sintakse programskog jezika. Ali, slično kao što formalno zadavanje sintakse ima svoje prednosti (na primer, omogućava nam automatsko generisanje parsera), tako i formalno zadavanje semantike ima svoje značajne primene.

Primer 7.0.1 Razmotrimo sintaksu narednog jednostavnog jezika koji će nam služiti kao primer za objašnjavanje različitih tipova semantika:

```
n ::= 0 | 1 |
      n 0 | n 1
a ::= n | x |
      a1 + a2 | a1 * a2 | a1 - a2
b ::= true | false |
      a1 = a2 | a1 <= a2 |
      !b | b1 && b2
S ::= x := a |
      skip |
      S1 ; S2 |
      if b then S1 else S2 |
      while b do S
```

U okviru ovog jezika:

n označava numerale koji se formiraju od konstanti 0 i 1

a označava aritmetičke izraze, pri čemu je `x` oznaka za promenljivu a +, - i * su binarni operatori

b označava logičke izraze pri čemu su `true` i `false` logičke konstante, = i <= binarni operatori nad aritmetičkim izrazima, && binarni operator nad logičkim izrazima i `!` unarni operator nad logičkim izrazom

S označava naredbe programa, pri čemu su `skip`, `if`, `then`, `else`, `while` i `do` ključne reči, a `;` je separator.

7.1 Neformalna semantika

Uloga neformalne semantike je da programer može da razume kako se program izvršava pre njegovog pokretanja. Na primer, semantika naredbe `if(a<b) a++;` neformalno se opisuje sa „ukoliko je vrednost promenljive `a` manja od vrednosti promenljive `b`, onda uvećaj vrednost promenljive `a` za jedan". Programski jezici su kompleksni i zadavanje semantike programskog jezika je kompleksno.

Primeri zadavanja neformalne semantike nekad i sad dati su na slikama 7.1 i 7.2. Na slikama vidimo da se zadavanje neformalne semantike nije suštinski promenilo od početka razvoja programskih jezika šezdesetih godina prošlog veka.

A procedure statement serves to invoke (call for) the execution of a procedure body (cf. section 5.4. procedure declarations). Where the procedure body is a statement written in Algol the effect of this execution will be equivalent to the effect of performing the following operations on the program at the time of execution of the procedure statement.

- 4.7.3.1. Value assignment (call by value). All formal parameters quoted in the value part of the procedure declaration heading are assigned the values (cf. section 2.8. Values and types) of the corresponding actual parameters, these assignments being considered as being performed explicitly before entering the procedure body. The effect is as though an additional block embracing the procedure body were created in which these assignments were made to variables local to this fictitious block with types as given in the corresponding specifications (cf. section 5.4.5). As a consequence, variables called by value are to be considered as nonlocal to the body of the procedure, but local to the fictitious block (cf. section 5.4.3).
- 4.7.3.3. Body replacement and execution. Finally the procedure body, modified as above, is inserted in place of the procedure statement and executed. If the procedure is called from a place outside the scope of any non-local quantity of the procedure body the conflicts between the identifiers inserted through this process of body replacement and the identifiers whose declarations are valid at the place of the procedure statement or function designator will be avoided through suitable systematic changes of the latter identifiers.
- 4.7.4. Actual-formal correspondence. The correspondence between the actual parameters of the procedure statement and the formal parameters of the procedure heading is established as follows: The actual parameter list of the procedure statement must have the same number of entries as the formal parameter list of the procedure declaration heading. The correspondence is obtained by taking the entries of these two lists in the same order.

Slika 7.1: Deo neformalne semantike za programski jezik Algol 60

Next, a `for` iteration step is performed, as follows:

- If the `Expression` is present, it is evaluated. If the result is of type `Boolean`, it is subject to unboxing conversion (5.1.8). If evaluation of the `Expression` or the subsequent unboxing conversion (if any) completes abruptly, the `for` statement completes abruptly for the same reason. Otherwise, there is then a choice based on the presence or absence of the `Expression` and the resulting value if the `Expression` is present:
- If the `Expression` is not present, or it is present and the value resulting from its evaluation (including any possible unboxing) is true, then the contained `Statement` is executed. Then there is a choice:
- If execution of the `Statement` completes normally, then the following two steps are performed in sequence: 1. First, if the `ForUpdate` part is present, the expressions are evaluated in sequence from left to right; their values, if any, are discarded. If evaluation of any expression completes abruptly for some reason, the for statement completes abruptly for the same reason; any `ForUpdate` statement expressions to the right of the one that completed abruptly are not evaluated. If the `ForUpdate` part is not present, no action is taken. 2. Second, another for iteration step is performed.
- If execution of the `Statement` completes abruptly, see 14.14.1.3 below.
- If the `Expression` is present and the value resulting from its evaluation (including any possible unboxing) is false, no further action is taken and the for statement completes normally.

Slika 7.2: Deo neformalne semantike za programski jezik Java

Primer 7.1.1 Semantika jezika opisanog u primeru 7.0.1 može se osloniti na standardnu semantiku brojeva, aritemtičkih, logičkih i relacionih izraza, kao i na standardne konstrukte za kontrolu toka u imperativnim jezicima:

n su odgovarajuće binarne vrednosti

a +, - i * su standardni aritmetički operatori sabiranja, oduzimanja i množenja

b `true` i `false` su standardne logičke konstante ⊤ i ⊥, = i <= su standardni relacioni operatori manje i manje-jednako, ! označava logičku negaciju, a && označava logičku konjunkciju

S naredbe programa mogu biti redom naredba dodele, naredba `skip` koja ne menja stanje programa, sekvenca naredbi, selekcija i iteracija.

Problemi sa neformalnim opisima:

Dvosmislenost se javlja kada neki konstrukt jezika nije adekvatno opisan: usled nedostatka potrebnih detalja ili usled upotrebe nepreciznih izraza moguća su različita tumačenja.

Nekonzistentnost (kontradikcije) se javlja u kompleksnim semantikama tako što se na različitim mestima opisuju međusobno zavisni konstrukti jezika na različite i pritom kontradiktorne načine. Nekonzistentnosti u semantici koja je neformalno zadata je teško pronaći, dok u formalno zadatim semantikama je moguće automatski proveriti konzistentnost.

Nedorečenost se javlja kada se prilikom definisanja semantike konstrukta neke informacije podrazumevaju. Nedorečenost često može da izazove i dvosmislenost.

Primer 7.1.2 Primetimo da je semantika zadata u primeru 7.1.1 u velikoj meri nedorečena. Osnovni problemi u prethodnoj semantici su što se podrazumeva „standardno” značenje za većinu jezičkih konstrukata. U okviru neformalnih definicija semantika programskih jezika ovakvo pozivanje na standardnu semantiku zapravo nije dozvoljeno već je sve potrebno detaljno opisati. I pored toga, za iskusne programere je ovakva semantika često sasvim dovoljna, pa čak i suvišna jer se mogla naslutiti i na osnovu same sintakse jezika.

Semantika programskog jezika se obično ne uči čitanjem specifikacije. Programeri obično razvijaju svoju mentalnu sliku semantike svakodnevnim korišćenjem jezika, tj. kroz kompajler/interperter . Znanje koje se stiče iskustvenim putem na osnovu toga šta neki programi rade je neprecizno i nepotpuno, ali je za neke obične upotrebe to sasvim u redu.

Međutim, za razvoj kompajlera neophodno je da se čita specifikacija programskog jezika. Ako je specifikacija neformalna, moguće je da će je različiti programeri različito razumeti

i da će se zato različiti kompajleri ponašati drugačije. Ako se kompajleri različito ponašaju, kako znati koji je usklađen sa predviđenom semantikom tog programskog jezika? Formalnom semantikom mogu se prevazići ovi problemi.

Formalna semantika se može zadati i za realne programske jezike — npr, semantika jezika ML je zadata formalno: "The definition of standard ML", MIT Press, M. Tofte, R. Milner, R. Harper.

7.2 Osnovne vrste formalnih semantika

Cilj formalnog zadavanja semantike je precizno definisati značenje (ponašanje) programa. Formalnom semantikom se definiše značenje samo sintaksno ispravnih programa. U skladu sa time, precizna semantika je definisana implementacijom kompajlera: na primer, precizna semantika jezika C/C++ je zapravo data kompajlerom *gcc* ili *clang*. Međutim, iako precizna, takva semantika za mnoge probleme nije upotrebljiva:

- Ukoliko nas interesuje da naučimo neki programski jezik, upotrebom kompajlera, bez dodatnih objašnjenja, to ne možemo da postignemo.
- Ukoliko nas interesuje da li su dve implementacije neke funkcije ekvivalentne, korišćenjem kompajlera, možemo da proverimo da li se te dve implementacije poklapaju na nekim ulazima, ali ne možemo da dokažemo njihovu ekvivalentnost.

Formalna semantika omogućava formalno rezonovanje o svojstvima programa. Na primer, formalna semantika nekog jezika, zajedno sa modelom programa (tranzicionim sistemom) omogućava ispitivanje različitih uslova ispravnosti/korektnosti tog programa, kao i odnosa između dva programa.

Primer: „Formalizacija LLVM međureprezentacije za verifikaciju transformacija programa" https://www.cis.upenn.edu/~stevez/vellvm/
Formalizing the LLVM Intermediate Representationfor Verified Program Transformations
https://www.cis.upenn.edu/~stevez/papers/ZNMZ12.pdf

Postoji veliki broj mogućnosti zadavanja semantike programa, naredna tri su osnovne vrste semantike

Operaciona semantika — Šta program radi?
  Odgovara izvršavanju u okviru apstraktne mašine

Denotaciona semantika — Šta program znači?
  Matematički objekti u nekom adekvatnom semantičkom domenu (npr. funkcija iz stanja u stanje, odnosno transformacija ulaza u izlaz)

Aksiomatska semantika — Koje osobine ima program?
  Kolekcija logičkih osobina koje obezbeđuje program (npr. preduslovi i postuslovi, prisutno u paradigmi *design by contract*)

Različitim programskim jezicima prirodno odgovaraju različite semantike. Na primer, za funkcionalne programske jezike često je prirodno definisati denotacionu semantiku, a za imperativne operacionu semantiku. Planirana upotreba semantike takođe utiče na izbor semantike:

- razumevanje jezika — neformalna semantika
- verifikacija i analiza programa — aksiomatska semantika
- pravljenje prototipa — operaciona semantika
- konstrukcija kompilatora — zavisi od jezika...

7.2.1 Rezonovanje o osobinama programa

O osobinama programa rezonuju svi programeri na svakodnevnom nivou. Za ovakvu vrstu rezonovanja, može da bude sasvim dovoljno i dobro poznavanje neformalne semantike:

- Da li kôd radi ono što želim? — Testiranjem možemo samo da podignemo pouzdanost da je kôd dobar, ali semantika nam govori šta se tu dešava i da li je to u skladu sa našim očekivanjima.
- Da li je zamena koda *boljim* kodom u redu (u okviru procesa optimizacije ili procesa refaktorisanja)?

Primer 7.2.1 Optimizacijom od koda koji radi, često dobijamo kôd koji je brži, ali ne radi. Na primer, da li je naredna jednostavna zamena

```
x=x+1;
x=x+1;
```

sa

```
x=x+2;
```

u redu u svakom kontekstu i u svakoj izvršnom okruženju, uključujući i konkurentno izvršavanje i atomičnost ili odsustvo atomičnosti izvršavanja sabiranja i naredbe dodele? Da bismo dali odgovor na ovo pitanje u nekom konkretnom programu, moramo da dobro poznajemo semantiku tog jezika i uopšte kontekst izvršavanja datog programa.

O osobinama programa rezonuju i dizajneri programskih jezika. Za njih formalna semantika može da bude veoma korisna jer omogućava brz razvoj prototipa jezika. U tom kontekstu je posebno važna operaciona semantika jer ona daje opis apstraktne mašine koja izvršava program i na taj

način omogućava automatizaciju razvoja prototipa. Takođe, ako je za neki konstrukt programskog jezika potrebno previše prostora da se formalno zapiše, onda je verovatno u pitanju loša ideja koju ne treba implementirati jer je previše kompleksna i niko je neći razumeti kako treba.

U okviru razvoja kompajlera od posebne važnosti su optimizacije i impelementacije naprednih tehnika koje podrazumevaju netrivijalne transformacije koda kako bi se kod preveo sa visokog apstraktnog nivoa u ekvivalentan kod koji se efikasno izvršava (na primer, u kontekstu funkcionalnih jezika potrebno je ostvariti podršku za beskonačne strukture podataka i lenjo izračunavanje). U kontekstu razvoja kompajlera rezonovanje o semantici programskog jezika je posebno važno jer neispravne optimizacije i transformacije programa mogu da budu veoma opasne i da kada se ispolje naprave velike štete.

dodati primer za SQL

Primer 7.2.2 (Nastavak primera 7.0.1.) Nezavisno od tipa semantike, može se definisati preslikavanje $\mathcal{N}$ koje numeralima definisanim u primeru 7.0.1 dodeljuje ceo broj $\mathcal{N} : Num \longrightarrow Z$

$$
\begin{aligned}
\mathcal{N}[\![0]\!] &= \mathbf{0} \\
\mathcal{N}[\![1]\!] &= \mathbf{1} \\
\mathcal{N}[\![n\ 0]\!] &= \mathbf{2} \cdot \mathcal{N}[\![n]\!] \\
\mathcal{N}[\![n\ 1]\!] &= \mathbf{2} \cdot \mathcal{N}[\![n]\!] + \mathbf{1}
\end{aligned}
$$

Na ovaj način, se niskama karaktera formiranih od nula i jedinica (sintaksnim objektima) pridružuje značenje, odnosno odgovarajući celi (zapravo prirodni) brojevi.

Stanje označava trenutnu sliku memorije, tj koju vrednost ima koja promenljiva:

$$State : Var \longrightarrow Z$$

pri čemu je $Var \longrightarrow Z$ funkcija iz skupa promenljivih Var u skup celih brojeva Z. Na primer, neka je $Var = \{x, y, z\}$. Jedno stanje može biti zadato listom $[x \longrightarrow 2, y \longrightarrow 5, z \longrightarrow 0]$.

Stanje određuje vrednost za promenljivu, pa tako $s\ x$ označava vrednost promenljive x u stanju s. Ta vrednost, za prethodno stanje je 2. Vrednost izraza $x + 1$ u tom stanju

bi bila 3.

Semantika aritmetičkog izraza
Za dat aritmetički izraz $Aexp$ i za zadato stanje $State$ funkcija $\mathcal{A}$ definiše vrednost aritmetičkog izraza u datom stanju.

$$\mathcal{A} : Aexp \longrightarrow (State \longrightarrow Z)$$

$$\begin{aligned}
\mathcal{A}[\![n]\!]s &= \mathcal{N}[\![n]\!] \\
\mathcal{A}[\![x]\!]s &= s\ x \\
\mathcal{A}[\![a_1 + a_2]\!]s &= \mathcal{A}[\![a_1]\!]s + \mathcal{A}[\![a_2]\!]s \\
\mathcal{A}[\![a_1 \star a_2]\!]s &= \mathcal{A}[\![a_1]\!]s \cdot \mathcal{A}[\![a_2]\!]s \\
\mathcal{A}[\![a_1 - a_2]\!]s &= \mathcal{A}[\![a_1]\!]s - \mathcal{A}[\![a_2]\!]s
\end{aligned}$$

Prvi red definicije govori da ukoliko je izraz numeral, vrednost numerala ne zavisi od stanja već samo od funkcije $\mathcal{N}$. Drugi red definicije govori da ukoliko je izraz promenljiva, vrednost izraza je vrednost promenljive u stanju za koje se vrednost izraza razmatra. Naredna tri reda definišu rekurzivno vrednost izraza prevodeći sintaksne konstrukte +, - i * u odgovarajuće izraze nad celim brojevima.

