Formalni dokazi u programiranju

Šarić, Kristijan

Sveučilište Jurja Dobrile u Puli
Fakultet ekonomije i turizma
«Dr. Mijo Mirković»

Kristijan Šarić

FORMALNI DOKAZI U PROGRAMIRANJU

Završni rad

Pula, 2015.

Sveučilište Jurja Dobrile u Puli
Fakultet ekonomije i turizma
«Dr. Mijo Mirković»

Kristijan Šarić

FORMALNI DOKAZI U PROGRAMIRANJU

Završni rad

JMBAG: 0145022678, izvanredni student

Studijski smjer: Informatika

Predmet: Programiranje

Mentor: Doc. dr. sc. Krunoslav Puljić

Pula, rujan 2015.

Posveta

Zlatku Sirotiću, jer je dokaz da godine ne mogu
ubiti znatiželju i potragu za znanjem

Sadržaj

1. **Uvod** ... 1
2. **Uvod u formalne dokaze** ... 2
2.1. Zašto formalni dokazi? ... 4
2.1.1. Mars Climate Orbiter ... 5
2.1.2. Mariner i svemirska sonda ... 5
2.1.3. Ariane 5 let 501 ... 6
2.1.4. EDS greška ... 6
2.1.5. Therac-25 Medical Accelerator ... 6
2.1.6. Multidata systems ... 7
2.2. Niste uvjereni? ... 8
2.3. Što su formalni dokazi zapravo? ... 9
2.5. Propozicijska i predikatna logika ... 10
3. **COQ** ... 13
3.1. Osnove COQ-a ... 15
3.2. Peano brojevi ... 18
4. **Primjer formalnog dokaza** ... 20
4.1. Podatkovna struktura, lista ... 23
4.2. Lista, formalni dokaz ... 25
5. **Zaključak** ... 29
Popis literature ... 30
Popis slika ... 31

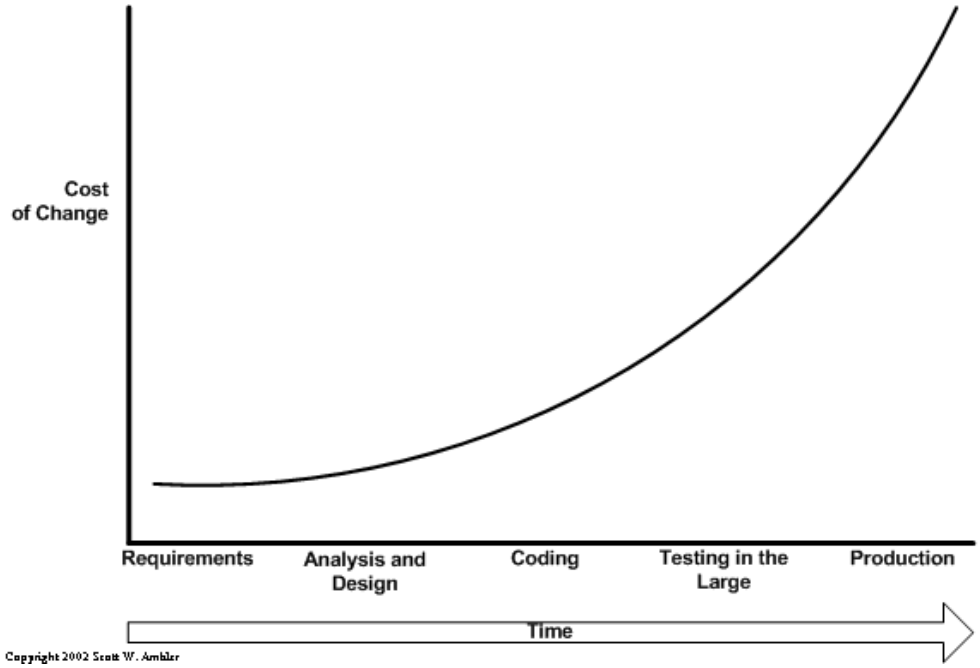
1. Uvod

Cilj ovog rada je pokazati kako formalni dokazi imaju mjesta u programiranju i kako je sve potrebnije da budu dio svakodnevnog programiranja. Kako vrijeme ide naprijed, tehnologija napreduje, te nam korištenje formalnih dokaza kao specifikacija programa i kao njihovih provjera postaje sve lakše i lakše. Autor će pokušati na jednostavan način, kroz primjere, proći osnove koje su potrebne da bi se razumjeli formalni dokazi. Samim time, autor se nada da će čitaoci moći vidjeti da formalni dokazi nisu magični, te da su zapravo, vrlo - logični. Potrebno je malo promijeniti način gledanja, da bismo uvidjeli kako možda možemo napraviti program, koji formalno verificiran, generiran iz specifikacije, može raditi bez grešaka. Pretpostavka je da čitatelji već poznaju osnove programiranja, osnove neke od (statičnih) funkcionalnih jezika (Haskell, OCaml, Scala, ...) te da poznaju osnove logike - propozicijska, predikatna. Iako je autor imao namjeru sve ovo uvesti u rad kao uvod, previše je toga da bi se pisalo, te ako niste upoznati sa navedenim, molim pogledati materiju te se vratiti na rad.

2. Uvod u formalne dokaze

Softver danas. Softver danas pišu ljudi. Ljudi rade greške. Stoga softver sadržava greške. I to u velikim količinama. Na žalost ne čujemo puno priča o tome, jer su, nažalost, zanimljive one priče u kojima ljudi izgube živote, a takvih je, još uvijek, relativno malo direktno vezano uz računalne programe. Cijena izmjene ili popravka greški (*eng. bug-a*) dok je softver u produkciji je ogromna, otprilike 100 puta veća nego u fazi oblikovanja.

Slika 1*Traditional cost of change curve*



Izvor: Ambler, S. (2014): Examining the Agile Cost of Change Curve, http://www.agilemodeling.com/essays/costOfChange.htm (15. lipanj 2015.)

Samo taj iznos, 100 puta veći iznos od onog u početku, dovoljan je da se čovjek zapita da li je potrebno imati neke metode provjere sa kojima možemo uložiti više vremena na početku kako bismo bili sigurni da nećemo imati greške u programu.

Najčešće, u svijetu programiranja, dobijemo pojedini zadatak za obaviti. Ljudi smo, i kao takvi, ograničeni. Pojedini zadatak razbijemo u manje dijelove kako bi "nam stao u glavu". Budući da je kratkotrajno pamćenje ograničeno, ta tzv. radna memorija našeg uma (uma s razlogom) mora sadržavati sve podatke o kojima razmišljamo, tako da možemo "vidjeti cijelu sliku" tog zadatka. Ali to možemo napraviti na dva načina. Ili puno ponavljati taj zadatak dok nam ne pređe u dugotrajno pamćenje, sa kojim onda možemo sadržavati puno više podataka nego prije. To je ono što se razvija vježbom i za to treba vremena. Ali kada dobijemo zadatak koji treba napraviti, u većini slučajeva nemamo vremena puno vremena provesti sa njime. Tako nam ostaje drugo rješenje - da razbijemo problem u manje dijelove koje možemo "držati u glavi", u kratkotrajnom pamćenju. Ti zadatci su jako mali, te nije čudno da, kao programeri, radimo puno grešaka. Vidjeti iglu u sijenu je isto kao i vidjeti mali dio programa u svojoj glavi. Nemamo pojma što ostalo sijeno radi, a samim time, ne vidimo posljedice naših izmjena.

Naravno, pojava novih, agilnih metodologija, popravila je situaciju, budući da sada taj ostatak programa provjerava samo računalo kada mi pokrenemo testove. Testovi nam onda služe kao nekakvi ugovori koje mi postavimo da uvjerimo sami sebe kako bi ostatak programa trebao raditi kako smo definirali. Ako je ugovor prekršen, test javi grešku. Ako svi testovi prođu, onda znači da bi naš program vjerojatno trebao odgovarati našim specifikacijama. Nažalost, bez testova nemamo taj ugovor i nemamo potvrdu da li ostatak programa radi u skladu sa našim očekivanjima. Ignorantno je i bahato misliti da ćemo nešto napraviti točno bez da sami sebe provjeravamo kada radimo takve delikatne poslove. Cilj je biti pedantan i nepovjerljiv, kao i svaki pravi znanstvenik.

Međutim, testovi su ograničeni, te ne mogu pokriti sve mogućnosti. Sa testovima pokrijemo možda pokoju mogućnost pojedinog djela programa i kažemo da program odgovara našoj specifikaciji. To baš nije sasvim točno. Budući da je specifikacija prilično sveobuhvatna, ne pokriva samo par mogućih vrijednosti. Tako su testovi možda samo prvi korak do specifikacije.

Sljedeći oblik definiranja specifikacije popularan je u funkcionalnim jezicima kao što su Haskell, Scala i drugi. Taj oblik se naziva testiranje svojstva (*eng. Property-based testing*). Najpopularnija biblioteka za tu svrhu zove se *QuickCheck*. Razlika između "običnog" testiranja i testiranja svojstava jest što se u potonjem definiraju svojstva kao što je npr. "funkcija max(n) ne smije vratiti manji broj od maksimuma brojeva ove liste". Ovakav način testiranja je ograničen sa druge strane jer opet ima ograničenu količinu provjera (standardno je 100 provjera pojedinog svojstva).

Na koncu, postoji i ona krajnost koja opisuje što je formalna verifikacija, formalni dokaz ispravnosti programa. Verifikacija programa je slično što i validacija programa, ali postoji sitna razlika. Verifikacija programa je provjera da li program radi ono što programer želi da radi, a validacija je da li program radi ono što korisnik očekuje da radi. Možemo reći da su verifikacija i validacija ista stvar ako ne marimo iz čije perspektive gledamo sam razvoj – razvojne ili korisničke. Radi se o verifikaciji programa koji je već napisan ili generiranju programa iz formalne specifikacije. Formalna verifikacija dokazuje da je program potpuno korektan za sve vrijednosti za koje to dokažemo te da se ponaša potpuno u skladu sa ugovorom kojeg smo postavili. Nije potrebno testirati ga jer već radi u skladu sa našim ugovorima. Ako se ugovor prekrši, program javlja grešku, i onda programer mora intervenirati. Nije moguće pokrenuti program koji nije ispravan, tj. potpuno verificiran. Za formalnu verifikaciju potrebno je puno vremena te je cijena koja je potrebna da se ona postigne jako visoka. Osim toga, potreban je i jako stručan kadar, koji je bolji što je bliži matematici, što stavlja granicu prilično visoko.

2.1. Zašto formalni dokazi?

Zašto bismo onda koristili formalne dokaze? Zašto bismo trebali potrošiti više vremena, više truda, više energije, više novaca u nešto što već sada radi dovoljno dobro? Zašto bi to bilo bitno? Zašto bismo trebali nešto popravljati više nego što je potrebno? Pa na koncu, moj program se tu i tamo malo zagrcne, malo zamrzne, prestane raditi na koju minutu. Nije to neki veliki problem, kao što sam objasnio korisnicima, samo popijete kavu i malo pričekate, to vam je signal za odmor. To što koja datoteka nestane svako toliko, nije problem. To što ljudi koji testiraju moj softver više ne znaju što testiraju i zašto se javljaju greške, to je njihov problem.