Semantika logičkog izraza
Neka je $T = \{tt, ff\}$ gde tt označava tačno, a ff netačno. Za dat logički izraz $Bexp$ i za zadato stanje $State$ funkcija $\mathcal{B}$ definiše vrednost logičkog izraza u datom stanju

$$\mathcal{B} : Bexp \longrightarrow (State \longrightarrow T)$$

$$\begin{aligned}
\mathcal{B}[\![\texttt{true}]\!]s &= \mathbf{tt} \\
\mathcal{B}[\![\texttt{false}]\!]s &= \mathbf{ff} \\
\mathcal{B}[\![a_1 = a_2]\!]s &= \begin{cases} \mathbf{tt} & \text{if } \mathcal{A}[\![a_1]\!]s = \mathcal{A}[\![a_2]\!]s \\ \mathbf{ff} & \text{if } \mathcal{A}[\![a_1]\!]s \neq \mathcal{A}[\![a_2]\!]s \end{cases} \\
\mathcal{B}[\![a_1 \leq a_2]\!]s &= \begin{cases} \mathbf{tt} & \text{if } \mathcal{A}[\![a_1]\!]s \leq \mathcal{A}[\![a_2]\!]s \\ \mathbf{ff} & \text{if } \mathcal{A}[\![a_1]\!]s > \mathcal{A}[\![a_2]\!]s \end{cases} \\
\mathcal{B}[\![\neg\ b]\!]s &= \begin{cases} \mathbf{tt} & \text{if } \mathcal{B}[\![b]\!]s = \mathbf{ff} \\ \mathbf{ff} & \text{if } \mathcal{B}[\![b]\!]s = \mathbf{tt} \end{cases} \\
\mathcal{B}[\![b_1 \wedge b_2]\!]s &= \begin{cases} \mathbf{tt} & \text{if } \mathcal{B}[\![b_1]\!]s = \mathbf{tt} \text{ and } \mathcal{B}[\![b_2]\!]s = \mathbf{tt} \\ \mathbf{ff} & \text{if } \mathcal{B}[\![b_1]\!]s = \mathbf{ff} \text{ or } \mathcal{B}[\![b_2]\!]s = \mathbf{ff} \end{cases}
\end{aligned}$$

Semantika
Neka je Stm je skup svih programa. Za dati program funkcija $\mathcal{S}$ vraća značenje (odnosno semantiku) kao preslikavanje iz stanja u stanje.

$$\mathcal{S} : Stm \rightarrow (State \hookrightarrow State)$$

pri čemu je $State \hookrightarrow State$ skup parcijalnih funkcija iz $State$ u $State$.

Parcijalnost označava da funkcija može biti nedefinisana za neke ulazne parametre (npr program se ne zaustavlja, beskonačna petlja, beskonačna rekurzija...).

7.3 Operaciona semantika

Operaciona semantika opisuje kako se izračunavanje izvršava. Najčešće se koristi za opis i rezonovanje o imperativnim jezicima jer individualni koraci izračunavanja opisuju na koji način se menja stanje programa. U okviru ove semantike postoje **prirodna operaciona semantika** (eng. *big-step semantics*), opisuje ukupne rezultate izračunavanja i **strukturna operaciona semantika** (eng. *small-step semantics*), opisuje individualne korake izračunavanja. Zadavanje semantike za različite jezike i koncepte može da varira.

7.3.1 Prirodna semantika

U okviru prirodne semantike koristi se naredna oznaka

$$\langle\, S, s \,\rangle \rightarrow s'$$

Intuitivno ona znači da izvršavanje programa S sa ulaznim stanjem s će se završiti i rezultujuće stanje će biti s'.

Pravila generalno imaju formu

$$\frac{\langle S_1,s_1\rangle \rightarrow s_1', \ldots, \langle S_n,s_n\rangle \rightarrow s_n'}{\langle S,s\rangle \rightarrow s'}$$

gde $S_1, \ldots, S_n$ nazivamo neposrednim konstituentima (eng. *immediate constituents*) od S. Pravilo se sastoji iz određenog broja *premisa* (nalaze se iznad linije) i jednog *zaključka* (nalazi se

ispod linije). Pravilo sa praznim skupom premisa se naziva *aksioma*.

Pravilo takođe može imati određeni broj *uslova* (nalaze se sa desne strane linije) koji moraju biti ispunjeni kako bi se primenilo pravilo.

Primer 7.3.1 Primer prirodne semantike za jezik iz primera 7.0.1 i 8.2.2.

$[ass_{ns}]$	$\langle x := a, s\rangle \to s[x \mapsto \mathcal{A}[\![a]\!]s]$
$[skip_{ns}]$	$\langle skip, s\rangle \to s$
$[comp_{ns}]$	$\frac{\langle S_1,s\rangle\to s', \langle S_2,s'\rangle\to s''}{\langle S_1;S_2,s\rangle\to s''}$
$[if^{tt}_{ns}]$	$\frac{\langle S_1,s\rangle\to s'}{\langle if\ b\ then\ S_1\ else\ S_2,\ s\rangle\to s'}\ if\ \mathcal{B}[\![b]\!]s = tt$
$[if^{ff}_{ns}]$	$\frac{\langle S_2,s\rangle\to s'}{\langle if\ b\ then\ S_1\ else\ S_2,\ s\rangle\to s'}\ if\ \mathcal{B}[\![b]\!]s = ff$
$[while^{tt}_{ns}]$	$\frac{\langle S,s\rangle\to s', \langle\ while\ b\ do\ S,\ s'\rangle\to s''}{\langle\ while\ b\ do\ S,\ s\rangle\to s''}\ if\ \mathcal{B}[\![b]\!]s = tt$
$[while^{ff}_{ns}]$	$\langle\ while\ b\ do\ S,\ s\rangle \to s\ if\ \mathcal{B}[\![b]\!]s = ff$

Primer 7.3.2 z := x; x := y; y := z
Neka s_0 bude stanje koje preslikava x u 5, y u 7 i z u 0. Tada dobijamo sledeće stablo izvođenja:

$$\frac{\frac{\langle z:=x,s_0\rangle\to s_1 \quad \langle x:=y,s_1\rangle\to s_2}{\langle z:=x;x:=y,s_0\rangle\to s_2} \quad \langle y:=z,s_2\rangle\to s_3}{\langle z:=x;x:=y;y:=z,s_0\rangle\to s_3}$$

Važi $s_1 = s_0[z \mapsto 5]$, $s_2 = s_1[x \mapsto 7]$, $s_3 = s_2[y \mapsto 5]$. Stablo izvođenja ima tri primene aksiome $[ass_{ns}]$, dok je $[comp_{ns}]$ primenjeno dva puta.

Prethodno izvođenje nije interesantno u smislu da ga izvodi čovek, već je interesantno jer može da ga izvodi računar.

Kada imamo definisanu semantiku, to nam omogućava da rezonujemo o osobinama programa.

- Ako želimo da dokažemo da nešto nije ispunjeno, dovoljno je naći primer koji narušava dato svojstvo.
- Ako želimo da dokažemo da nešto uvek važi, onda to ne može da se uradi primerom, potrebno je da postoji semantika koja nam daje mogućnost da matematički dokažemo da je svojstvo uvek ispunjeno.

Problem da li su dve naredbe semantički ekvivalentne možemo da formulišemo kao pitanje da li za svaka dva stanja s i s' važi sledeće

$$\langle S_1, s\rangle \rightarrow s' \text{ akko } \langle S_2, s\rangle \rightarrow s'$$

Dokazi se obično izvode indukcijom po obliku stabla izvođenja:

- Dokaži da svojstvo važi za svako jednostavno stablo pokazivanjem da važi za aksiome tranzicionog sistema.
- Dokaži da svojstvo važi i za sva složena stabla izvođenja: za svako pravilo pretpostavi da svojstvo važi za njegove premise (induktivna hipoteza) i dokaži da takođe važi i za zaključak pravila ukoliko su uslovi pravila zadovoljeni.

Lemma 7.3.1 *Naredba*

```
1 while b do S
```

je semantički ekvivalentna sa narednim kodom:

```
1 if b then ( S; while b do S )
2 else skip
```

Dokaz. Potrebno je dokazati dva smera, tj

$$\langle while\ b\ do\ S, s\rangle \rightarrow s''$$

ako i samo ako

$$\langle if\ b\ then\ (S;\ while\ b\ do\ S)\ else\ skip, s\rangle \rightarrow s''$$

Dokaz se izvodi konstruisanjem stabla izvođenja na osnovu pravila izvođenja koja su data semantikom. Na primer, ako pretpostavimo da važi

$$\langle while\ b\ do\ S, s\rangle \rightarrow s''$$

onda postoji stablo izvođenja kojima se dolazi do stanja s″. Ukoliko pogledamo pravila izvođenja, do takvog stabla možemo da dođemo samo primenim pravila $[while^{tt}_{ns}]$ ili pravila $[while^{ff}_{ns}]$. Potrebno je razmotriti i jedan i drugi slučaj.

(Prvi slučaj): ukoliko se primeni pravilo $[while^{tt}_{ns}]$ to se

onda stablo izvođenja svodi na primenu ovog pravila:

$$\frac{\langle S, s\rangle \rightarrow s', \langle\ while\ b\ do\ S,\ s'\rangle \rightarrow\ s''}{\langle\ while\ b\ do\ S,\ s\rangle \rightarrow\ s''}$$

pri čemu važi $\mathcal{B}[\![b]\!]s\ =\ tt$ odnosno stablo izvođenja izgleda ovako

$$\frac{T_1 \quad T_2}{\langle\ while\ b\ do\ S,\ s\rangle \rightarrow\ s''}$$

pri čemu je T_1 nekakvo izvođenje za $\langle S, s\rangle \rightarrow s'$ a T_2 nekakvo izvođenje za $\langle\ while\ b\ do\ S,\ s'\rangle \rightarrow\ s''$

Koristeći T_1 i T_2, možemo da primenimo i pravilo kompozicije $[comp_{ns}]$ i da izvedemo sledeći zaključak

$$\frac{T_1 \quad T_2}{\langle S;\ while\ b\ do\ S,\ s\rangle \rightarrow\ s''}$$

Pošto znamo da je $\mathcal{B}[\![b]\!]s\ =\ tt$, možemo da primenimo $[if^{tt}_{ns}]$ i da nam stablo izvođenja sada izgleda ovako, čime smo pokazali da željeno svojstvo važi:

$$\frac{\frac{T_1 \quad T_2}{\langle S;\ while\ b\ do\ S,\ s\rangle \rightarrow\ s''}}{\langle if\ b\ then\ (S;\ while\ b\ do\ S)\ else\ skip, s\rangle \rightarrow s''}$$

Dalje je na sličan način potrebno razmotriti $[while^{ff}_{ns}]$ □

7.3.2 Strukturna operaciona semantika

U okviru strukturne operacione semantike tranzicionu relaciju zapisujemo ovako

$$\langle\ S, s\ \rangle \Longrightarrow \gamma$$

Ovo treba razmatrati kao prvi korak izvršavanja programa S u stanju s koji vodi do stanja γ.
Možemo razlikovati dva slučaja za γ:

1. $\gamma = \langle\ S', s'\ \rangle$: izvršavanje programa S sa ulaznim stanjem s nije završeno, i ostatak izračunavanja će biti izraženo srednjom konfiguracijom $\langle\ S', s'\ \rangle$.
2. $\gamma = s'$: izvršavanje programa S sa ulaznim stanjem s se završilo sa završnim stanjem s'.

Primer 7.3.3 (Nastavak primera 7.2.2.) Strukturna operaciona semantika

$[\mathrm{ass}_{\mathrm{sos}}]$	$\langle x := a,\ s\rangle \Rightarrow s[x \mapsto \mathcal{A}[\![a]\!]s]$
$[\mathrm{skip}_{\mathrm{sos}}]$	$\langle \texttt{skip},\ s\rangle \Rightarrow s$
$[\mathrm{comp}^{1}_{\mathrm{sos}}]$	$\dfrac{\langle S_1,\ s\rangle \Rightarrow \langle S_1',\ s'\rangle}{\langle S_1; S_2,\ s\rangle \Rightarrow \langle S_1'; S_2,\ s'\rangle}$
$[\mathrm{comp}^{2}_{\mathrm{sos}}]$	$\dfrac{\langle S_1,\ s\rangle \Rightarrow s'}{\langle S_1; S_2,\ s\rangle \Rightarrow \langle S_2,\ s'\rangle}$
$[\mathrm{if}^{\mathrm{tt}}_{\mathrm{sos}}]$	$\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \Rightarrow \langle S_1,\ s\rangle$ if $\mathcal{B}[\![b]\!]s = \mathbf{tt}$
$[\mathrm{if}^{\mathrm{ff}}_{\mathrm{sos}}]$	$\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \Rightarrow \langle S_2,\ s\rangle$ if $\mathcal{B}[\![b]\!]s = \mathbf{ff}$
$[\mathrm{while}_{\mathrm{sos}}]$	$\langle \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \Rightarrow$ $\langle \texttt{if}\ b\ \texttt{then}\ (S;\ \texttt{while}\ b\ \texttt{do}\ S)\ \texttt{else}\ \texttt{skip},\ s\rangle$

Strukturna operaciona semantika omogućava praćenje „sitnijih detalja" izračunavanja.

Slično kao kod prirodne semantike, i u okviru strukturne operacione semantike moguće je dokazati razna interesantna svojstva jezika. U ovom slučaju, dokazi se izvode kao indukcija po dužini izvođenja:

- Dokaži da svojstvo važi za izvođenje dužine nula.
- Dokaži da svojstvo važi i za sva konačna izvođenja: pretpostavi da svojstvo važi za dužinu izvođenja k (induktivna hipoteza) i dokaži da takođe važi i za dužinu izvođenja $k + 1$.

Za ove prirodnu i strukturnu operacionu semnatiku se može dokazati ekvivalentnost: za svako izvođenje u okviru prirodne semantike postoji odgovarajuće izvođenje u okviru strukturne operacione semantike i obrnuto. Dodavanjem naprednijih koncepata jezika, na primer, pozivi funkcija, paralelno izvršavanje..., ove semantike omogućavaju izražavanje i dokazivanje različitih svojstava.

7.4 Denotaciona semantika

Denotaciona semantika definiše značenje prevođenjem u drugi jezik, za koji se pretpostavlja da je poznata semantika. Najčešće je taj drugi jezik nekakav matematički formalizam. Povezivanje svakog dela programskog jezika sa nekim matematičkim objektom kao što je broj ili funkcija: svaka sintaksna

definicija se tretira kao objekat na koji se može primeniti funkcija koja taj objekat preslikava u matematički objekat koji definiše značenje.

Dodeljivanjem značenja delovima programa dodeljuje se značenje celokupnom programu, tj **semantika jedne programske celine definisana je preko semantike njenih poddelova.** Ova osobina denotacione semantike naziva se **kompozitivnost**. Funkcije prirodne i strukturne operacione semantike nisu primeri denotacionih definicija jer ne zadovoljavaju svojstvo kompozitivnosti. Definisane funkcije $\mathscr{A}$ i $\mathscr{B}$ su primeri denotacionih definicija jer su to funkcije koje zadovoljavaju svojstvo kompozitivnosti.

Primer 7.4.1 (Kompozitivnost) Razmotrimo jednostavan primer jezika aritmetičkih izraza koji brojeve u osnovi 10 i operator $\oplus$.

Koristićemo skraćenice: C za cifre, N za numerale i I za izraze.

Sintaksa:

$$
\begin{aligned}
C &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\
N &::= C \mid N\ C \\
I &::= N \mid I \oplus I
\end{aligned}
$$

Sledeći korak jeste definisanje matematičkih objekata koji će predstavljati semantičke vrednosti. Ti matematički objekti čine **semantički domen**. Kompleksnost semantičkog domena zavisi od toga koliko je kompleksan programski jezik kojem dajemo značenje, u ovom jednostavnom primeru, semantička vrednost je prirodan broj.

Semantički domen

$\mathbb{N} = \mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3}, \ldots.$ Skup prirodnih brojeva

Radi lakšeg raspoznavanja sintaksnih i semantičkih objekata, prirodni brojevi su napisani podebljano.

Funkcije značenja daju značenje uvedenim sintaksnim definicijama. One povezuju sintaksne objekte sa njihovom semantikom. Intuitivno, funkcija $\mathscr{C}$ povezuje cifru sa prirodnim brojem, funckija $\mathscr{N}$ povezuje numeral sa prirodnim

brojem, a funkcija $\mathcal{I}$ povezuje izraz sa prirodnim brojem. U okviru definisanja semantike, koriste se i pomoćne funkcije sabiranja i množenja prirodnih brojeva, + i ·.

Definicije funkcija značenja (definicija semantike)

$$\mathcal{C} : C \to \mathbb{N}, \quad \mathcal{N} : N \to \mathbb{N}, \quad \mathcal{I} : I \to \mathbb{N}$$

$$\begin{aligned}
\mathcal{C}[\![0]\!] &= \mathbf{0} \\
&\dots \\
\mathcal{C}[\![9]\!] &= \mathbf{9} \\
\\
\mathcal{N}[\![C]\!] &= \mathcal{C}[\![C]\!] \\
\mathcal{N}[\![NC]\!] &= \mathbf{10} \cdot \mathcal{N}[\![N]\!] \; + \; \mathcal{N}[\![C]\!] \\
\\
\mathcal{I}[\![N]\!] &= \mathcal{N}[\![N]\!] \\
\mathcal{I}[\![I_1 \oplus I_2]\!] &= \mathcal{I}[\![I_1]\!] \; + \; \mathcal{I}[\![I_2]\!]
\end{aligned}$$

Zagrade $[\![,]\!]$ imaju ulogu da razdvoje semantički deo od sintaksnog dela. U okviru zagrada nalazi se sintaksni deo izraza, a funkcija značenja za datu sintaksu pridružuje odgovarajuću semantiku.
Na primer, cifra 9 kao sintaksni objekat nema značenje, dok prirodni broj **9** daje značenje pridruživanjem semantike cifri 9 i to se zapisuje sa $\mathcal{C}[\![9]\!] = \mathbf{9}$.

Pronaći značenje izraza $2 \oplus 32 \oplus 61$.

$$\begin{aligned}
\mathcal{I}[\![2 \oplus 32 \oplus 61]\!] &= \mathcal{I}[\![2 \oplus 32]\!] + \mathcal{I}[\![61]\!] \\
&= (\mathcal{I}[\![2]\!] + \mathcal{I}[\![32]\!]) + \mathcal{N}[\![61]\!] \\
&= (\mathbf{2} + \mathbf{32}) + \mathbf{61} = \mathbf{34} + \mathbf{61} = \mathbf{95}
\end{aligned}$$

jer je:

$$
\begin{aligned}
\mathcal{I}[\![2]\!] &= \mathcal{N}[\![2]\!] \\
&= \mathcal{C}[\![2]\!] \\
&= \mathbf{2} \\
\mathcal{I}[\![32]\!] &= \mathcal{N}[\![32]\!] \\
&= \mathbf{10} \cdot \mathcal{N}[\![3]\!] + \mathcal{N}[\![2]\!] \\
&= \mathbf{10} \cdot \mathcal{C}[\![3]\!] + \mathcal{C}[\![2]\!] \\
&= \mathbf{10} \cdot \mathbf{3} + \mathbf{2} = \mathbf{30} + \mathbf{2} = \mathbf{32} \\
\mathcal{I}[\![61]\!] &= \mathcal{N}[\![61]\!] \\
&= \mathbf{10} \cdot \mathcal{N}[\![6]\!] + \mathcal{N}[\![1]\!] \\
&= \mathbf{10} \cdot \mathcal{C}[\![6]\!] + \mathcal{C}[\![1]\!] \\
&= \mathbf{10} \cdot \mathbf{6} + \mathbf{1} = \mathbf{60} + \mathbf{1} = \mathbf{61}
\end{aligned}
$$

Primer 7.4.2 (Nastavak primera 7.2.2.) U okviru primera zadavanja denotacione semantike za razmatrani jezik, oznaka $\circ$ je kompozicija funkcija, a koriste se i pomoćne funkcije $cond$ i FIX.

$$
\begin{aligned}
\mathcal{S}_{\mathrm{ds}}[\![x := a]\!]s &= s[x \mapsto \mathcal{A}[\![a]\!]s] \\
\mathcal{S}_{\mathrm{ds}}[\![\texttt{skip}]\!] &= \mathrm{id} \\
\mathcal{S}_{\mathrm{ds}}[\![S_1 \ ; \ S_2]\!] &= \mathcal{S}_{\mathrm{ds}}[\![S_2]\!] \circ \mathcal{S}_{\mathrm{ds}}[\![S_1]\!] \\
\mathcal{S}_{\mathrm{ds}}[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!] &= \mathsf{cond}(\mathcal{B}[\![b]\!], \mathcal{S}_{\mathrm{ds}}[\![S_1]\!], \mathcal{S}_{\mathrm{ds}}[\![S_2]\!]) \\
\mathcal{S}_{\mathrm{ds}}[\![\texttt{while } b \texttt{ do } S]\!] &= \mathsf{FIX}\ F \\
\text{where } F\ g &= \mathsf{cond}(\mathcal{B}[\![b]\!], g \circ \mathcal{S}_{\mathrm{ds}}[\![S]\!], \mathrm{id})
\end{aligned}
$$

Efekat kompozicije naredbi je kompozicija funkcija izvršavanja nardbi S_1 i S_2. Kompozicija funkcija je definisana tako da ako je jedna funkcija nedefinisana za dato stanje, onda je njihova kompozicija takođe nedefinisana.

$$
\mathcal{S}_{\mathrm{ds}}[\![S_1 \ ; \ S_2]\!]s = (\mathcal{S}_{\mathrm{ds}}[\![S_2]\!] \circ \mathcal{S}_{\mathrm{ds}}[\![S_1]\!])s
$$
$$
= \begin{cases}
s'' & \text{if there exists } s' \text{ such that } \mathcal{S}_{\mathrm{ds}}[\![S_1]\!]s = s' \\
 & \text{and } \mathcal{S}_{\mathrm{ds}}[\![S_2]\!]s' = s'' \\
\underline{\text{undef}} & \text{if } \mathcal{S}_{\mathrm{ds}}[\![S_1]\!]s = \underline{\text{undef}} \\
 & \text{or if there exists } s' \text{ such that } \mathcal{S}_{\mathrm{ds}}[\![S_1]\!]s = s' \\
 & \text{but } \mathcal{S}_{\mathrm{ds}}[\![S_2]\!]s' = \underline{\text{undef}}
\end{cases}
$$

Primer 7.4.3 Primer kompozicije naredbi dodele. Kako program predstavlja skup naredbi odvojenih simbolom ';', efekat celog programa je kompozicija efekata svih individualnih naredbi.
$S_{ds}[\![z := x; x := y; y := z]\!] =$
$S_{ds}[\![y := z]\!] \circ S_{ds}[\![x := y]\!] \circ S_{ds}[\![z := x]\!]$
Za konkretne vrednosti:

$$\begin{aligned} S_{ds}[\![z := x; x := y; y := z]\!]([x \to 5, y \to 7, z \to 0]) &= \\ S_{ds}[\![y := z]\!] \circ S_{ds}[\![x := y]\!] \circ S_{ds}[\![z := x]\!]([x \to 5, y \to 7, z \to 0]) &= \\ S_{ds}[\![y := z]\!] \circ S_{ds}[\![x := y]\!]([x \to 5, y \to 7, z \to 5]) &= \\ S_{ds}[\![y := z]\!]([x \to 7, y \to 7, z \to 5]) &= \\ [x \to 7, y \to 5, z \to 5] & \end{aligned}$$

Primer 7.4.4 Funkcija $cond$ je uvedena za modelovanje `if-then-else` naredbe

$$\text{cond}: (\textbf{State} \to \textbf{T}) \times (\textbf{State} \hookrightarrow \textbf{State}) \times (\textbf{State} \hookrightarrow \textbf{State}) \to (\textbf{State} \hookrightarrow \textbf{State})$$

$$\text{cond}(p,\ g_1,\ g_2)\ s = \begin{cases} g_1\ s & \text{if } p\ s = \textbf{tt} \\ g_2\ s & \text{if } p\ s = \textbf{ff} \end{cases}$$

$$\begin{aligned} &\mathcal{S}_{\text{ds}}[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!]\ s \\ =\ &\text{cond}(\mathcal{B}[\![b]\!], \mathcal{S}_{\text{ds}}[\![S_1]\!], \mathcal{S}_{\text{ds}}[\![S_2]\!])\ s \\ =\ &\begin{cases} s' & \text{if } \mathcal{B}[\![b]\!]s = \textbf{tt} \text{ and } \mathcal{S}_{\text{ds}}[\![S_1]\!]s = s' \\ & \text{or if } \mathcal{B}[\![b]\!]s = \textbf{ff} \text{ and } \mathcal{S}_{\text{ds}}[\![S_2]\!]s = s' \\ \underline{\text{undef}} & \text{if } \mathcal{B}[\![b]\!]s = \textbf{tt} \text{ and } \mathcal{S}_{\text{ds}}[\![S_1]\!]s = \underline{\text{undef}} \\ & \text{or if } \mathcal{B}[\![b]\!]s = \textbf{ff} \text{ and } \mathcal{S}_{\text{ds}}[\![S_2]\!]s = \underline{\text{undef}} \end{cases} \end{aligned}$$

dadati primer

Primer 7.4.5 Funkcija FIX
Najteže je definisati semantiku za naredbu *while*, a da se pritom zadrži kompozitivnost. Prilikom definisanja, mora da važi ekvivalentnost naredbe *while* sa narednom naredbom

```
1 if b then (S ; while b do S) else skip
```

S obzirom na defnisan if i definisan $skip$, potrebno je da

važi sledeće

$$\mathcal{S}_{ds}[\![\texttt{while } b \texttt{ do } S]\!] =$$
$$\textbf{cond}(\mathcal{B}[\![b]\!], \mathcal{S}_{ds}[\![\texttt{while } b \texttt{ do } S]\!] \circ \mathcal{S}_{ds}[\![S]\!], id)$$

Primetimo da to ne može da se koristi kao definicija za *while*, jer nema svojstvo kompozitivnosti.

Međutim, prethodno nam pokazuje da je

$$\mathcal{S}_{ds}[\![\texttt{while } b \texttt{ do } S]\!]$$

fiksna tačka funkcionala

$$F\ g = \textbf{cond}(\mathcal{B}[\![b]\!], g \circ \mathcal{S}_{ds}[\![S]\!], id)$$

odnosno, važi

$$\mathcal{S}_{ds}[\![\texttt{while } b \texttt{ do } S]\!] = F\ (\mathcal{S}_{ds}[\![\texttt{while } b \texttt{ do } S]\!])$$

Na taj način se dobija kompozitivnost, i to zapisujemo na sledeći način:

$$\mathcal{S}_{ds}[\![\texttt{while } b \texttt{ do } S]\!] = \textbf{FIX}\, F$$

pri čemu važi

$$F\ g = \textbf{cond}(\mathcal{B}[\![b]\!], g \circ \mathcal{S}_{ds}[\![S]\!], id)$$

Pri čemu je tip funkcije FIX

FIX: $((\textbf{State} \hookrightarrow \textbf{State}) \rightarrow (\textbf{State} \hookrightarrow \textbf{State})) \rightarrow (\textbf{State} \hookrightarrow \textbf{State})$

Međutim, ne mora svaka funkcija da ima fiksnu tačku, i takođe ako postoji fiksna tačka ne mora da bude jedinstvena. Zbog toga je potrebno dodatno definisati osobine funkcije FIX kako bi ova definicija bila ispravna.

dadati primer

Denotaciona semantika apstrahuje izvršavanje programa. Koristi se često za definisanje semantike funkcionalnih programskih jezika — funkcionalno programiranje zasniva se na pojmu matematičkih funkcija i izvršavanje programa svodi se na evaluaciju funkcija. Međutim, prethodni primer je bio vezan za jezik koji je imperativan. Analiziranje programa se svodi na analiziranje matematičkih objekata, što olakšava formalno dokazivanje semantičkih svojstava programa. Može se dokazati ekvivalentnost prethodno definisanih operacionih

semantika sa denotacionom semantikom.

7.5 Aksiomatska semantika

Aksiomatska semantika omogućava viši nivo apstrakcije i udaljavanje od konkretne semantike programskog jezika. Aksiomska semantika zasniva se na Horovoj logici.

Horova trojka `{P} S {Q};` gde se `{P}` naziva preduslov a `{Q}` postuslov, opisuje kako izvršavanje dela koda menja stanje izračunavanja:

- ako je ispunjen preduslov `{P}`,
- ako se izvršavanje komande `S` završava za dato stanje
- onda će `{Q}` da važi u stanju u kojme se `S` završilo

Primer 7.5.1 Razmotrimo kôd koji računa faktorijel prirodnog broja

```
{x = n}
y = 1;
while (x != 1) {
  y = x * y;
  x = x - 1;
}
{y = n!, n > 0}
```

Promenljiva n u ovom primeru je specijalna *logička* promenljiva i ona se ne pojavljuje u razmatranim naredbama. Njena uloga je da zapamti inicijalnu vrednost promenljive x. Ne možemo zapisati $y = x! \wedge x > 0$, jer bi to označavalo finalnu vrednost promenljive x, koja nije ista sa početnom. Dakle, imamo dve vrste promenljivih: logičke i programske promenljive. Logičke promenljive se ne menjaju u okviru programa i njihove vrednosti su fiksirane.

Postoje dva pristupa definisanju preduslova i postuslova:

intenzionalni pristup — ideja je da se uvede novi eksplicitini jezik koji se koristi za opisivanje uslova, pri čemu taj jezik treba da bude dovoljno bogat i izražajan.

ekstenzioni pristup — ideja je da se uslovi opisuju kao predikati, tj funkcije iz $State \rightarrow T$, jednostavniji pristup.

Primer 7.5.2 Za razmatrani jezik, aksiomatska semantika

se može definisati na sledeći način (sheme aksioma koje se generišu posebno za svako svojstvo P koje je od intresa):

$[ass_p]$	$\{P[x \to \mathscr{A}[\![a]\!]]\}\ x := a\ \{P\}$
$[skip_p]$	$\{P\}$ skip $\{P\}$
$[comp_p]$	$\dfrac{\{P\}\ S_1\ \{Q\},\ \{Q\}\ S_2\ \{R\}}{\{P\}\ S_1;\ S_2\{R\}}$
$[if_p]$	$\dfrac{\{\mathcal{B}[\![b]\!] \wedge P\}\ S_1\ \{Q\},\ \{\neg\mathcal{B}[\![b]\!] \wedge P\}\ S_2\ \{Q\}}{\{P\}\ if\ b\ then\ S_1\ else\ S_2\ \{Q\}}$
$[while_p]$	$\dfrac{\{\mathcal{B}[\![b]\!] \wedge P\}\ S\ \{P\}}{\{P\}\ while\ b\ do\ S\ \{\neg\mathcal{B}[\![b]\!] \wedge P\}}$
$[cons_p]$	$\dfrac{\{P'\}\ S\ \{Q'\}}{\{P\}\ S\ \{Q\}}$ ako $P \Rightarrow P'$ i $Q' \Rightarrow Q$

Aksioma dodele ass_p: ako stanje s u okviru kojeg je je x postavljeno na $\mathscr{A}[a]$ zadovoljava svojstvo P, tada će svojstvo P da važi nakon dodele $x := a$.

Aksioma skip $skip_p$: ako P važi pre naredbe $skip$, onda važi i posle.

Pravilo izvođenja — pravilo kompozicije $comp_p$: Ako P važi pre S_1 i Q važi nakon izvršavanja S_1, i ako Q važi pre izvršavanja S_2 i R važi nakon S_2 onda možemo da zaključimo da će ako važi P pre izvršavanja $S_1; S_2$, nakon izvršavanja da važi R.

Pravilo izvođenja — uslova if_p: Ako P važi i ispunjen je uslov b pre izvršavanja S_1 i Q važi nakon izvršavanja S_1, i ako P važi i nije ispunjen uslov b pre izvršavanja S_2 i Q važi nakon izvršavanja S_2, onda možemo da zaključimo da ako P važi pre izvršavanja if naredbe, da će Q da važi nakon izvršavanja.

Pravilo izvođenja — petlje $while_p$: Ako P važi i ispunjen je uslov b pre izvršavanja S i ako P važi nakon izvršavanja S, onda ukoliko je uslov P ispunjen pre ulaska u petlju, tada je uslov P ispunjen i nakon izlaska iz petlje, pri čemu tada b (tj uslov ulaska u petlju) nije ispunjen. Pravilo petlje definiše P kao invarijantu petlje koja važi pre i nakon izvršavanja petlje.

Pravilo jačanja preduslova i slabljenja postuslova $cons_p$: ako važi $\{P'\}\ S\ \{Q'\}$ i ako se iz P može zaključiti P' i ako se iz Q' može zaključiti Q onda važi i $\{P\}\ S\ \{Q\}$. Odnosno, jači preduslov P garantuje da će nakon izvršavanja S i dalje da važi Q'. Slabiji postuslov garantuje da će nakon izvršavanje S na osnovu od-

govarajućeg preduslova, sigurno da važi i Q jer ako znamo da će da važi Q', onda na osnovu toga sigurno važi i slabiji uslov Q.

Sa ovako postavljenim sistemom, mogu se dokazati razna interesantna svojstva programa koji se razmatra.

Literatura

- Semantics with Applications: A Formal Introduction. Hanne Riis Nielson, Flemming Nielson. John Wiley & Sons, Inc. http://faculty.sist.shanghaitech.edu.cn/faculty/songfu/course/spring2018/CS131/sa.pdf

Proveravanje modela 8

Proveravanje modela je tehnika verifikacije zasnovana na sistematskom ispitivanju svih mogućih putanja u izvršavanju nekog sistema. Proveravanjem modela se ispituje da li sistem ispunjava neko zadato svojstvo (invarijantu, sigurnosno svojstvo, i sl.). Proveravanje modela se zasniva na preciznom formalnom opisu modela sistema, kao i svojstava koja se proveravaju. Oslanja se na matematičku logiku, teoriju formalnih jezika, teoriju grafova. Primenjiva na hardverske sisteme, konkurentne sisteme i komunikacione protokole, kao i na softver.

8.1 Osnovni pojmovi i primene 127
8.2 Pravljenje modela 131
8.3 Formalna specifikacija 138
8.4 Algoritmi za proveravanje modela . . 144

Značajni naučni radovi

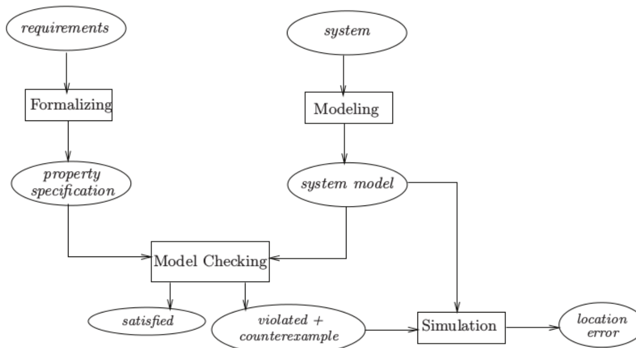
- Amir Pnueli: The Temporal Logic of Programs FOCS 1977: 46–57
- E.M.Clarke, E.A.Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic (1982)
- J.P.Queille. J. Sifakis. Specification and verification of concurent systems in CESAR (1982)

8.1 Osnovni pojmovi i primene

Osnovni pojmovi proveravanja modela obuhvataju pravljenje modela, definisanje specifikacije modela i proveravanje da li napravljeni model zadovoljava definisanu specifikaciju.