Kako svijet sve više i više kreće u pohod automatizacije i povezivanje kroz računala i kroz internet, stvara se ogroman pritisak na developerima. Sada i bakica od 90 godina treba znati što je „Internet stvari“ (*eng. Internet of things*) i da je to u modi, i da ne može provesti ostatak svog života u miru i neznanju, ne znajući što su „Veliki podatci“ *(eng. BigData)* i „Računarstvo u oblaku“ (*eng. Cloud computing*), kao i još neke štosne nazive koje proizvode zatupljeni mediokriteti koji se više brinu da nešto prodaju nego da nešto proizvode, kao i oni koji, u velikom broju, slijede te iste takozvane eksperte. Danas je očito popularno slijediti popularna mišljenja i ponavljati velike riječi. Bilo je uvijek, ali se danas, možda uz pojavu velike povezanosti i komunikacije, to vidi bolje. Ne želim reći da mi to sve počinje podsjećati na nacističku Njemačku, kako svi odjednom odjekuju sa „Računarstvo u oblaku“, i svaki čovjek na ulici zna nešto o programiranju i računalima jer zna što je je „Računarstvo u oblaku“, iako niti ja sam nisam siguran da znam što to znači. Ne želim to reći.

Kako se rađaju sve mnogobrojnije mogućnosti gdje možemo primijeniti računala i gdje možemo imati nekakav softver, tako se i povećava udio softvera koji se nalazi u kritičnim dijelovima mašinerija o kojima ovise životi ljudi. Koji puta direktno, koji puta indirektno. Imajmo na umu da neke od greška koje ću spomenuti pokazuju greške do “danas”, dok će se količina softvera vjerojatno uvišestručiti tijekom slijedećih desetak godina, a samim time i takve greške ako im ne stanemo na kraj.

2.1.1. Mars Climate Orbiter

1998, “Mars Climate Orbiter”, NASA-ina robotska sonda, vrijedna 125 milijuna dolara pokušala se stabilizirati u atmosferi Marsa. Pokušala, jer se zabila u Mars. Malo je teško pogoditi planet, ali uspjeli su. Greška je bila u pretvorbi jedinica u softveru, te je umjesto metričkog sustava, koristila američki.

2.1.2. Mariner i svemirska sonda

Dok je upisivao formulu koja je bila napisana na papiru u navigacijski računalni sustav, programer je zaboravio upisati eksponentu. Ta jedna greška uzrokovala je da navigacijski računalni sustav sve varijacije tretira kao greške te je sonda nakon 237 sekundi u kojem je trebala krenuti prema Veneri, morala biti uništena iznad Atlanitika. Vrijedila je 18.2 miljuna dolara u 1962.

2.1.3. Ariane 5 let 501

1995, nova europska raketa koja treba lansirati satelit, Ariane 5, koristila je softver od prijašnje rakete, Ariane 4. Na žalost, Ariane 5 je imala brže motore koji su uspjeli izazvati bug koji je, u principu, pokušao ugurati 64-bitni broj u 16-bitni prostor. Greška koja je nastala uzrokovala je pad glavnog i pomoćnog računala te je nakon 36.7 sekundi nakon lansiranja pokrenut mehanizam samo-uništenja, te je na nebu nastao skupi vatromet. Cijena? Sitnica, 8 milijardi za raketu koja je nosila satelit od 500 milijuna dolara.

2.1.4. EDS greška

2004, velika softverska kompanija, EDS, sagradila je ogromni, kompleksni IT sustav za Veliko-britansku agenciju za djecu (*CSA - Child Support Agency*). U točno isto vrijeme, odjel za poslove i penzije (*DWP - Department for Work and Pensions*) odlučio je restrukturirati cijelu agenciju. Restruktura i novi softver nisu bili kompatibilni, i došlo je do neispravljivih grešaka. Sa više od 500 bugova otvorenih na novom sustavu, sudar ta dva događaja obustavio je CSA mrežu. Rezultat je da je sustav uspio preplatiti 1.9 miliona ljudi, podplatiti 700.000, imao je 7 miljardi funti u neskupljenim iznosima za djecu, te je cijeli sustav koštao preko jednu miljardu do danas.

2.1.5. Therac-25 Medical Accelerator

Therac-25 je uređaj za terapiju zračenjem koji su izgradili AECL i CGR Francuske koji se primjenjivao 1985. U mogućnosti je isporučiti dvije različite vrste terapija zračenjem: ili nisko-energetski elektronski snop (beta čestice) ili X-zrake. Nažalost, operativni sustav koji koristi Therac-25 je projektiran i izgrađen od strane programera koji nije imao formalnu obuku. OS je sadržavao suptilnu grešku, i zbog toga je tehničar slučajno mogao konfigurirati Therac-25 tako da elektronska zraka puca u načinu visoke snage bez odgovarajućeg oklopa pacijenta.

Rezultat - u najmanje 6 incidenata (s više osumnjičenih), pacijentima su slučajno davane smrtonosne ili blizu smrtonosne doze zračenja - oko 100 puta veće u odnosu na namjeravane doze. Najmanje 5 smrtnih slučajeva izravno se pripisuju njemu, a drugi su teško ozlijeđeni.

2.1.6. Multidata systems

Američka tvrtka, Multidata Systems International, stvorila je softver za planiranje terapija koji je dizajniran za izračunavanje odgovarajuće doze zračenja za bolesnike podvrgnute terapiji zračenjem. Softver omogućuje stručnjaku za zračenje da nacrta na ekranu gdje će staviti metalne štitove (pod nazivom "blokovi") na pacijenta tijekom liječenja. Ovi blokovi štite zdrave stanice od zračenja. Sam program omogućuje plasman samo 4 bloka, ali panamski liječnici obično koriste pet. Da bi se otarasili ograničenja u softveru, liječnici su odlučili prevariti softver za crtanje tako da svih pet blokova nacrtaju kao jedan blok s rupom u sredini. Nažalost, bug u softveru Multidata uzrokovao je različite rezultate, ovisno o tome kako je nacrtana rupa. Nacrtajte ga na jedan način, doziranje je točno. Nacrtajte ga u drugom smjeru i softver preporučuje dva puta veću dozu.

Rezultat - najmanje 8 pacijenata je umrlo, dok je još 20 primilo predoziranje koje će vjerojatno izazvati značajne zdravstvene probleme. Liječnici, koji su zakonski obvezani da provjere izračune na računalu rukom, optuženi su za ubojstvo.

2.2. Niste uvjereni?

Niste uvjereni da je potrebno imati formalne dokaze? Napravimo jedan misaoni eksperiment da vidite na što mislim. Zamislite svojih par prijatelja, svojih par članova obitelji, par ljudi koji su vam koliko-toliko bliski. Ne pretjerano inteligentni, ne pretjerano nadareni za znanost, dobri prijatelji i obični ljudi. Zamislite da sada sa njima odlazite na put. Odlazite na mjesec, tamo je novi zabavni park koji jednostavno morate vidjeti. Svi se ukrcate na svemirski brod koji vas treba odvesti na mjesec. Nakon što se ukrcate, sjednete u svoju prostoriju (kažimo da postoje prostorije sa ljudima, možete rezervirati pojedine prostorije) sa tim svojim poznanicima. I napredni, ogromni svemirski brod počinje se paliti. Sve se zatrese, ipak trebate savladati gravitaciju, iznimno je moćan taj brod. Kada odjednom začujete neku čudnu buku, neko čudno pištanje. Odjednom, motori se ugase, i vi čujete sa razglasa stjuardesu ugodnog glasa: "Oprostite na smetnji, čini se da postoji neki mali problemi sa pokretanjem broda, molimo putnike da malo pričekaju". Malo strašno, ne? Zamislite se malo.

Ali ne toliko strašno kao da sada dignete pogled na ostatak ljudi u prostoriji i čujete "Da li si popravio onaj bug?", u trenutku kada shvatite da ti ljudi ispred vas rade za firmu koja proizvodi softver za ovaj isti svemirski brod. U šoku pogledate uokolo kada vas vaš suputnik, glavni programer, pogleda direktno u oči sa smješkom na licu koji pokazuje potpuni nedostatak brige, namigne i kaže: "Ne brinite!". Ne znam za vas, ali ja bih ubrzo krenuo na zrak, trčeći od tog broda u postepeno većoj brzini.

Točnost je varijabilna. Apsolutna točnost je samo ideja, te ne postoji u stvarnosti. Apsolutnosti možemo pripisivati božanstvima, one nisu za ljude. Ali maksimalna točnost je realna ideja. Možemo stvoriti softver koji će raditi apsolutno točno ako hardveru radi apsolutno točno. Što znači da imamo maksimalnu točnost, ako može doći do greške u hardveru. Ne možemo utjecati na prirodu apsolutno, uzmimo za primjer kozmičke zrake. Činjenica je da postoje situacije u kojima je potpuna korektnost jedini način da napravimo maksimalno točan softver. Koji puta se zadovoljimo i proglasimo neki softver točnim čim smo ga testirati par puta, primjerice kada radimo program za pisanje teksta. Ako radimo programe koji brinu kako avioni lete, ili kako vaš novac ide sa jednog računa na drugi, koliko terapije trebate dobiti i slične kritične situacije, nemamo izbora nego raditi sa maksimalnom točnosti ili korektnosti jer najmanja greška može proizvesti smrt - direktno ili indirektno.

2.3. Što su formalni dokazi zapravo?

Formalni dokaz je konačni niz koraka sa kojim, kroz mehanički način, poput igre, potvrđujemo teorem koji je postavljen na početku. Objasniti formalnu verifikaciju kroz formalne dokaze kroz neki povijesni period u kojem se povezuju i nadograđuju cjeline jako je opširan posao, te preporučam svim čitateljima da pokušaju uzeti ovaj uvod kao jako kratak, nepotpun ali jednostavan uvod u formalnu verifikaciju kroz formalne dokaze. Ono na što se želimo fokusirati biti će konkretan primjer formalno verificirane podatkovne strukture, konkretno liste.

2.5. Propozicijska i predikatna logika

Možemo ukratko opisati osnove propozicijske i predikatne logike kako bi čitatelj ipak imao okvira za razumijevanje formalnih dokaza.
Krenimo sa propozicijskom logikom (*eng. proposition logic*) koja se tvori, kao što sam naziv kaže propozicijama (*eng. proposition*).
Propozicija je nekakva tvrdnja koju mi damo. U matematičkoj logici nije bitno da li je tvrdnja točna ili ne, to je na onome tko tu tvrdnju postavlja. Drugim riječima, ako ja kažem „Mjesec je zelen“ i ako ja tu tvrdnju koristim dalje u postupku dokazivanja točnosti te tvrdnje, matematika neće imati ništa za reći protiv toga.
Matematička logika je samo „mašinerija“ koja služi kako bi kompleksnije izraze dokazala ili opovrgnula - ona nema svijest, religiju niti mišljenje, ljubav ili mržnju, ideju ili stvarnost. Drugim riječima, matematička logika je samo alat koji može potvrditi formalnost (oblik) izjava. Često koristimo P, Q, R kao varijable iliti propozicije. Same propozicije mogu biti točne ili netočne. Uzmimo primjer:

P = „Mjesec je zelen“

Očito je da je izjava netočna. Ili možda ne u potpunosti, možda postoji neki dio mjeseca koji pod određenim svijetlom stvarno jest zelen – ali ne brinemo se oko takvih varijabilnih vrijednosti, u propozicijskoj logici imamo samo „točno“ i „netočno“. Za varijabilne vrijednosti, čitatelj može pročitati nešto više o neizrazitoj logici (*eng. fuzzy logic*). Ono što još imamo na raspolaganju u propozicijskoj logici su tzv. veznici, načini na koje možemo povezati propozicije. Ako imamo dvije propozicije:

P = „Mjesec je zelen“
Q = „Zemlja se okreće“

Onda možemo i njih kombinirati, kao i u samom jeziku, tvoreći kompleksnije izjave. Primjer:

P ∧ Q = „Mjesec je zelen i Zemlja se okreće“

Kao što vidimo, „∧“ je znak koji ovdje služi kao „i“, te se ta operacija naziva **konjukcija.** Konjukcija je operacija čiji **rezultat može biti točan ako su obje tvrdnje točne**, za ostale slučajeve može biti samo netočna. U ovom našem slučaju možemo se složiti da se Zemlja okreće, ali ne da je i mjesec zelen, stoga je ta izjava netočna.
Isto tako, imamo i druge operacije, tj. druge veznike. Pogledajmo ovaj primjer:

P ∨ Q = „Mjesec je zelen ili Zemlja se okreće“

Vidimo sada da imamo znak „∨“ koji služi kao „ili“ te se naziva **disjunkcija.** Disjunkcija je operacija čiji **rezultat može biti točan ako je barem jedna od tvrdnji točna**. U jeziku često koristimo isključivo (*eng. exclusive*) ili kao što je „Ili pojedi ručak ili se nećeš maknuti sa ovog stola“ – ovdje ili služi kako bi naglasio uvjetovanje, dok je „ili“ koji mi koristimo u logici uključivo (*eng. inclusive*) ili. To znači da je onda ova naša tvrdnja točna, jer iako je netočno da je mjesec zelen, Zemlja se okreće, pa je stoga čitava tvrdnja točna.
Također, postoji i operacija za negaciju:

P = „Mjesec je zelen“
¬P = „Mjesec nije zelen“

Kao što vidimo, negacija (*eng. negation*) označena simbolom „¬“ negira trenutnu vrijednost unutar tvrdnje, propozicije.
Isto tako, možemo vidjeti još jedan prilično bitan veznik, a to je uvjetovanje, ili logička implikacija (*eng. implication*).

P = „Ako kupiš mlijeko“
Q = „Piti ćeš mlijeko“
P ⇒ Q = „Ako kupiš mlijeko onda ćeš moći piti mlijeko“

Taj veznik služi kao što služi IF … THEN uvjet u programiranju, iako ima malo neintuitivno ponašanje – **izjava je netočna samo onda kada je P točan a Q netočan**.

Pošto možemo reći da su to jako osiromašene osnove propozicijske logike, pogledajmo što još imamo u predikatnoj logici, odnosno logici na kojoj ćemo se mi zadržati, iako nije ona sama potrebna za razumijevanje formalnih dokaza, logici prvog reda.
Logika prvog reda dodaje kvantifikatore (*eng. quantifiers*) i bikondicional koji nije toliko bitan. **Predikat je funkcija koja vraća točno ili netočno**. Primjer:

∃x(dobar_programer(x)) = „Postoji programer koji dobro programira“

Simbol „**∃**“ znači da postoji **minimalno jedna vrijednost za navedeni predikat**. Drugim riječima da postoji minimalno jedan programer koji dobro programira. Takav kvantifikator naziva se *egzistencijalni kvantifikator*.

∀x(lijen_programer(x)) = „Svi programeri su lijeni“

Simbol „**∀**“ znači da **vrijednost vrijedi za sve predikate**. Vrijedi za sve programere, svi programeri su lijeni. Takav kvantifikator naziva se *univerzalni kvantifikator*.

Logika drugog reda proširuje logiku prvog reda sa mogućnosti da se kvantifikacija može izvršavati i preko relacija/funkcija – pa samim time i predikata.

Logika višeg reda proširuje logiku drugog reda sa mogućnosti da se kvantificira preko kvantificiranih relacija/funkcija. Tek u logici drugog reda možemo koristiti matematičku indukciju, budući da onda indukciju možemo izraziti kao N-puta kvantificiranu funkciju.

Dakle, *logika prvog reda* kvantificira varijable individualnih vrijednosti, *logika drugog reda* kvantificira preko skupova, a *logika višeg reda* kvantificira skupove skupova.

3. COQ

COQ je program koji služi kao pomoć pri dokazivanju (*eng. proof assistant*). Njegova je jezgra jako mala, te je napisana u višem programskom jeziku "Gallina" koji je baziran na "Calculus of Inductive Contructions" koji je kombinacija logike višeg reda i jakog funkcijskog programskog jezika. "Gallina" je jako mali jezik, te u sebi niti ne sadržava definiciju jednakosti tipova. COQ omogućava formalnu specifikaciju i verifikaciju, kao i ekstrakciju postojećih programa koji su verificirani u neke jednostavnije programske jezike kao što su - OCaml i Haskell.