Pravljenje modela (modelovanje): proces formulisanja apstraktnog modela sistema na osnovu konkretne implementacije. Apstraktni model je tranzicioni sistem koji obuhvata skup stanja i relaciju prelaska nad tim skupom. Model sistema se obično automatski generiše iz programskog jezika ili iz jezika za opisivanje hardvera. Specifikacija opisuje šta sistem treba/ne treba da radi dok model adresira ponašanje sistema. Proveravanje modela ispituje sva relevantna stanja sistema kako bi se proverilo da li ona zadovoljavaju željena svojstva.

Disclaimer Svaka verifikacija koja koristi tehnike zasnovane na modelima je onoliko dobra koliko je dobar i model sistema!

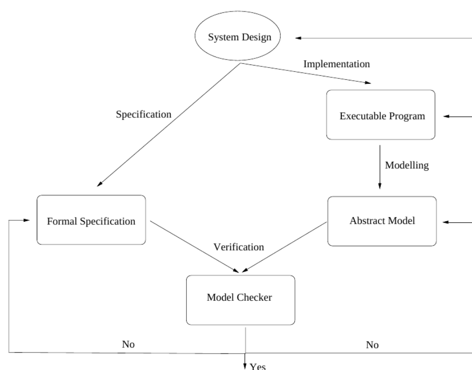


Slika 8.1: Strukturni prikaz — sistemi uopšte

Formalna specifikacija: precizan opis svojstava koje apstraktni model treba da zadovolji (na jeziku matematičke logike). Potrebno je opisati ponašanje sistema matematički precizno i nedvosmisleno — dešava se da i pre same verifikacije, tj već u fazi definisanja formalne specifikacije, otkriju nekompletnosti, dvosmislenosti i nekonzistentnosti neformalne specifikacije sistema.

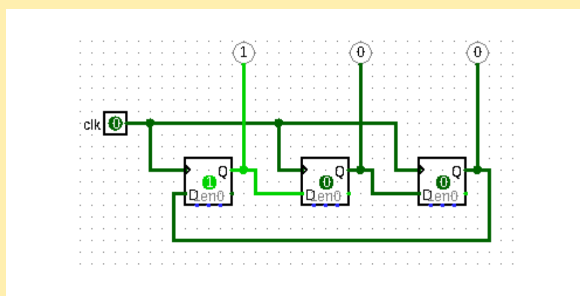
Primer 8.1.1 Formalizacija svih osobina podskupa ISDN (engl. *Integrated Services Digital Network*) korisničkog protokola (digitalna telefonska tehnologija nastala 70tih godina) je otkrila da je 55% originalnih neformalnih zahteva sistema bilo nekonzistentno!

Proveravač modela: alat koji automatski proverava da li apstraktni model zadovoljava formalnu specifikaciju. U slučaju da ne zadovoljava, proveravač daje kontraprimer u vidu putanje u modelu koja narušava zadato svojstvo. Kontraprimer se koristi za otkrivanje greške, korekciju modela i/ili specifikacije. Proveravač modela automatski istražuje sva moguća stanja sistema koristeći neki algoritam za proveravanje modela ili grubom silom. Broj mogućih stanja koja su dostupna ovom tehnikom je značajno povećan prethodnih godina i od 10^8–10^9 povećan je (korišćenjem pametnih algoritama i struktura podataka) na 10^{20} pa čak i u nekim primerima do 10^{476}!



Slika 8.2: Strukturni prikaz — softver

Primer 8.1.2 Hardver



Dat je trobitni ciklični registar sa početnom vrednošću 100. Interesuje nas svojstvo:
„U svakom trenutku je tačno jedan bit registra uključen".

Model

- Stanja:

$$\{p,q,r\} \in \{\{0,0,0\},\{0,0,1\},\{0,1,0\},\{0,1,1\},\{1,0,0\},\{1,0,1\},\{1,1,0\},\{1,1,1\}\}$$

- Relacija prelaska: $\{p,q,r\} \longrightarrow \{r,p,q\}$
- Početno stanje: 100
- Dostižna stanja: 100, 010, 001

Sva dostižna stanja zadovoljavaju traženo svojstvo.

Primer 8.1.3 Softver

```
1  while(true)
2  {
3    x = (x+1) % 3;
4    if(x == 0)
5     y = -y;
6    z = z*y + 1;
7  }
8
```

Dat je C-kod, sa početnim vrednostima:

$$x = 0, y = 1, z = 1$$

Interesuje nas svojstvo:
„Kada god je $y < 0$, u nekom od narednih koraka biće $y > 0$".

Model

- Stanja su određena vrednostima promenljivih i trenutnom pozicijom u kodu.
- Početno stanje je određeno početnim vrednostima promenljivih i ulaznom tačkom programa.
- Relacija prelaska je određena semantikom naredbi programa.
- Zadato svojstvo nije vezano za pojedinačno stanje, već za odnose među stanjima.

8.1.1 Primene proveravanja modela

Primene proveravanja modela su raznovrsne. Pre svega, proveravanje modela je nastalo u kontekstu verifikacije hardvera i konkurentnih sistema, ali se danas koristi i za verifikaciju softvera.

Tjuringova nagrada 2007. E.M.Clarke, E.A.Emerson i J. Sifakis su dobitnici Tjuringove nagrade zbog svojih doprinosa zasnivanju i razvoju proveravanja modela — veoma efikasne verifikacijske tehnike koja je široko prihvaćena u industriji hardvera i softvera.

Verifikacija hardvera

Proizvodnja novog hardvera je izuzetno skup proces. Popravljanje grešaka u hardveru nakon isporuke ima veoma visoke troškove. Greške u softveru se mogu ispravljati sa zakrpama i update-ovima, dok ispravljanje grešaka u hardveru zahteva ponovnu proizvodnju i ponovnu isporuku prozvoda.

Primer 8.1.4 Zamena Intelovog Pentium II procesora (problematična implementacija operacije deljenja u pokretnom zarezu) je uzrokovala gubitak od 475 miliona dolara (1994. godine).

Zbog toga, **dizajn čipa obično iznosi manje od trećine ukupnog napora i ulaganja u razvoj, dok se ostatak posvećuje detekciji i prevenciji grešaka.** Ali, nekada ni to nije dovoljno.

Primer 8.1.5 19. avgusta **2016**. godine pušten je u prodaju *Samsung Galaxy Note 7*. Telefon je suspendovan 2. septembra (u Americi), a 15. septembra povučen iz prodaje jer je litijumska baterija povremeno mogla da razvije visoku temperaturu i da se zapali. Najpre je pokušano sa zamenama baterija, ali kako to nije pomoglo, 10. oktobra je obustavljena proizvodnja i aparat je povučen u potpunosti iz prodaje (u celom svetu). **Procena štete: 17 milijardi dolara!**

Zanimljivost: Korisnicima uređaja *Samsung Galaxy Note 7* na svim aerodromima je bio zabranjen ulazak u avion sa telefonom. Samsung je na nekim aerodromima bio omogućio preuzimanje i zamenu telefona pre leta.

Standardne tehnike ispitivanja ispravnosti hardvera obuhvataju emulaciju, simulaciju i fabričko testiranje.

Emulacija koristi generički hardver (emulator) koji se konfiguriše tako da se ponaša kao kolo koje se razmatra i onda se tako konfigursan hardver intenzivno testira (tj proverava se da li se ponaša u skladu sa specifikacijom čipa). Kod simulacije, model kola koji se razmatra se konstruiše koristeći neki od jezika za opisivanje hardvera (npr Verilog ili VHDL) i onda se taj model testira.

Testiranje i kod emulacije i kod simulacije može da bude sa nekim određenim ulazima ili sa automatski generisanim ulazima, npr sa random generisanim ulazima. Testiranje, kao i kod softvera, ima preveliki broj mogućih ulaza da bi

ponašanje hardvera moglo da se u potpunosti istraži na taj način.

Fabričko testiranje odgovara traženju grešaka koje nastaju u procesu proizvodnje.

Emulacija, simulacija i fabričko testiranje su tehnike testiranja hardvera i kao takve služe za pronalaženje grešaka u hardveru, ne i za utvrđivanje da li je hradver ispravan.

Verifikacija konkurentnih sistema

Konkurentni sistemi su svuda. Grešake u konkurentnim sistemima se mogu ispoljavati nedeterministički. To znači da sa istim test primerom, sistem nekada može da se ponaša ispravno a nekada neispravno. Razumevanje konkurentnih grešaka je često veoma teško. Posebno su bitne greške mrtvih/živih petlji (deadlock/livelock) i izgladnjivanja (lockout).

Verifikacija softvera

8.2 Pravljenje modela

Kripke struktura je vrsta tranzicionog sistema koji je originalno razvio Saul Kripke a koja se koristi u proveravanju modela za predstavljanje ponašanja sistema. Neformalno, to je graf čiji su čvorovi dostupna stanja sistema a grane predstavljaju prelaske između stanja sistema. Funkcija obeležavanja (labeliranja, mapiranja) je funkcija koja preslikava svaki čvor u skup osobina koje važe u odgovarajućem stanju. Originalno, Kripke struktura je uređena četvorka.

Dodati kripke strukture i dodati primere — trenutno je to samo u slajdovima.

8.2.1 Tranzicioni sistem

Tranzicioni sistem je uređena šestorka

$$(S, Act, \longrightarrow, I, \mathcal{V}, \lambda)$$

gde je:

- S skup stanja
- Act skup akcija $\{\alpha_0, \alpha_1, \ldots, \alpha_n\}$
- $\longrightarrow \subseteq S \times Act \times S$ relacija prelaska (u oznaci $s_i \xrightarrow{\alpha} s_j$,a ukoliko akcija nije bitna, onda se oznaka α izostavlja)

- $I \subseteq S$ skup početnih stanja
- $\mathcal{V}$ skup predikata
- $\lambda : S \to \mathbb{P}\mathcal{V}$ funkcija obeležavanja: svakom stanju pridružuje skup predikata koji važe u tom stanju.

Napomena: ukoliko akcije nisu bitne, onda se one ne navode, a tranzicioni sistem se u tom slučaju posmatra kao uređena petorka $(S, \longrightarrow, I, \mathcal{V}, \lambda)$.

Skup stanja može biti konačan ili beskonačan — tipično je konačan, ali veoma veliki. Bez ograničenja opštosti, može se pretpostaviti da je relacija $\longrightarrow$ totalna:

$$(\forall s \in S)(\exists s' \in S)(s \longrightarrow s')$$

(uslov koji se zahteva u definiciji Kripke struktura). Sistem je deterministički (po akcijama) ako važi

$$(\forall s \in S)(\forall \alpha \in Act)(\exists! s' \in S)(s \xrightarrow{\alpha} s')$$

i nedeterministički. Kod determinističkih sistema, podrazumeva se tačno jedno početno stanje.

Putanja u sistemu $T = (S, Act, \longrightarrow, I, \mathcal{V}, \lambda)$ je beskonačni niz koji alternira stanja i akcije

$$\sigma = s_0 \alpha_0 s_1 \alpha_1 s_2 \ldots$$

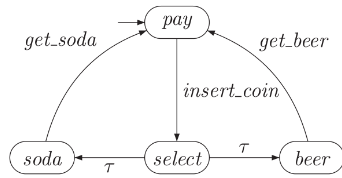
takav da važi $s_i \xrightarrow{\alpha_i} s_{i+1}$ za svako $i \geq 0$.

Koriste se naredne oznake:

- $\sigma_i := s_i$ za i-to stanje putanje i
- $\sigma_{|i} := s_i \alpha_i s_{i+1} \alpha_{i+1} s_{i+2} \ldots$ za „rep" putanje počev od i-tog stanja.

Izvršavanje u sistemu T je bilo koja putanja σ takva da je $\sigma_0 \in I$, tj da početno stanje putanje pripada skupu početnih stanja.
Za stanje $s \in S$ kažemo da je **dostižno** ako postoji izvršavanje σ takvo da $s \in \sigma$.

Primer 8.2.1 Mašina za piće



Tranzicioni sistem za mašinu za piće

- Skup stanja:

$$S = \{pay, select, soda, beer\}$$

- Početno stanje:

$$I = \{pay\}$$

- Skup akcija:

$$Act = \{insert_coin, get_soda, get_beer, \tau\}$$

  Akcija τ predstavlja nedeterministički izbor. U ovom slučaju, mašina nedeterministički može da da sodu odnosno pivo.
- Prelazi u sistemu su:

$$\begin{array}{rcl} pay & \xrightarrow{insert_coin} & select \\ select & \xrightarrow{\tau} & soda \\ select & \xrightarrow{\tau} & beer \\ soda & \xrightarrow{get_soda} & pay \\ beer & \xrightarrow{get_beer} & pay \end{array}$$

- $\mathcal{V}$ zavisi od svojstava koja razmatramo.
  Na primer, ako razmatramo svojstvo
  „Mašina daje piće samo nakon što dobije novčić"
  onda su nam bitni predikati

$$\mathcal{V} = \{paid, drink\}$$

  U kontekstu razmatranog svojstva, važno je da ne postoji stanje u kojem je piće dato, a nije plaćeno (odnosno stanje u kojem važi samo predikat $drink$ a ne važi predikat $paid$.
- Razmotrimo stanja i predikate koji važe u tim stanji-

ma:

$$\begin{aligned} \lambda(pay) &= \emptyset \\ \lambda(select) &= \{paid\} \\ \lambda(soda) &= \{paid, drink\} \\ \lambda(beer) &= \{paid, drink\} \end{aligned}$$

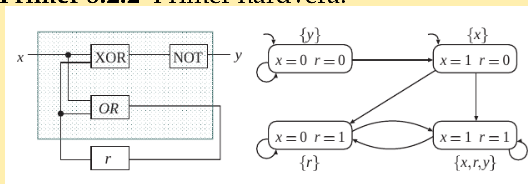
Na osnovu pregleda svih stanja i predikata koji u tim stanjima važe, vidimo da je u sistemu ispunjeno željeno svojstvo.

Primetimo da u okviru ovog sistema može da se desi da neko plati piće ali da ga ne dobije, tj postoji stanje u kojem je zadovoljen predikat $paid$ ali nije istovremeno zadovoljen i predikat $drink$. To je stanje $select$.

8.2.2 Modelovanje hardvera

Stanja su određena stanjem registara u datom hardverskom sklopu i vrednostima bitova na ulazu. Ulazi dolaze iz spoljašnjeg okruženja (druge komponente, interakcija sa korisnikom), i njihove vrednosti se ne mogu predvideti (nedeterminizam). Relacija prelaska opisuje ponašanje kola u svakom ciklusu časovnika. Tipično imamo veliki broj stanja (eksponencijalan u odnosu na broj bitova registara i bitova ulaza). Skup $\mathcal{V}$ obično odgovara bitovima ulaza, bitovima registara, kao i bitovima izlaza. $\lambda(s)$ je tada skup svih bitova koji imaju vrednost 1 u stanju s.

Primer 8.2.2 Primer hardvera.



Razmatramo jednobitni hardver: x – ulaz, y – izlaz, r – registar. Izlaz se računa po funkciji $\neg(x \oplus r)$, a registar kao $x \vee r$. Ulaz x uvodi nedeterminizam.

- Stanje: par bitova $\{x, r\}$
- Početna stanja: $\{x = 0, r = 0\}$ i $\{x = 1, r = 0\}$
- Graf relacije prelaska je prikazan na desnoj slici:

- Za početno stanje $\{x = 0, y = 1, r = 0\}$ (gore levo)
  - * Ukoliko se na ulazu ponovo nađe 0, onda stanje ostaje nepromenjeno
  - * Ukoliko se na ulazu nađe 1, onda se prelazi u stanje u kojem je $\{x = 1, y = 0, r = 0\}$ (gore desno)
- Za početno stanje $\{x = 1, y = 0, r = 0\}$ (gore desno)
  - * Ukoliko se na ulazu ponovo nađe 1, onda stanje prelazi u stanje u kojem je $\{x = 1, y = 1, r = 1\}$ (dole desno). U ovom stanju sistem ostaje sve dok je na ulazu $x = 1$.
  - * Ukoliko se na ulazu nađe 0, onda se prelazi u stanje u kojem je $\{x = 0, y = 0, r = 1\}$ (dole levo). U ovom stanju sistem ostaje sve dok je na ulazu $x = 0$.
- Prelasci između stanja $\{x = 1, y = 1, r = 1\}$ i $\{x = 0, y = 0, r = 1\}$ se ostvaruju u zavisnosti od ulazne vrednosti x.