Sam COQ napisan je u OCaml-u, te je rezultat preko 30 godina istraživanja. Njegovi autori su Thierry Coquand and Gérard Huet, a pretpostavka je da je program nazvan po prezimenu svog autora - Coquand. Iako postoji više teorija po čemu je sam program nazvan.
Sam COQ počinje od pretpostavke da imamo nekakvu početnu izjavu koju želimo dokazati. Riječi poputa 'Lemma', 'Theorem', 'Proof', 'Conjecture' u biti su sinonimi unutar samog COQ-a, a čak i u samoj matematici imaju jako slična značenja. 'Lemma' je pomoćni dokaz koji služi da se izgrade veći dokazi. 'Theorem' ili 'Proof' je dokazana pretpostavka/izjava pomoću drugih teorema i aksioma. 'Conjecture' je teorija koja nije još dokazana, a pretpostavlja se da ju je moguće dokazati. 'Axiom' je temeljna pretpostavka koja se prihvaća kao istina - kao temeljna neupitna činjenica koja je istinita za "svijet" koji promatramo.

Dokazivanje u COQ-u sastoji se od primjena taktika. Taktike su načini na koje možemo pokazati COQ-u da je naš program verificiran ili validan. Taktike možemo zamisliti kao alate koje imamo kao korisnik COQ-a sa kojima COQ uvjeravamo da je dokaz točan. Primjer taktika je indukcija, transformacija izraza (*eng. rewrite*), destrukcija na osnovne tipove podataka, itd.

Primjer većeg projekta koji je napravljen u COQ-u jest "CompCert" projekt koji je pokušaj da se uvedu formalno dokazani programi na integriranu (*eng. embedded*) razinu - projekt[1] koji je formalno dokazao C kompajler i njegovo ponašanje. Time se izbjegavaju bugovi na razini samog kompajlera i time se omogućuje daljnje korektno ponašanje razvojnih alata baziranih na toj platformi. Ideja iza toga je da čak i da imate formalno verificiran program koji trebate negdje instalirati, kada ga kompajlirate možete razviti bug u svojoj aplikaciji a da za to niste ni krivi. Ovim projektom taj se slučaj izbjegava, budući da je pokrivena verifikacija cijelog razvojnog alata.

[1] http://compcert.inria.fr/compcert-C.html

3.1. Osnove COQ-a

Kao što je već spomenuto, postoje dva „pogleda“ na sam program. Prvo imamo dio u kojem pišemo funkcionalni programski jezik koji se zove Galina, te možemo programirati samo sa tim dijelom. Ono što COQ čini zanimljivim je i to da se taj programski jezik može provjeravati formalnom verifikacijom – drugim riječima da možemo matematički dokazati da taj program uvijek radi točno kako smo mi zamislili, što je drugi „pogled“ na sam program. Sam program je niz instrukcija koje su odijeljene sa točkom.
Uzmimo jedan primjer koji će se kasnije pojaviti u primjerima formalnih dokaza.

```
Theorem plus_0_r : forall n:nat, n + 0 = n.
Proof.
 intros n. induction n as [| n'].
 reflexivity.
 simpl. rewrite → IHn'. reflexivity.
Qed.
```

Ovakav dokaz možemo razbiti u dva dijela:

- definiciju ili samo tvrdnju koju imamo o programu
- dokaz te tvrdnje

Definicija je ovaj prvi dio koda.

```
Theorem plus_0_r : forall n:nat, n + 0 = n.
```

U kojem piše i mi možemo pročitati da teorem koji se zove „plus_0_r“ sadržava sljedeću definiciju – za svaki (*eng. for all*) *n* koji je prirodni broj (*eng. nat - natural*), tvrdimo da kada taj prirodni broj zbrojimo sa nulom dobijemo taj isti *n* prirodni broj kao rezultat. „Forall“ je ključna riječ koja se još može i prikazati simbolom ∀ jer je to zapravo univerzalni

kvantifikator. Drugim riječima kada napišemo „forall“ prije same varijable tvrdimo da to pravilo vrijedi za sve varijable tog tipa. Konkretno, ovdje tvrdimo da je tvrdnja $n + 0 = n$ važeća za **svaki** prirodni broj. Ako bismo to prikazali u matematici jedan od oblika koji bi to lijepo pokazivao, bio bi sljedeći:

$$\forall\, n\ (n \in N, n + 0 = n)$$

Kada ovaj prvi dio koda učitamo u COQ, on prikaže da postoji jedan cilj koji je potrebno dokazati (*eng. subgoals*), što je sada obaveza drugog dijela programa.

Dokaz ovakve tvrdnje/cilja je ovaj drugi dio koda koji sadržava formalni dokaz. Dokaz tvrdnje počinje sa korištenjem ključne riječi dokaz (*eng. proof*).

```
Proof.
 intros n. induction n as [| n'].
 reflexivity.
 simpl. rewrite → IHn'. reflexivity.
Qed.
```

Dokaz završava sa ključnom riječi „Qed“, što je latinska kratica za „ono što je trebalo biti dokazano“ (*lat. quod erat demonstrandum*), što je inače konvencija, iako ne jedina, koja se koristi u matematičkim dokazima kada je dokaz završen.

Ono što se nalazi između prve i zadnje ključne riječi formalni je dokaz, koji, korak po korak, dokazuje početnu tvrdnju iliti teorem koji smo postavili.

Kako učitamo sljedeću instrukciju, budući da su, kao što sam napisao instrukcije odijeljene točkom, dobijemo kvantificiranu varijablu na raspolaganje – „intros n.“ pomakne varijablu iz definicije sa kvantifikatorom u kontekst trenutne pretpostavke.
Nakon toga, koristimo taktiku „induction“. Ta taktika je, kako naziv kaže, zapravo matematička indukcija nad varijablom n (*eng. induction*). Kada učitamo tu instrukciju dobijemo dva pod cilja:

1. $0 + 0 = 0$
2. $S\,n' + 0 = S\,n'$

COQ automatski kreće sa učitavanjem prvog pod cilja, te taj cilj postaje glavni cilj koji trebamo dokazati. Ali vidimo da je dokaz prvog slučaja trivijalan.