▶ Predikati koje pratimo su $\mathscr{V} = \{x, y, r\}$
▶ Ispunjeni predikati u svakom stanju grafa $\lambda(s)$ su prikazane uz svako stanje u grafu.

8.2.3 Modelovanje softvera

Svaki program P se sastoji iz:

(1) konačnog skupa promenljivih $Var(P) = x_1, \ldots, x_n$, pri čemu promenljiva x_i uzima vrednosti iz skupa $Dom(x_i)$
(2) konačnog skupa lokacija $Loc(P) = L_0, L_1, \ldots, L_m$, pri čemu je L_0 početna lokacija (tj. ulazna tačka programa)
(3) skupa uslova $Cond(P)$ nad $Var(P)$
(4) skupa preduslova $Pred(P) \subset Cond(P)$ koji moraju biti zadovoljeni na početku izvršavanja programa
(5) skupa akcija $Act(P)$ nad $Var(P)$
(6) relacije toka programa koja se sastoji od pojedinačnih prelazaka iz stanja u stanje

$$\rightarrow \subseteq Loc(P) \times (Cond(P) \cup Act(P)) \times Loc(P)$$

Primer 8.2.3 Skup uslova može da sadrži, na primer, naredne uslove $x_1 < x_2$ i $x_1 > 0$.

Primer 8.2.4 Skup akcija može da sadrži, na primer, naredne akcije $x_1 \leftarrow x_1 + 1$ i $x_2 \leftarrow x_1 \cdot x_1$.

Primer 8.2.5 Relacija toka može da sadrži naredni prelazak

$$L_0 \xrightarrow{x_1 \leftarrow x_1 + 1} L_1$$

On označava da se na lokaciji L_0 izvršava akcija $x_1 \leftarrow x_1 + 1$ kojom se menja vrednost promenljive x_1 i prelazi se na lokaciju L_1.

Primer 8.2.6 Relacija toka može da sadrži naredni prelazak

$$L_1 \xrightarrow{x_1 < x_2} L_2$$

On označava da se na lokaciji L_1 ispituje uslov $x_1 < x_2$ i ako je ispunjen, prelazi se na lokaciju L_2.

Primer 8.2.7 (Nastavak primera 8.1.3.)

```
1       while(true)
2       {
3   L0:   x = (x+1) % 3;
4   L1:   if(x == 0)
5   L2:    y = -y;
6   L3:   z = z*y + 1;
7       }
8
```

Formalni opis programa:

(1) Skup promenljivih i njihovi domeni
$Var(P) = \{x, y, z\}$,
$Dom(x) = Dom(y) = Dom(z) = Int$

(2) Skup lokacija
$Loc(P) = \{L_0, L_1, L_2, L_3\}$

(3) Skup uslova $Cond(P)$ koje razmatramo zavisi od svojstva koje ćemo pratiti i želeti da dokažemo.

(4) Preduslov
$Pred(P) = \{x = 0, y = 1, z = 1\}$

(5) Skup akcija
$x \leftarrow (x+1)\%3$
$y \leftarrow -y$
$z \leftarrow z * y + 1$

(6) Relacija prelaska:
$L_0 \xrightarrow{x \leftarrow (x+1)\%3} L_1$
$L_1 \xrightarrow{x=0} L_2$
$L_1 \xrightarrow{x \neq 0} L_3$
$L_2 \xrightarrow{y \leftarrow -y} L_3$
$L_3 \xrightarrow{z \leftarrow z \cdot y+1} L_0$

Izvršavanje programa P se može formalno predstaviti sledećim tranzicionim sistemom:

(1) skup stanja je $S = Loc(P) \times Val(P)$ gde je

$$Val(P) = Dom(x_1) \times Dom(x_2) \ldots \times Dom(x_n)$$

skup valuacija promenljivih $x_1, \ldots, x_n$ iz $Var(P)$

(2) skup početnih stanja

$$I = \{L_0\} \times PredVal(P)$$

gde je $PredVal(P)$ skup svih valuacija iz $Val(P)$ koje zadovoljavaju sve preduslove iz $Pred(P)$

(3) relacija prelaska je definisana na sledeći način:

- ako važi $L_i \xrightarrow{a} L_j$, gde je $a \in Act(P)$, tada za svaku valuaciju $\eta \in Val(P)$ postoji prelaz $(L_i, \eta) \longrightarrow (L_j, \eta[a])$, pri čemu je $\eta[a]$ valuacija koja nastaje primenom akcije a na valuaciju η
- ako važi $L_i \xrightarrow{c} L_j$, gde je $c \in Cond(P)$, tada za svaku valuaciju $\eta \in Val(P)$ postoji prelaz $(L_i, \eta) \longrightarrow (L_j, \eta)$ akko valuacija η zadovoljava uslov c

(4) skup predikata $\mathcal{V}$ je bilo koji konačan podskup skupa $Cond(P)$

(5) obeležavanje $\lambda(L_i, \eta)$ sadrži uslove iz $\mathcal{V}$ koji su zadovoljeni u valuaciji η

Primer 8.2.8 Tranzicioni sistem za primer 8.1.3.

```
1      while(true)
2      {
3  L0:   x = (x+1) % 3;
4  L1:   if(x == 0)
5  L2:    y = -y;
6  L3:   z = z*y + 1;
7      }
8
```

(1) Stanja: $(L_i, [v_x, v_y, v_z])$, gde je L_i lokacija u programu, a $[v_x, v_y, v_z]$ valuacija promenljivih x, y, z

(2) Početno stanje: $(L_0, [0, 1, 1])$

(3) Relacija prelaska:

$$
\begin{array}{lcl}
(L_0, [v_x, v_y, v_z]) & \xrightarrow{x=(x+1)\%3} & (L_1, [(v_x+1)\%3, v_y, v_z]) \\
(L_1, [0, v_y, v_z]) & \xrightarrow{x=0} & (L_2, [0, v_y, v_z]) \\
(L_1, [v_x, v_y, v_z]) & \xrightarrow{x\neq 0} & (L_3, [v_x, v_y, v_z]) \\
(L_2, [v_x, v_y, v_z]) & \xrightarrow{y=-y} & (L_3, [v_x, -v_y, v_z]) \\
(L3, [v_x, v_y, v_z]) & \xrightarrow{z=z\cdot y+1} & (L_0, [v_x, v_y, v_z \cdot v_y + 1])
\end{array}
$$

(Primetimo da je ovo deterministički tranzicioni sistem.)

(4) Skup predikata je npr. $\mathcal{V} = \{y > 0, |z| < 3\}$

(5) Obeležavanje je, na primer,

$$\lambda(L_2, [1, -1, -2]) = \{|z| < 3\}$$

Pri modelovanju softvera, broj stanja je veoma veliki (npr. samo jedna `int` promenljiva u C-u ima preko četiri milijarde mogućih stanja. Postoje različiti načini da se izborimo sa tim problemom (npr. apstrakcija predikata). Softver se takođe

može modelovati nedeterminističkim tranzicionim sistemom (tipično za konkurentne aplikacije, operativne sisteme i sl.)

8.3 Formalna specifikacija

Formalnom specifikacijom zadaju se svojsta koja će biti proveravana.

8.3.1 Tipovi svojstava

Postoje različiti tipovi svojstava koji se mogu proveravati. Najjednostavnija svojstva su *svojstva stanja* (eng. *state property*). Ova svojstva izražavaju uslov koji se odnosi na pojedinačno stanje. Mogu se izraziti iskaznim formulama nad predikatima tranzicionog sistema. Nisu preterano zanimljiva sama za sebe (jer ne uzimaju u obzir relaciju prelaska) već kao gradivni element pri formulisanju složenih svojstava.

Primer 8.3.1 Za neko stanje, možemo da razmatramo naredno svojstvo stanja $x > 0 \wedge ((z + y) \bmod 2 = 0)$

Neka interesantna složena svojstva koja ćemo razmatrati su

- invarijante,
- sigurnosna svojstva,
- svojstva živosti i
- svojstva pravednosti.

Invarijante

Primer 8.3.2 Primer invarijante: „u svim dostižnim stanjima je $x \neq 0$".

Invarijanta je svojstvo koje treba da važi u svim dostižnim stanjima tranzicionog sistema.

Primer 8.3.3 Invarijanta „u svim dostižnim stanjima je $x \neq 0$" je istovremeno i sigurnosno svojstvo „nikada se neće desiti da je $x = 0$".

Sigurnosna svojstva

Primer 8.3.4 Sigurnosno svojstvo koje nije invarijanta „kada x postane 1, tada x mora ostati 1 dokle god je $y = 0$".

Sigurnosno svojstvo (eng. *safety property*) je svojstvo koje izražava da se neka negativna pojava nikada neće desiti. Svaka invarijanta je i sigurnosno svojstvo ali postoje sigurnosna svojstva koja nisu invarijante. Sigurnosno svojstvo je narušeno u tranzicionom sistemu ako postoji loš prefiks, tj. izvršavanje σ takvo da za neki njegov konačni prefiks ne važi dato svojstvo. To znači da je putanja greške konačna.

Primer 8.3.5 Svojstvo iz primera 8.3.4 je narušeno ako imamo prefiks $\sigma_0\sigma_1 \ldots \sigma_i\sigma_{i+1}$ izvršavanja σ takav da u stanju σ_i važi $x = 1, y = 0$, a u stanju σ_{i+1} važi $x = 0, y = 0$.

Svojstva živosti

Svojstvo živosti (eng. *liveness property*) je svojstvo koje izražava da će se neka pozitivna pojava sigurno desiti u budućnosti. Ovakva svojstva koristimo za izražavanje progresa u sistemu (npr. odsustvo uzajamnog blokiranja). Ovakvo svojstvo ne može biti narušeno lošim prefiksom (za razliku od sigurnosnog svojstva), jer ako nešto ne važi do nekog trenutka, možda će važiti u budućnosti. To znači da je putanja greške beskonačna.

Primer 8.3.6 Svojstvo živosti „kada x postane 1, sigurno će u nekom narednom trenutku i y postati 1".

Svojstva pravednosti

Svojstvo pravednosti (eng. *fairness property*) je svojstvo koje izražava da će se neka pojava dešavati beskonačno puta tokom izvršavanja. Ovakva svojstva koristimo da izrazimo da u sistemu nema „izgladnjivanja" (npr. da će svaki proces na nekom operativnom sistemu kontinuirano dobijati priliku da se izvršava).

Primer 8.3.7 Svojstvo pravednosti „ako x postaje 1 beskonačno puta tokom izvršavanja, tada će i y postajati 1 beskonačno puta tokom izvršavanja".

8.3.2 Temporalne logike

Temporalna logike su logike koje omogućavaju predstavljanje tvrđenja i rasuđivanje o tvrđenjima koja su kvalifikovana vremenskim odrednicama.

Primer 8.3.8 U temporalnoj logici možemo da izrazimo tvrđenja kao što su

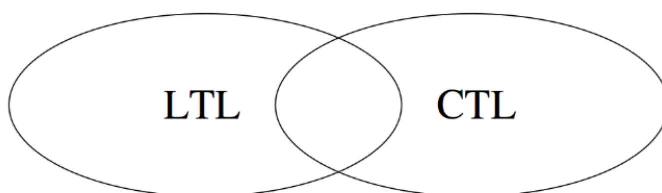
- „Ja sam **uvek** gladan."
- „Ja ću **u nekom trenutku** biti gladan."
- „Ja ću biti gladan **sve dok** ne pojedem nešto."

Složena svojstva koja se koriste u verifikaciji softvera mogu se formalno izraziti u terminima temporalnih logika. U temporalnim logikama srećemo različit izbor dodatnih operatora, i samim tim one opisuju različite modele vremena. Dakle, ne postoji jedinstvena temporalna logika, već čitav niz različitih logičkih sistema koji, svaki na svoj način, opisuju fenomene koji su povezani sa vremenom. Najčešći operatori koji se dodaju su operatori

- G (eng. ***G**lobaly, always*) — globalno, uvek,
- F (eng. ***F**uture, eventually*) — u nekom trenutku u budućnosti,
- X (eng. *ne**X**t*) — u sledećem trenutku u budućnosti,
- U (eng. ***U**ntil*) — neko svojstvo će važiti dok se ne ispuni drugo svojstvo i
- W (eng. ***W**eak until*) — neko svojstvo će važiti zauvek ili dok se ne ispuni drugo svojstvo.

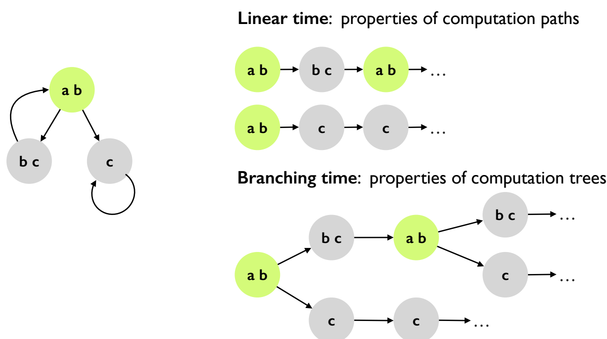
Posebno važne temporalne logike za proveravanje modela su:

- LTL — linearna temporalna logika (eng. *linear temporal logic*)
- CTL — logika stabla izračunavanja (eng. *computational tree logic*)
- CTL* je vrsta logike stabla izračunavanja koja slobodno kombinuje temporalne operatore i kvantifikatore putanja.



Slika 8.3: Odnos temporalnih logika

Odnos pomenutih temporalnih logika je prikazan na slici 8.3. CTL* je nadskup od LTL i CTL. Postoje tvrđenja koja se mogu iskazati u LTL logici a ne mogu u CTL. Postoje tvrđenja koja se mogu iskazati u CTL logici a ne mogu u LTL. Postoje tvrđenja koja se mogu iskazati u obe logike. Praktična važna razlika je u algoritmima odlučivanja — najveća složenost je za CTL*, najefikasniji algoritam odlučivanja je za CTL, a najintuitivnije za razumevanje je modelovanje u LTL.



Slika 8.4: LTL i CTL

8.3.3 Linearna temporalna logika

Formulama temporalne logike zadajemo svojstva od interesa za tranzicioni sistem za koji proveravamo ispravnost. Zbog toga se LTL formula definiše u kontekstu tranzicionog sistema.

Tjuringova nagrada 1996. A. Pnueli je dobitnik Tjuringove nagrade zbog uvođenja temporalne logike u računarstvo i za izuzetne doprinose verifikaciji programa i sistema.

Sintaksa LTL formula nad tranzicionim sistemom

$$T = (S, Act, \longrightarrow, I, \mathcal{V}, \lambda)$$

je zadata rekurzivno:

- logičke konstante $\top$ i $\bot$ su LTL formule,
- atomičke formule $p \in \mathcal{V}$ su LTL formule,
- ako je ϕ LTL formula onda je i
$$\neg\phi$$
LTL formula,
- ako su ϕ_1 i ϕ_2 LTL formule, onda su i
$$\phi_1 \wedge \phi_2, \phi_1 \vee \phi_2, \phi_1 \Rightarrow \phi_2$$
LTL formule,
- ako su ϕ_1 i ϕ_2 LTL formule, onda su i
$$\phi_1 U \phi_2 \text{ i } \phi_1 W \phi_2$$
LTL formule,
- ako je ϕ LTL formula, onda su i
$$X\phi, F\phi \text{ i } G\phi$$
LTL formule.

Semantika LTL formule se takođe zadaje u kontekstu tranzicionog sistema. Neka je data proizvoljna putanja σ nad T. Formula ϕ je tačna na σ se označava kao $\sigma \models \phi$ i važi:

- za logičke konstante: $\sigma \models \top$, $\sigma \not\models \bot$
- za atomičke formule p: $\sigma \models p$ ako $p \in \lambda(\sigma_0)$, tj ako je p tačno u početnom stanju putanje
- $\sigma \models \neg\phi$ ako je ϕ LTL formula za koju ne važi $\sigma \models \phi$ (tj važi $\sigma \not\models \phi$),
- $\sigma \models \phi_1 \wedge \phi_2$ ako za LTL formule ϕ_1 i ϕ_2 važi $\sigma \models \phi_1$ i $\sigma \models \phi_2$,
- $\sigma \models \phi_1 \vee \phi_2$ ako za LTL formule ϕ_1 i ϕ_2 važi $\sigma \models \phi_1$ ili $\sigma \models \phi_2$,
- $\sigma \models \phi_1 \Rightarrow \phi_2$ ako za LTL formule ϕ_1 i ϕ_2 važi da $\sigma \models \phi_1$ povlači $\sigma \models \phi_2$,

- $\sigma \models X\phi$ ako $\sigma_{|1} \models \phi$ (intuitivno: formula ϕ važi ako se posmatra putanja bez nultog stanja, dakle ϕ važi počevši od prvog narednog stanja),
- $\sigma \models \phi_1 U \phi_2$ ako $(\exists j)(\sigma_{|j} \models \phi_2 \land (\forall i < j)(\sigma_{|i} \models \phi_1))$ (intuitivno: postoji neko stanje j tako da u tom stanju važi ϕ_2 a da je pritom u svakom prethodnom stanju važilo ϕ_1 — primetio da u stanju j formula ϕ_1 može i ne mora da važi, to definicijom nije precizirano)
- $F\phi$ je ekvivalentno sa $\top U \phi$ (intuitivno: nekada u budućnosti važiće ϕ)
- $G\phi$ je ekvivalentno sa $\neg F \neg \phi$ (intuitivno: nikada u budućnosti neće važiti $\neg\phi$, dakle uvek važi ϕ)
- $\phi_1 W \phi_2$ je ekvivalentno sa $(\phi_1 U \phi_2) \lor G\phi_1$

Primetimo da se semantika operatora U, W, F, G i X definiše rekurzivno u odnosu na odgovarajuće repove putanje σ (rep putanje je takođe putanja). Takođe, operatori F, G i W su definisani na osnovu osnovnih operatora X i U i oni su zapravo uvedeni radi lakšeg iskazivanja željenih svojstava.

Intuitivno:

neXt $X\phi$ znači da ϕ važi u sledećem stanju putanje σ
Until $\phi_1 U \phi_2$ znači da ϕ_1 važi duž putanje σ dokle god ϕ_2 ne postane tačno („ϕ_1 dok se ne ispuni ϕ_2")
Future $F\phi$ znači da će ϕ sigurno postati tačno u nekom od narednih stanja putanje („pre ili kasnije ϕ")
Globally $G\phi$ znači da će ϕ važiti duž cele putanje („zauvek ϕ")
Weak until $\phi_1 W \phi_2$ je slično sa U, s tim što ϕ_2 nikada ne mora da postane tačno, u kom slučaju će ϕ_1 važiti zauvek, duž cele putanje („ϕ_1 dok se eventualno ne ispuni ϕ_2")

Intuitivno značenje za neke česte kombinacije operatora:

$XG\phi$ — „važi ϕ počev od sledećeg stanja zauvek"
$GF\phi$ — „ϕ važi u beskonačno mnogo stanja duž putanje" (za svako stanje važi da postoji stanje u budućnosti na kojem važi ϕ)
$FG\phi$ — „od nekog trenutka u budućnosti, zauvek će važiti važiti ϕ"
$G(\phi_1 \Rightarrow F\phi_2)$ — „kad god važi ϕ_1 u nekom stanju, tada mora da važi ϕ_2 u nekom od narednih stanja"
$G(\phi \Rightarrow X\neg\phi)$ — „kad god važi ϕ u nekom stanju, tada ne važi u sledećem"

Formula ϕ **je zadovoljiva** u tranzicionom sistemu T ako postoji izvršavanje σ (tj. putanja koja kreće iz nekog početnog stanja), takvo da je $\sigma \models \phi$. Formula ϕ **je valjana** u tranzicionom sistemu T ako za svako izvršavanje σ važi da je $\sigma \models \phi$.

Svojstva na LTL jeziku

LTL omogućava formulisanje većine svojstava koja se tipično javljaju u verifikaciji zasnovanoj na proveravanju modela. Svojstva se formulišu kao LTL formule tako da sistem koji se verifikuje zadovoljava dato svojstvo akko je odgovarajuća formula valjana u tranzicionom sistemu T koji je model početnog sistema.

Invarijante se izražavaju formulama oblika $G\phi$, gde je ϕ iskazna formula.

Sigurnosna svojstva takođe imaju G kao vodeći veznik, ali podformula može sadržati i druge ne-iskazne veznike.

Svojstva živosti podrazumevaju da će neki događaji uvek alternirati.

Svojstva pravednosti podrazumevaju da će se neki događaji dešavati uvek beskonačno mnogo puta.

Primer svojstva koje se ne može izraziti na LTL jeziku je svako svojstvo koje uključuje vremensko rezonovanje o svim putanjama. Na primer,

- „Iz svakog dostižnog stanja se može stići u neko početno stanje".
- „Počevši od nekog trenutka, sve putanje će imati svojstvo ϕ."

Primer 8.3.9 (Invarijanta)

$$G(x = 0 \lor y = 0)$$

„u svim dostižnim stanjima je x ili y jednako nula"

Primer 8.3.10 (Sigurnosno svojstvo)

$$G(x = 1 \Rightarrow (x = 1\ W\ y \neq 0))$$

„kad x postane 1, ostaće 1 dokle god je y jednako 0"

Primer 8.3.11 (Svojstvo živosti)

$$G(x = 1 \Rightarrow F(y = 1))$$

„kad god x postane 1, sigurno će i y u nekom narednom trenutku postati 1".

Primer 8.3.12 (Svojstvo pravednosti)

$$G(F(x = 1)) \Rightarrow G(F(y = 1))$$

„ako x postaje 1 beskonačno često, tada i y mora postati 1 beskonačno često".

8.3.4 CTL* i CTL

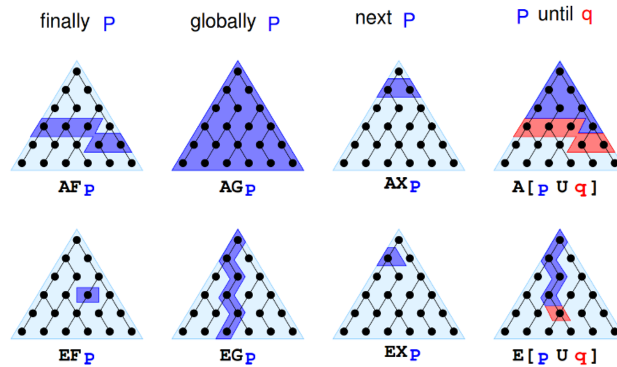
U okviru logika CTL* i CTL uvode se operatori nad putanjama A i E. Intuitivno, semantika ovih operatora je:

A ϕ — formula ϕ važi na svim putanjama počevši od trenutnog stanja,

E ϕ — postoji najmanje jedna putanja počevši od trenutnog stanja tako da na njoj važi formula ϕ.

U CTL*, temporalni opeartori mogu da se mešaju proizvoljnim redosledom. U CTL-u, ovi operatori se ne mogu koristiti u proizvoljnom poretku, već isključivo u kombinaciji sa operatorima nad stanjima (X, G, F, U, W), tj uvek moraju da budu u grupama od dva: jedan operator nad putanjom za kojim sledi operator nad stanjem.

Intuitivno značenje CTL operatora je prikazana na slici 8.5.



Slika 8.5: Intuitivno značenje CTL operatora

8.4 Algoritmi za proveravanje modela

Invarijanta je svojstvo koje treba da važi u svim dostižnim stanjima. Jednostavan način da se ovo proveri je obilazak grafa tranzicionog sistema (polazeći iz početnih stanja). Najjednostavniji algoritmi za obilazak grafova obuhvataju:

- Obilazak u dubinu: omogućava jednostavno pronalaženje kontraprimera (kada naiđemo na stanje u kome ne važi invarijanta, na steku se nalaze svi njegovi prethodnici na toj putanji)
- Obilazak u širinu: omogućava pronalaženje najkraćeg puta do nekog stanja koje narušava invarijantu (za rekonstrukciju celog puta neophodno je čuvati dodatne informacije tokom pretrage)

Međutim, u slučaju prevelikog broja stanja graf postaje nemoguće obići u potpunosti. Takođe, obilazak grafa na ovaj način ne može da obezbedi proveru važenja složenih svojstava.

8.4.1 Bihijevi automati

Bihijevi automati se koriste za verifikaciju opštih LTL svojstava (ne samo invarijanti). Zasnivaju se na uopštenju teorije formalnih jezika. Izvršavanja u tranzicionom sistemu se posmatraju kao beskonačne reči nad azbukom stanja.

Julias Richard Büchi (1924-1984): švajcarski matematičar

Regularnim jezicima odgovaraju konačni automati dok konteksno slobodnim jezicima odgovaraju potisni automati.

Neka je data konačna azbuka Σ. ω-reč nad azbukom Σ je bilo koji beskonačni niz $w = a_0a_1a_2\dots$ slova azbuke Σ. Skup svih ω-reči označavamo sa Σ^ω. ω-jezik nad Σ je bilo koji podskup skupa Σ^ω.

ω-jezik je ω-regularan ako se dobija konačnom primenom sledećih pravila:

- Ako je L neprazan regularan jezik koji ne sadrži ϵ, tada je $L^\omega = \{w_1w_2\dots \mid w_i \in L\}$ ω-regularan
- Ako je L_1 regularan jezik, a L_2 ω-regularan, tada je i L_1L_2 ω-regularan
- Ako su L_1 i L_2 ω-regularni, tada je i $L_1 \cup L_2$ ω-regularan

Primer 8.4.1 Primer ω-regularnog jezika

$$(0|1)^*(10)^\omega \mid 01^\omega$$

skup svih beskonačnih reči koje imaju „rep” oblika $1010\dots$ ili reč $0111\dots$

Bihijev automat je uređena petorka

$$\mathcal{A} = (Q, \Sigma, Q_0, \delta, F)$$

gde je:

- Q konačan skup stanja
- Σ konačna azbuka
- $Q_0 \subseteq Q$ skup početnih stanja
- $F \subseteq Q$ skup završnih stanja
- $\delta : Q \times \Sigma \longrightarrow \mathbb{P}Q$ funkcija prelaska

Izračunavanje u Bihijevom automatu je bilo koje beskonačno izvođenje oblika $q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} q_3\dots$, pri čemu je $q_0 \in Q_0$ i $q_{i+1} \in \delta(q_i, a_{i+1})$ za svako $i = 0, 1, 2, \dots$. Etiketa tog izračunavanja je ω-reč $a_1a_2a_3\dots$. Za izračunavanje kažemo da je uspešno ako sadrži beskonačno mnogo završnih stanja. ω-jezik $L(\mathcal{A})$ definisan automatom $\mathcal{A}$ je skup svih ω-reči koje su etikete uspešnih izračunavanja automata $\mathcal{A}$.

Theorem 8.4.1 *ω-jezik L je ω-regularan akko je prepoznat nekim (u opštem slučaju nedeterminističkim) Bihijevim automatom.*

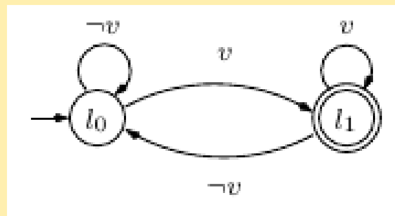
LTL i Bihijevi automati

Za svaki tranzicioni sistem $T = (S, \longrightarrow, I, \mathcal{V}, \lambda)$, svaka njegova putanja $\sigma = s_0 s_1 s_2 \ldots$ je jedna ω-reč nad azbukom stanja S. Proizvoljna LTL formula definiše skup putanja koje je zadovoljavaju, tj. definiše jedan ω-jezik nad azbukom S. Može se pokazati da je svaki tako definisan jezik ω-regularan, pa se može prepoznati Bihijevim automatom.

Primer 8.4.2 Bihijev automat za formulu

$$GFv$$

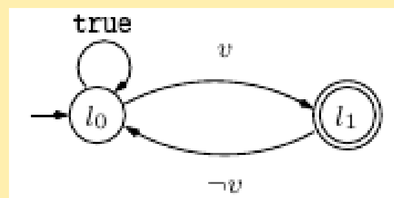
(„beskonačno puta važi v"), gde je v neko svojstvo stanja (u prikazu automata su stanja tranzicionog sistema apstrahovana azbukom $\{v, \neg v\}$, jer nas samo to svojstvo zanima)



Primer 8.4.3 Bihijev automat za formulu

$$GF(v \wedge X\neg v)$$

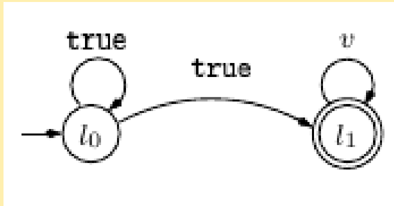
(„beskonačno puta imamo da u nekom stanju važi v, pri čemu odmah u sledećem stanju važi $\neg v$")



Primer 8.4.4 Bihijev automat za formulu

$$FGv$$

(„počev od nekog stanja v zauvek važi”). Za ovo svojstvo ne postoji deterministički Bihijev automat



Postupak provere LTL svojstva ϕ

- Konstruišemo Bihijev automat $\mathcal{A}$ za formulu $\neg\phi$
- Napravimo presek dobijenog automata sa tranzicionim sistemom T (tj. razmatramo samo one ω-reči koje predstavljaju izvršavanja u sistemu T)
- Ako dobijeni presek definiše prazan jezik, svojstvo ϕ je valjano u T

Za proizvoljnu LTL formulu ϕ veličina dobijenog automata $\mathcal{A}$ je eksponencijalna u odnosu na veličinu formule ϕ. Ovo obično nije problem jer formule koje opisuju svojstva u principu nisu velike. Međutim, izračunavanje veličine preseka automata $\mathcal{A}$ sa tranzicionim sistemom T je proporcionalno proizvodu njihovih veličina, tj $|\mathcal{A}| \cdot |T|$. Iako su formule (i njima odgovarajući automati) često relativno male, tranzicioni sistemi po pravilu nisu.

8.4.2 Kombinatorna eksplozija

Na primeru Bihijevih automata videli smo da je glavni problem u prevelikom broju stanja tranzicionih sistema koji se verifikuju. U slučaju jako velikog broja stanja, često nije moguće ni predstaviti odgovarajući graf u memoriji. Ovaj problem se rešava na više načina:

- Apstrakcijom: stanja se apstrahuju skupovima predikata koji su u tim stanjima zadovoljeni
- Simboličkom proverom modela: stanja i relacija prelaska se ne predstavljaju eksplicitno, već pomoću iskaznih formula i BDD dijagrama
- Ograničenom proverom modela: tražimo kontraprimer u prvih k koraka, za dato k

8.4.3 Apstrakcija predikata

Neka je dat tranzicioni sistem $T = (S, \longrightarrow, I, \mathcal{V}, \lambda)$. Apstrakcija predikata podrazumeva sistem T' kod koga je:

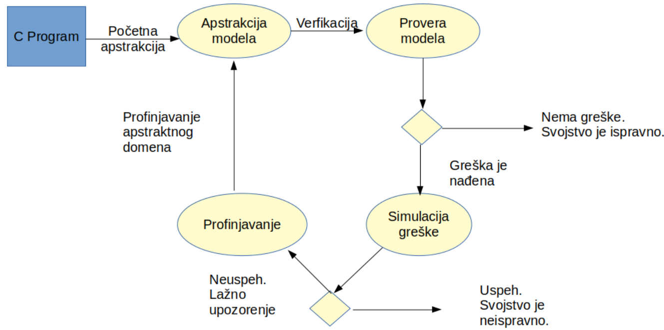
- skup stanja $S' = \{s' \mid s' = \lambda(s)\} \subseteq \mathbb{P}\mathcal{V}$ je familija podskupova skupa $\mathcal{V}$ (stanje $s' = \lambda(s)$ nazivamo apstrakcijom stanja s)
- skup početnih stanja I' odgovara apstrakcijama stanja iz I
- relacija prelaska: prelaz $s_1' \longrightarrow' s_2'$ postoji u T' akko postoji prelaz $s_1 \longrightarrow s_2$ u T, gde su s_1' i s_2' apstrakcije stanja s_1 i s_2 sistema T
- skup predikata $\mathcal{V}'$ se poklapa sa $\mathcal{V}$
- funkcija obeležavanja je funkcija identiteta: $\lambda'(s') = s'$

Ako neko svojstvo važi u apstraktnom sistemu, važi i u polaznom. Obrnuto ne mora da važi, tj. ako neko svojstvo ne važi u apstraktnom modelu, možda je to zato što je model suviše apstraktan (u tom slučaju ga moramo popraviti).

Primer 8.4.5 Nastavak primera 8.1.3.

```
1     while(true)
2     {
3 L0:   x = (x+1) % 3;
4 L1:   if(x == 0)
5 L2:    y = -y;
6 L3:   z = z*y + 1;
7     }
```

- Stanja: $(L_i, [v_x, v_y, v_z])$, gde je L_i lokacija u programu, a $[v_x, v_y, v_z]$ valuacija promenljivih x, y, z
- Ako su x, y, z promenljive tipa `int`, imamo preko $4000000000^3 \cdot 4$ stanja
- Neka je skup predikata $\mathcal{V} = \{y > 0, |z| < 3\}$
- Apstrakcijom predikata dobijamo skup stanja oblika (L_i, uv) gde je u istinitonosna vrednost predikata $y > 0$, a v predikata $|z| < 3$ (dakle 2 mogućnosti za u i dve mogućnosti za v i četiri moguće lokacije, ukupno $2 \cdot 2 \cdot 4 = 16$ stanja)
- Početno stanje: $(L_0, 11)$



Slika 8.6: CEGAR — prikaz procesa

- Relacija prelaska:

$$
\begin{array}{ccc}
(L_0, uv) & \longrightarrow & (L_1, uv) \\
(L_1, uv) & \longrightarrow & (L_2, uv) \\
(L_1, uv) & \longrightarrow & (L_3, uv) \\
(L_2, uv) & \longrightarrow & (L_3, \overline{u}v) \\
(L_3, uv) & \longrightarrow & (L_0, uv) \\
(L_3, uv) & \longrightarrow & (L_0, u\overline{v})
\end{array}
$$

Primetimo da je ovo nedeterministički tranzicioni sistem.