Tako da sljedeća izjava „reflexivity“ završava taj pod cilj – „reflexivity“ znači da jedna i druga strana cilja imaju identičnu vrijednost. Pošto je 0 + 0 = 0, onda je COQ dovoljno pametan da zaključi „pa da, ovo stvarno je isto“. Time smo dokazali prvi pod cilj.

Tada COQ automatski kreće na drugi pod cilj. Drugi pod cilj neće biti baš jasan dok čitatelj na pročita rad dalje, ali sljedećih par instrukcija/taktika se brine da dokaže taj jednostavni pod cilj. Nakon što je drugi pod cilj dokazan, onda je dokazano sve i onda COQ primijeti instrukciju kraja dokaza, „qed“ i provjeri i kaže „stvarno je sve dokazano, teorem je potvrđen“.

Nakon toga imamo teorem, dok možemo reći da je prije nego što je taj teorem bio dokazan on bio hipoteza (pretpostavka).

3.2. Peano brojevi

Još nam je jedna stvar potrebna prije nego krenemo na stvarni primjer. Kako smo za našu domenu odabrali prirodne brojeve, koji su, u odnosu na druge domene koje smo mogli odabrati, prilično jednostavni, potrebno je pokazati njihov temelj u formalnoj matematici.
Brojevi koje ćemo koristiti da bismo reprezentirali prirodne brojeve nazivaju se Peano brojevi.
Peano brojevi su reprezentacija prirodnih brojeva, N. Zanimljivost vezana za Peano brojeve jest što sam Peano nije napravio velike doprinose. Postoji pet aksioma[2] koji se iskazuju na relativno jednostavan način. S je zapravo funkcija „sljedbenika“ prirodnog broja koji ulazi u funkciju. Ako pošaljemo 3 u funkciju, S(3) je 4.
Peano-vi aksiomi (osnovni):

1. 0 je prirodan broj.
2. Za svaki prirodni broj n, S(n) je prirodni broj
3. Za sve prirodne brojeve m i n, m = n akko je S(m) = S(n). S je injekcija
4. Za svaki prirodni broj S(n) = 0 nije točno. Drugim riječima ne postoji prirodan broj čiji je sljedbenik 0
5. Ako je φ unarni predikat (pripadnosti) tako da:
   a) φ (0) je istina, i
   b) za svaki prirodni broj φ (n) je istina, onda je φ(S(n)) istina, i onda φ (n) vrijedi za svaki prirodni broj

Brojevi bi bili:
0 = 0
1 = S 0
2 = S (S 0)
3 = S (S (S 0))
Način na koji je autor rada sebi olakšao čitanje ovakve notacije jest da svaki puta kada vidi slovo S doda 1. Tada ‘S (S (S 0))’ postaje poslije - poslije - poslije nule - ili 0 + 1 + 1 + 1 = 3.

Budući da COQ već sadržava te definicije, olakšan nam je posao te Coq može “razumijeti” prirodne brojeve te ih može prebaciti u peano format i obrnuto.

[2] https://hr.wikipedia.org/wiki/Peanovi_aksiomi

4. Primjer formalnog dokaza

Za formalnu verifikaciju koristiti ćemo COQ, koji je opisan u prijašnjem poglavlju.
Uzmimo prvo definiciju prirodnih brojeva koju ćemo koristiti u listi i pogledajmo kako bismo je napravili i kasnije verificirali.

```
Inductive nat : Type :=
 | O : nat
 | S : nat → nat.
```

Prvo, ključna riječ ‘Inductive’ definira ‘Type’, tip nat kao induktivan. Kao što vidimo po definiciji, prirodni brojevi mogu biti sastavljeni na dva načina. Mogu biti 0, što je jasno samo po sebi, ostavljajući tako tip nat. I mogu biti definirani kao S 0, što bi značilo sljedbenik (engl. successor) 0 - što je 1, ostavljajući tako tip podatka nat → nat, što možemo čitati kao funkciju koja za prirodni broj koji mi damo ona vraća drugi prirodni broj.
Nema potrebe da dodajemo funkcije za baratanje prirodnim brojevima, ali mogli bismo pogledati jedan jako jednostavni primjer funkcije i kako ćemo je dokazati, zajedno sa samim kodom kojeg ćemo generirati.

```
Fixpoint plus (n : nat) (m : nat) : nat :=
 match n with
   | O ⇒ m
   | S n' ⇒ S (plus n' m)
 end.
```

E, sada kada imamo definiranu operaciju plus, za dva prirodna broja, hajdemo probati vidjeti da li radi ispravno. Kao prvo, možemo probati sa primjerima.

```
Example test_plus_1 in (plus 5 7) = 12.
 reflexivity. Qed.
```

Dobro. Jednostavni test smo napravili. Ništa različito nego što bismo napravili sa drugim programskim jezicima. Hajdemo sada otkriti zašto je formalna verifikacija superiorna nad programskim testovima te vidimo da li možemo dokazati komutativnost operacije za sve prirodne brojeve. Vi zamislite prirodni broj, ovo vrijedi za njega. Prirodnih brojeva ima beskonačno te da idemo enumerirati pojedinačne testove za svaki broj bilo bi nam potrebno - beskonačno testova. Ovo je puno lakše. I puno brže. Beskonačno puta brže. Ono što je važno imati na umu jest da mi definiramo da je funkcija „plus“ zapravo „+“, iako tu definiciju nemamo. Tako da je sljedeći teorem zapravo teorem „plus n 0 = n“.

```
Theorem plus_0_r : forall n:nat, n + 0 = n.
Proof.
 intros n. induction n as [| n'].
 reflexivity.
 simpl. rewrite → IHn'. reflexivity.
Qed.
```

Prvo, za razliku od prijašnjeg primjera koristili smo ključnu riječ Theorem umjesto Example. Ta razlika pitanje je stila; ključne riječi Example i Theorem znače identično.
Drugo, dodali smo kvantifikator (pogledati predikatnu logiku, pa “naviše” - logika prvog reda, ...) ∀ n:nat, tako da naš teorem govori o svim prirodnim brojevima n. Da bismo dokazali

teorem u ovoj formi, trebali bi pretpostaviti postojanje prirodnog broja n. To se postiže sa linijom intros n, koja pomakne kvantifikator iz definicija u kontekst trenutne pretpostavke. Drugim riječima, započinjemo dokaz tako da kažemo - "pretpostavimo da je n neki prirodni broj".