U prethodnom primeru dobili smo značajno manji tranzicioni sistem koji apstrahuje nebitne detalje (razmatra samo svojstva koja nas zanimaju i šta se sa njima dešava tokom izvršavanja). Dobijeni apstraktni sistem može biti nedeterministički. U prethodnom primeru, na lokaciji L_1 sledeća lokacija može biti L_2 ili L_3 i to ne zavisi od vrednosti predikata $y > 0$ i $|z| < 3$.

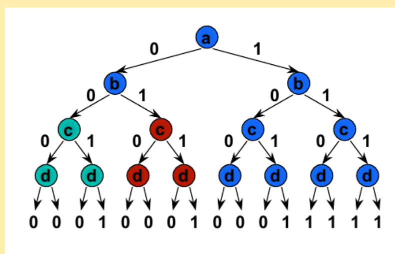
Apstrakcija predikata — profinjavanje

Apstrahovan sistem često može biti previše apstraktan i neprecizan (veoma mali broj svojstava se može dokazati u takvom modelu. Uobičajeno je da se dobijeni kontraprimeri u apstraktnom modelu iskoriste za dobijanje preciznije apstrakcije. Ovaj postupak se naziva CEGAR (eng. *CounterExample-Guided Abstraction Refinement*). Preciznije apstrakcije razmatraju veći broj predikata i samim tim imaju više stanja, ali se u njima može dokazati više svojstava.

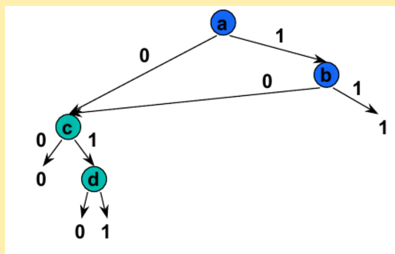
8.4.4 Binarni dijagram odlučivanja

Binarni dijagrami odlučivanja (eng. *binary decision diagram, (BDD)*) predstavljaju kompaktnu strukturu podataka kojom se opisuje semantika iskaznih formula. Ekvivalentne iskazne formule imaju identične BDD-ove. Logičke operacije konjunkcije, disjunkcije i negacije se jednostavno izvode nad BDD-ovima.

Primer 8.4.6 Stablo odlučivanja za formulu $(a \wedge b) \vee (c \wedge d)$:



Redukovani BDD za istu formulu:



Jedan način formiranja BDD-a je da se najpre napravi potpuno binarno stablo odlučivanja, a da se zatim izvrši redukcija tako što se eliminiše dupliranje identički jednakih podstabala. Efikasniji način je da se vrši dekompozicija formule, uz rekurzivno kreiranje BDD-a za podformule koji se zatim kombinuju primenom logičkih operacija. U svakom slučaju, formiranje BDD-a u najgorem slučaju zahteva eksponencijalno vreme u odnosu na veličinu formule

8.4.5 Primena BDD-ova u proveravanju modela

Setimo se da se svako stanje tranzicionog sistema može apstrahovati skupom predikata koji su tačni u tom stanju.

Ovaj skup predikata koji su tačni možemo da razumemo i kao valuaciju nad svim predikatima. Otuda svim stanjima kojima odgovara isti skup tačnih predikata odgovara jedinstveni BDD.

Slično, relacija prelaska $s \longrightarrow s'$ se može predstaviti BDD-om nad predikatima iz $\mathcal{V} \cup \mathcal{V}'$, gde je $\mathcal{V}' = \{p'_1, \ldots, p'_n\}$ skup kopija predikata $p_1, \ldots, p_n$ iz $\mathcal{V}$ (intuitivno, predikati $p_1, \ldots, p_n$ su iz stanja s, a $p'_1, \ldots, p'_n$ su iz stanja s'). Ovi BDD-ovi se dalje koriste za proveravanje svojstava, bez eksplicitnog čuvanja grafa tranzicionog sistema u memoriji.

Simboličko proveravanje invarijanti

Neka je $I(s)$ formula (ili BDD) koja opisuje početne uslove, a $T(s, s')$ formula koja opisuje relaciju prelaska tranzicionog sistema.

Stanja s' koja su dostupna iz početnih stanja s u jednom koraku se mogu predstaviti sledećom formulom:

$$(\exists s)(I(s) \wedge T(s, s'))$$

Primenom odgovarajućih operacija nad polaznim BDD-ovima dobijamo novi BDD $B_1(s)$ koji opisuje stanja dostižna u jednom koraku. Dijagram

$$D_1(s) = I(s) \vee B_1(s)$$

predstavlja uniju početnih stanja i stanja dobijenih u jednom koraku. Sličnim postupkom nad $D_1(s)$ i $T(s, s')$ dobijamo dijagram $D_2(s)$ koji opisuje stanja dostižna u najviše dva koraka, itd. (važi $D_1(s) \subseteq D_2(s) \subseteq D_3(s) \subseteq \ldots$). Kako je ukupan broj stanja konačan, opisani postupak ima fiksnu tačku (tj. za neko k važi $D_{k+1}(s) = D_k(s)$). Dijagram $D(s) = D_k(s)$ predstavlja sva dostižna stanja.

Ako je invarijanta data iskaznom formulom $\phi = \phi(s)$, tada formula $D(s) \wedge \neg\phi(s)$ treba da bude nezadovoljiva (kako bi bilo ispunjeno svojstvo da je u pitanju invarijanta).

8.4.6 Ograničena provera modela

Kod ograničene provere modela (eng. *bounded model checking*) ideja je da kontraprimer tražimo samo u prvih k koraka.

Vrednost k se može povećavati do neke razumne granice, zavisno od resursa kojima raspolažemo. Metoda je korisna kod jako velikih modela, jer se ograničava samo na jedan deo grafa, bez njegove eksplicitne konstrukcije. Može pronaći mnoge česte tipove grešaka, ali može i propustiti neke greške. Ograničena provera modela se zasniva na korišćenju SAT i SMT rešavača.

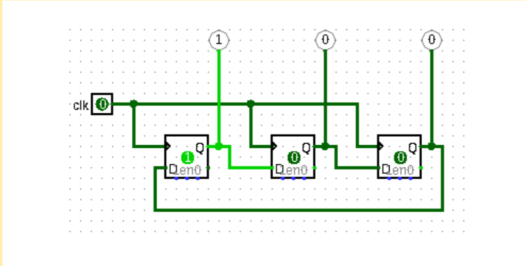
U okviru ograničene provere modela upotrebljava se ograničena semantika LTL operatora. Naime, semantika LTL formula je definisana na (beskonačnim) putanjama σ tranzicionog sistema T. Ukoliko se ograničimo na prefikse tih putanja dužine k, tada to menja semantiku LTL operatora. Formulu ćemo smatrati tačnom ako nema kontraprimera dužine najviše k. Formalno, za rep $\sigma_{|i}$ putanje σ definišemo:

- $\sigma_{|i} \vDash X\phi$ ako $\sigma_{|i+1} \vDash \phi$ za $i < k$, odnosno uvek ako je $i \geq k$ (jer ne možemo da pretpostavimo da nije, ako je to izvan domašaja prefiksa dužine k)
- $\sigma_{|i} \vDash F\phi$ za svako σ i svako i (jer je kontraprimer za $F\phi$ putanja na kojoj ϕ ne važi nikad, a to ne možemo da tvrdimo na osnovu prefiksa dužine k)
- $\sigma_{|i} \vDash G\phi$ ako je $\bigwedge_{j=i,\ldots,k}(\sigma_{|j} \vDash \phi)$ za $i < k$; za $i \geq k$ je uvek tačna (jer nemamo kontraprimer dužine manje od k)
- Slično za $\phi_1 U \phi_2$ i $\phi_i W \phi_2$

SAT kodiranje tranzicionog sistema

Neka je $\mathcal{V} = \{p_1, \ldots, p_n\}$ skup predikata tranzicionog sistema T. Pretpostavimo ponovo da je u T izvršena apstrakcija datih predikata, tj. da su stanja poistovećena sa skupovima predikata koji su tačni u tim stanjima. Za dato k napravimo $k+1$ kopija skupa $\mathcal{V}$: iskazne promenljive $p_1^i, \ldots, p_n^i$ odgovaraju predikatima u stanju s_i, za $i = 0, \ldots, k$. Neka je $I(s)$ iskazna formula nad promenljivama koje odgovaraju stanju s koja definiše uslove za početna stanja (tj. $I(s)$ znači „stanje s je početno stanje"). Neka je $T(s, s')$ iskazna formula koja opisuje relaciju prelaska, tj. vezu između vrednosti predikata u tekućem i u narednom stanju (tj. $T(s, s')$ znači „postoji prelaz iz s u s'". Sada se izvršavanje sistema T u prvih k koraka može opisati iskaznom formulom $I(s_0) \wedge T(s_0, s_1) \wedge \ldots \wedge T(s_{k-1}, s_k)$

Primer 8.4.7 Nastavak primera 8.1.2.



- Neka su p, q, r predikati koji odgovaraju bitovima registra sa leva u desno i neka je $k = 3$
- $I(s_0) \equiv (p^0 \wedge \neg q^0 \wedge \neg r^0)$ — početno stanje je 100.
- $T(s_i, s_{i+1}) \equiv (p^{i+1} \Leftrightarrow r^i) \wedge (q^{i+1} \Leftrightarrow p^i) \wedge (r^{i+1} \Leftrightarrow q^i)$ — s_i prethodi s_{i+1} i nastaje cikličnim pomeranjem s_i u desno.
- Rad kola u prva tri koraka se opisuje formulom:

$$I(s_0) \wedge T(s_0, s_1) \wedge T(s_1, s_2) \wedge T(s_2, s_3)$$

LTL formule se svode na SAT na sličan način, imajući u vidu njihovu ograničenu semantiku.

- $\sigma_{|i} \models X(\phi(p_1, \ldots, p_n))$ kodiramo sa $\phi(p_1^{i+1}, \ldots, p_n^{i+1})$ za $i < k$, ili sa $\top$ za $i \geq k$
- $\sigma_{|i} \models F(\phi(p_1, \ldots, p_n))$ kodiramo sa $\top$
- $\sigma_{|i} \models G(\phi(p_1, \ldots, p_n))$ kodiramo sa $\bigwedge_{j=i,\ldots,k} \phi(p_1^j, \ldots, p_n^j)$, ili sa $\top$ za $i \geq k$

Primer 8.4.8 Na primer, $\sigma \models G(p \Rightarrow Xq)$ se može predstaviti formulom $\bigwedge_{i=0,\ldots,k}(p^i \Rightarrow q^{i+1})$

Neka je svojstvo dato LTL formulom ϕ i neka je $\sigma \models \phi$ za prvih k koraka predstavljeno iskaznom formulom Φ. Problem ispitivanja da li je svojstvo ϕ valjano u tranzicionom sistemu T (do na k prvih koraka) svodi se na ispitivanje valjanosti formule

$$I(s_0) \wedge T(s_0, s_1) \wedge \ldots \wedge T(s_{k-1}, s_k) \Rightarrow \Phi$$

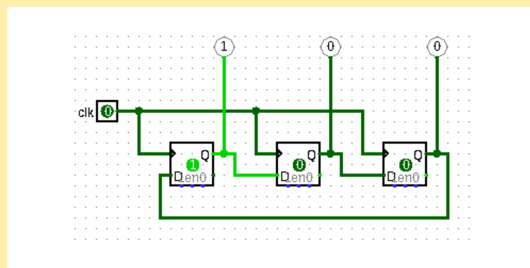
Ovo je ekvivalentno sa proverom zadovoljivosti formule

$$I(s_0) \wedge T(s_0, s_1) \wedge \ldots \wedge T(s_{k-1}, s_k) \wedge \neg\Phi$$

Ukoliko je formula nezadovoljiva, tada nema kontraprimera dužine k (ovo ne znači da nema dužih kontraprimera). Ukoliko je zadovoljiva, zadovoljavajuća valuacija nam daje niz

stanja, tj. prefiks izvršavanja koji ne zadovoljava svojstvo ϕ.

Primer 8.4.9 Nastavak primera 8.1.2.



- Svojstvo: „u svakom trenutku je tačno jedan bit uključen" se zadaje narednim formulama za svako $i = 0, 1, 2, 3$:

$$\Phi_i \equiv (p^i \vee q^i \vee r^i) \wedge (\neg p^i \vee \neg q^i) \wedge (\neg p^i \vee \neg r^i) \wedge (\neg q^i \vee \neg r^i)$$

- Kodiranje svojstva:

$$\Phi \equiv \Phi_0 \wedge \Phi_1 \wedge \Phi_2 \wedge \Phi_3$$

- Formula čija se zadovoljivost ispituje:

$$I(s_0) \wedge T(s_0, s_1) \wedge T(s_1, s_2) \wedge T(s_2, s_3) \wedge \neg\Phi$$

Ograničena provera modela u verifikaciji softvera

SAT kodiranje je obično dovoljno u verifikaciji hardvera, jer se sve izražava u funkciji bitova u sistemu. U slučaju softvera, predikati obično izražavaju složena svojstva poput $x < y$, $x \ mod \ y = 3$ i sl. Slično, relacija prelaska se opisuje akcijama oblika $x \leftarrow y + z$ i sl. Kodirati ove uslove i akcije na jeziku iskazne logike je veoma teško. Predikati se mogu apstrahovati iskaznim slovima, međutim, time se gubi semantička veza među njima (npr. iz $x < y$ i $y < z$ sledi $x < z$, ali to na iskaznom nivou nije moguće zaključiti. Rešenje: korišćenje SMT rešavača.

SMT rešavači predstavljaju nadogradnju SAT rešavača tako da omogućavaju rezonovanje u nekoj odlučivoj teoriji prvog reda. Najpre se ispituje zadovoljivost formule u iskaznom smislu (zanemarujući značenje predikata u teoriji), a zatim

se za dobijeni iskazni model proverava zadovoljivost u teoriji. SMT rešavači podržavaju

- aritmetičke teorije koje su pogodne za izražavanje akcija i uslova u standardnim programima,
- teoriju neinterpretiranih funkcija koja omogućava apstrakciju,
- teorije nizova i rekurzivnih struktura podataka (listi, stabala),
- teoriju bitvektora za precizno izražavanje semantike tipova u programskim jezicima.

U okviru postupka SMT kodiranja programske promenljive predstavljamo promenljivama u odgovarajućoj teoriji (npr. `int` promenljive možemo predstaviti promenljivama sorte `Int`). Lokaciju u programu takođe možemo predstaviti celobrojnom promenljivom. Za stanja $s_0, \ldots, s_k$ imamo $k+1$ kopiju odgovarajućih promenljivih (npr. programska promenljiva x u i-tom stanju biće predstavljena promenljivom x_i u SMT formuli). Početni uslovi se izražavaju kao formula nad promenljivama u stanju s_0. Relacija prelaska ima oblik višestruke $if - else - if$ konstrukcije koja razmatra različite lokacije i primenjuje odgovarajuće akcije na tim lokacijama (tj. povezuje promenljive iz tekućeg stanja sa promenljivama iz narednog stanja. Svojstva se izražavaju u terminima SMT promenljivih, na sličan način kao i u SAT slučaju.

Primer 8.4.10 Primer SMT kodiranja
Neka na lokaciji p imamo naredbu
```
x = x - 1
```
Neka je l_i promenljiva koja sadrži redni broj programske lokacije u i-tom stanju, a x_i promenljiva koja sadrži vrednost promenljive x u i-tom stanju. Relacija prelaska će biti izražena konstrukcijom

$$\begin{aligned} & if \quad l_i = p \\ & \quad then \quad x_{i+1} = x_i - 1 \wedge \bigwedge_{y \neq x} (y_{i+1} = y_i) \wedge l_{i+1} = p + 1 \\ & \quad else \quad R \end{aligned}$$

gde je R ostatak $if - else - if$ konstrukcije koji razmatra ostale lokacije. Ova formula iskazuje sledeće: „na lokaciji p umanjujemo x za jedan, dok ostale promenljive (označene sa y) ostaju nepromenjene. Prelazimo na sledeću lokaciju”

Primetimo da se upotrebom SMT rešavača izbegava potpuna apstrakcija predikata: stanja sistema jesu određena vrednostima atomičkih predikata koji se pojavljuju u formuli, ali se njihova semantika ne gubi, jer imamo SMT teoriju koja rezonuje o njima. Korišćenje aritmetičke teorije je samo aproksimacija stvarne semantike programa, jer su `Int` promenljive u SMT-u nad beskonačnim domenom $\mathbb{Z}$, dok je u računaru domen konačan. Teorija bitvektora podrazumeva semantiku koja je identična semantici tipova u programskim jezicima ali je rezonovanje u ovoj teoriji vrlo zahtevno. Korišćenje aproksimacije omogućava da se lakše/efikasnije dokažu neka tvrđenja.

Literatura

- Biere A., Heule M., van Maaren H. Handbook of satisfiability. 2009.
- Baier C., Katoen J. P., Larsen K. G. Principles of model checking. 2008.
- Merz, S. An introduction to model checking. 2008.

Apstraktna interpretacija 9

Apstraktna interpretacija (engl. *abstract interpretation*) je teorijski okvir za formalizaciju apstrakcije. Naime, konkretna semantika programa je previše kompleksna da bi se o njoj moglo efikasno rezonovati. Zbog toga je potrebno apstrahovati konkretnu semantiku programa u nekakav smislen nadskup o kome je moguće efikasno rezonovati. Apstrakcija treba da bude takva da ako se dokaže ispravnost u apstrahovanoj semantici, onda ispravnost sigurno važi i u konkretnoj semantici.

9.1 Odnos konkretne i apstraktne semantike 157
9.2 Primeri 160
9.3 Izbor apstrakcije . 162
9.4 Primene 164

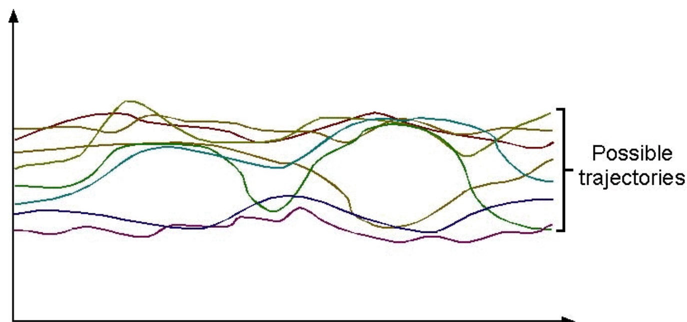
Osnovne karakteristike

- Skalira dobro na velikim programima
- Ne propušta greške, ali može imati lažna upozorenja
- Primena u avio-industriji, automobliskoj industriji, svemirske letilice — neki standardi zahtevaju upotrebu statičke analize i apstraktne interpretacije.

Osnovne ideje apstraktne interpretacije datiraju iz 1977. godine: *Patrick Cousot, Radhia Cousot: "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints"*

9.1 Odnos konkretne i apstraktne semantike

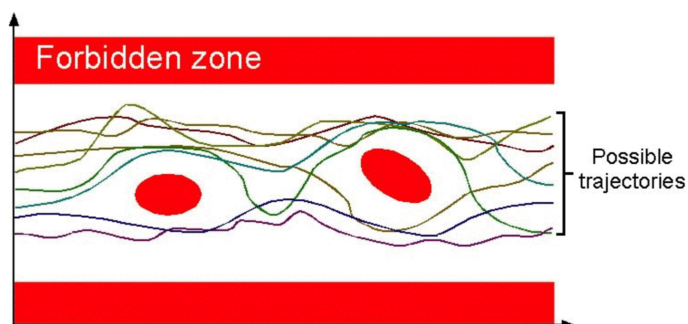
Konkretna semantika programa formalizuje skup svih mogućih izvršavanja programa u interakciji sa svim mogućim sredinama izvršavanja. Ako se izvršavanje predstavi kao kriva koja pokazuje promenu vektora x(t) vrednosti ulaza, stanja i izlaznih promenljivih programa kao funkciju vremena t, konkretna semantika onda može da se predstavi skupom krivih:



Konkretna semantika programa je jedan beskonačni matematički objekat koji nije izračunljiv: nije moguće napisati program koji može da predstavi i izračuna sva moguća izračunavanja za bilo koji program u svim njegovim mogućim uslovima izvršavanja. Prema tome, sva netrivijalna pitanja o konkretnoj semantici programa su neodlučiva: nije moguće napisati program koji može da odgovori na bilo koje pitanje o izvršavanjima bilo kog programa (jer bi inače konkretna semantika programa morala da bude izračunljiva).

9.1.1 Provera važenja sigurnosnih svojstava

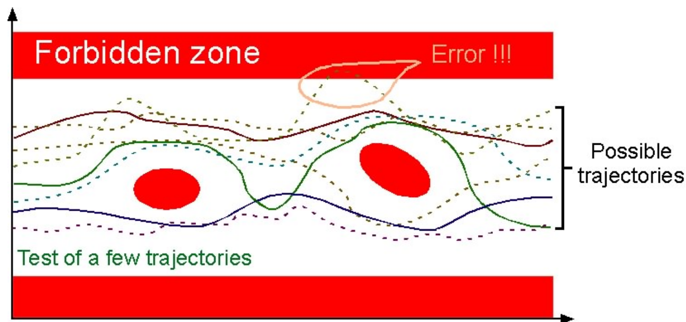
Sigurnosna svojstva programa izražavaju da ne postoji izvršavanje programa koje može da dođe do stanja greške. Grafički, skup stanja u kojima dolazi do greške predstavlja zabranjenu zonu, označenu crvenom bojom na slici.



Verifikacija sigurnosnog svojstva sastoji se u dokazivanju da je presek konkretne semantike programa sa zabranjenom zonom prazan. Kako je konkretna semantika neizračunljiva, problem verifikacije je neodlučiv. Nije uvek mougće da se odgovori na pitanje o sigurnosti programa kompletno automatski, sa konačnim resursima bez neizvesnosti u odgovor i bez ljudske intervencije.

Testiranje i simboličko izvršavanje

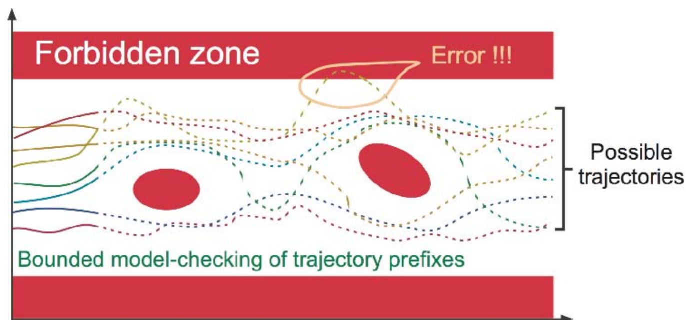
Testiranje se sastoji od razmatranja podskupa skupa svih mogućih izvršavanja:



U ovom slučaju, ako sve proverene putanje nemaju preseka sa zabranjenom zonom, to i dalje ne znači da ne postoji putanja koja nema preseka sa zabranjenom zonom. Slično važi i za simboličko izvršavanje jer ni sa simboličkim izvršavanjem najčešće ne možemo da prodjemo kroz sve moguće putanje do kraja.

Ograničeno proveravanje modela

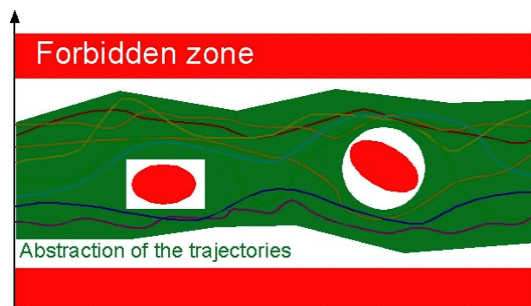
Ograničeno proveravanje modela se sastoji u istraživanju prefiksa svih mogućih izvršavanja:



Ograničeno proveravanje modela takođe može da propusti pronalaženje greške jer iako se posmatraju sve putanja, one se posmatraju samo da određenog dela a greška može kasnije da se javi.

Apstraktna interpretacija

Apstraktna interpretacija se sastoji od razmatranja apstraktne semantike, koja je nadskup konkretne semantike programa:



Apstraktna semantika pokriva sve moguće slučajeve. Prema tome, ako je apstraktna semantika bezbedna, tj ne preseca se sa zabranjenom zonom, onda je takva i konkretna semantika.

9.2 Primeri

Semantika programa može se opisati konkretnim domenom D_c i relacijama nad ovim domenom. Ove relacije se mogu menjati tokom izvršavanja naredbi programa. Za velike programe, ispitivanje da li neko svojstvo važi nad domenom D_c otežano je zbog veličine samog domena, zbog velikog broja mogućih putanji kroz program kao i zbog neodlučivosti koja se javlja u raznim kontekstima. Jedan način prevazilaženja ovih poteškoća je aproksimacija konkretnog domena D_c apstraktnim domenom D_a, odnosno konkretan domen vrednosti zamenjuje se apstraktnim domenom opisa ovih vrednosti.

Iako apstraktni domen nije tako precizan kao konkretan domen, on može u nekim situacijama da dâ odgovore o važenju nekih svojstva.

Primer 9.2.1 Da li je broj paran ili neparan?

- Konkretna interpretacija: proverimo ostatak pri deljenju sa dva.
- Apstraktna interpretacija: proverimo poslednju cifru broja.

U ovom primeru imamo konkretan domen koji čine svi prirodni brojevi i apstraktni domen koji čine cifre od 0 do 9, tj svaki broj se preslikava u svoju poslednju cifru. Razmatranje poslednje cifre je, u opštem slučaju, efikasnije nego deljenje.

Primer 9.2.2 Da li je broj pozitivan, nula ili negativan? Beskonačni domen skupa celih brojeva može se zameniti apstraktim domenom koji sadrži vrednosti znakova brojeva, tj. skupom $\{+, -, 0\}$.

$$\begin{aligned} a_0 &= \{0\} \\ a_+ &= \{n \mid n > 0\} \\ a_- &= \{n \mid n < 0\} \end{aligned}$$

Ovakva apstrakcija može da nam da potpuno precizan odgovor na pitanje znaka množenja dva broja:

$$\begin{aligned} 0 \times a_+ &= 0 \times a_- = a_+ \times 0 = a_- \times 0 = 0 \\ a_+ \times a_+ &= a_- \times a_- = a_+ \\ a_+ \times a_- &= a_- \times a_+ = a_- \end{aligned}$$

Međutim, ova apstrakcija ne može da nam da precizan odgovor u kontekstu znaka sabiranja i oduzimanja svaka dva broja:

$$\begin{aligned} a_+ + a_+ = a_+ - a_- = a_+ + 0 = 0 + a_+ = 0 - a_- &= a_+ \\ a_- + a_- = a_- - a_+ = a_- + 0 = 0 + a_- = 0 - a_+ &= a_- \\ a_+ + a_- = a_- + a_+ = a_+ - a_+ = a_- - a_- &= ? \end{aligned}$$

Zato je potrebno proširiti skup apstrakcija tako da obuhvata i sve moguće brojeve.

$$\begin{aligned} a_0 &= \{0\} \\ a_+ &= \{n \mid n > 0\} \\ a_- &= \{n \mid n < 0\} \\ a &= \{n\} \end{aligned}$$

Sada je:

$$\begin{aligned} 0 + 0 = 0 - 0 &= 0 \\ a_+ + a_+ = a_+ - a_- = a_+ + 0 = 0 + a_+ = 0 - a_- &= a_+ \\ a_- + a_- = a_- - a_+ = a_- + 0 = 0 + a_- = 0 - a_+ &= a_- \\ a_+ + a_- = a_- + a_+ = a_+ - a_+ = a_- - a_- &= a \\ a + a_+ = a_+ + a = a_- + a = a + a_- &= a \\ a - a_+ = a_+ - a = a_- - a = a - a_- &= a \\ a + 0 = 0 + a = 0 - a = a - 0 &= a \end{aligned}$$

Ovo proširenje, odnosno dodavanje a nam je zapravo gubitak informacije, situacija u kojoj ne znamo ništa o znaku rezultata.

9.3 Izbor apstrakcije

Apstrakcije se biraju u skladu sa problemom koji se ispituje. Pravi izbor apstrakcije je suštinski za apstraktnu interpretaciju.

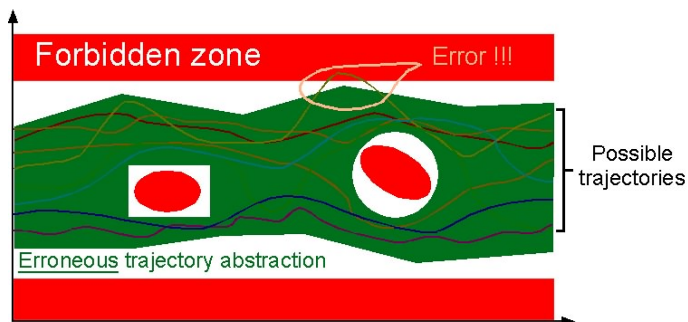
Dopuniti

Primeri nekih apstraktnih domena koji se često upotrebljavaju http://www.dsi.unive.it/~avp/domains.pdf

Apstraktna semantika treba da bude saglasna, da ostane dovoljno precizna da se izbegnu lažna upozorenja, kao i da ostane dovoljno jednostavna da se izbegne fenomen kombinatorne eksplozije.

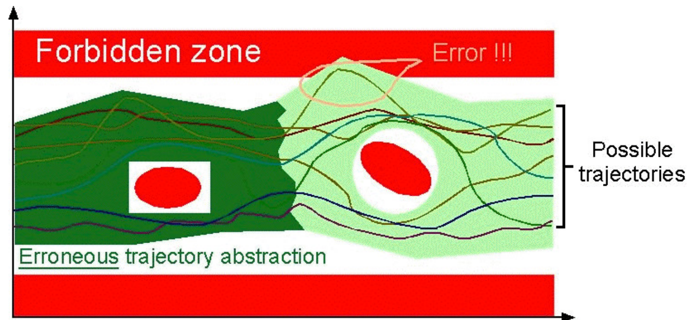
9.3.1 Pogrešne apstrakcije

U okviru formalnih metoda, apstraktne semantike moraju da se izaberu kao nadskup svih mogućih konkretnih semantika jer u suprotnom apstraktno reznovanje može da bude nekorektno za konkretno rezonovanje. Na narednoj slici predstavljena je pogrešna apstrakcija koja ne sadrži sva moguća izvršavanja i zbog toga propušta grešku koja se javlja za putanju koja nije obuhvaćena apstrakcijom.



Na sličan način se mogu posmatrati i apstrakcije koje se javljaju kod proveravanja modela. Kao što je već pominjano, tehnike kao što su ograničeno proveravanje modela, iako formalne, ne istražuju sve moguće putanje do kraja već samo

prefikse putanja. U slučaju apstrakcije predikata kod proveravanja modela sa profinjavanjem proces može da se ne završi, što vodi do toga da neke greške mogu da se ne otkriju.



Međutim, treba imati u vidu da su ovo pristupi koji se koriste pre svega za pronalaženje grešaka, a ne za dokazivanje korektnosti tako da apstrakcije nisu pogrešne ukoliko se uzme u obzir kontekst pronalaženja grešaka već su pogrešne samo ako se pristup koristi za dokazivanje ispravnosti programa.

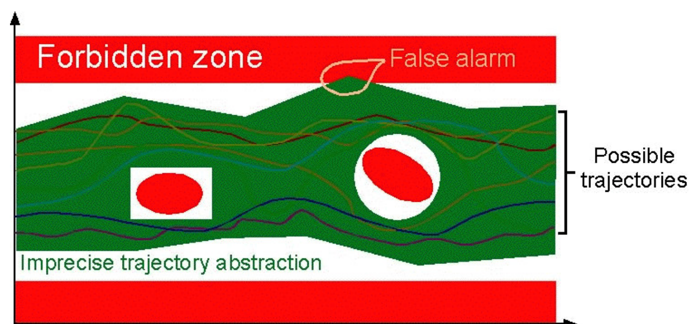
Lažna upozorenja

Apstraktne semantike na koju se formalne metode oslanjaju su:

- korektne/saglasne, odnosno su nadskup konkretne semantike i
- jednostavne, ili barem dovoljno jednostavne da mogu da se predstave u okviru mašine.

U odsustvu upozorenja, ovo vodi dokazu korektnosti programa. Ipak, posledica aproksimacije svih mogućih izvršavanja je da se razmatraju i neka nepostojeća izvršavanja, od kojih neka mogu da vode do greške, što onda vodi do lažnih upozorenja.

Lažna upozorenja odgovaraju slučajevima kada apstraktna semantika preseca zabranjenu zonu dok je konkretna semantika ne preseca. Dakle, signalizira se greška do koje ne može stvarno da dođe u realnosti:



Apstrahovanjem se mogu izgubiti važne informacije što je uzrok lažnih upozorenja. Biranje adekvatne aproksimacije domena oslanja se na monotone funkcije u okviru parcijalno uređenih skupova, posebno na teoriju mreža (eng. *lattice theory*), na računanje fiksnih tačaka parcijalno uređenih skupova, na Galoaove veze (eng. *Galois connections*).

9.4 Primene

Apstraktna interpretacija ima niz primena: kompilacija, određivanje invarijanti u verifikaciji, verifikacija: automobilska/avio/svemirska industrija.

Alati (sa skupim licencama)

- Astrée — AbsInt
- Polyspace Bug Finder — MathWorks
- Coverity — Synopsys

Alati koji su slobodno dostupni

- CPAchecker — Free software, Apache 2.0 License
- Frama-C value analysis — Open source software

Literatura

`http://www.di.ens.fr/~cousot/AI/`

Na srpskom:

- Seminarski rad, Milan Čugurović: Intuicija.
- Seminarski rad, Dimitrije Špadijer: Teorija.

Alphabetical Index

predgovor, v