Ključne riječi intros, simpl, i reflexivity primjeri su taktika. Taktika je nardba koja se koristi između Proof i Qed kako bi rekla COQ-u kako da provjeri točnost tvrdnje koje imamo. Generirati ćemo kod iz Coq programa te ga pogledati malo. Dodajući sljedeće dvije linije moći ćemo generirati kod (u Ocaml, Haskell, ...). Budući da je generiranje u Ocaml najzrelije, fokusirati ćemo se na njega.

```
Extraction Language Ocaml.
Extraction "test.ml" plus.
```

Te pogledajmo implementaciju (specifikacija sada nije bitna) – test.ml:

```
type nat =
| O
| S of nat

(** val plus : nat -> nat -> nat **)

let rec plus n m =
 match n with
 | O -> m
 | S n' -> S (plus n' m)
```

I sada ovaj kod koji imamo možemo koristiti. Teorem „plus_0_r“ je verificirao da kada koristimo funkciju „plus“ za dva prirodna broja ako ikad proslijedimo nulu u funkciju „plus“, dobiti ćemo originalni broj – onaj koji smo poslali sa nulom. Tada možemo mirno spavati znajući da ako ikada dodamo nulu na neki prirodni broj, dobiti ćemo nazad taj prirodni broj - možemo reći da smo dokazali funkciju identiteta.

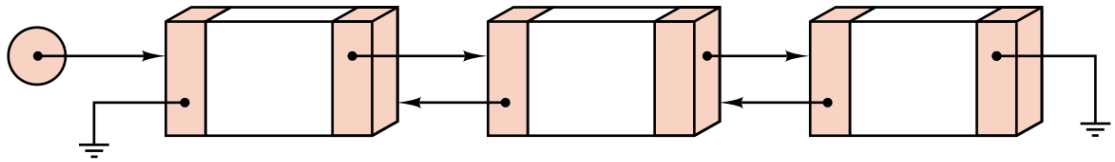
4.1. Podatkovna struktura, lista

Kao što sam napomeno, potrebno je verificirati program. Programi koje koristimo često koriste zajedničke algoritme i zajedničke podatkovne strukture. Budući da algoritam opisuje nekakav postupak koji je kompletno definiran, korak po korak, a podatkovna struktura čuva podatke, ili sadržava podatke, možemo reći da se svi programi sastoje od dvije stvari - podataka i operacija nad tim podacima, ili podatkovnih struktura i algoritama koji operiraju nad tim podatkovnim strukturama.

U ovom radu fokusirati ćemo se, radi jednostavnosti, na podatkovnu strukturu - listu. Bilo bi suludo pokušati napraviti čitav program, čitatelji će se sami uvjeriti koliko je truda i vremena potrebno za definirati jednostavne stvari kao što je lista. Što je zapravo lista? I kakvu ćemo mi listu raditi?

Lista (ili sekvenca) je skupina određenih vrijednosti povezana određenim redosljedom, u kojoj se pojedina vrijednost može pojaviti više puta. Možemo to zamisliti kao nekakvu gusjenicu gdje svaki njen dio sadržava vrijednost.

Slika 2*Struktura liste*



Izvor: Kruse L. R. (2000): *Data structures and Program Design in C*, Prentice-Hall, Simon & Schuster, str. 227

Čemu takva struktura može služiti? Pa za početak, možemo pohraniti rezultate za koje nam je potreban redoslijed. Npr. lista rezultata trkaća. Riječ koju želimo imati spremljenu u memoriji.

Znači, lista bi bila konačan niz elemenata povezanih zajedno. Lista je homogena ako sadržava isti tip elementa, te heterogena ako sadržava različite vrste podataka. Pretpostavimo da je lista koju mi radimo homogena i da može sadržavati samo prirodne brojeve.

Listu možemo promatrati kao skup[3] koji dopušta duplikate i u kojem treba paziti na poredak – moguće je listu direktno usporediti sa n-torkom, uređenim skupom koji ima poredak, sa time da bi još trebao dopuštati duplikate da bi bio lista. Svaki skup ima svoju duljinu pa je ima i lista. Svaki skup ima elemente, ako je duljina veća od 0, pa je tako ima i lista. Isto tako, elementi liste imaju svoj redoslijed, te samim time svoj indeks.

Elementi liste mogu se mijenjati - mogu se dodavati i brisati u svakom trenutku.

Uzmimo primjer za listu - zamislimo riječ “DOKAZ”.

Možemo reći da je “DOKAZ” skupina znakova određenog tipa i redoslijeda. Konkretno, znakovi su [‘D’, ‘O’, ‘K’, ‘A’, ‘Z’]. Svaki od tih znakova tipa je *Char*, te bismo mogli reći da je takva skupina znakova tipa [Char] - lista Char elementa. Kao što se to definira u nekim funkcijskim programskim jezicima, tip sinonim [Char] je *String*. Drugim riječima, tip RIJEČ je u biti skupina SLOVA.

Neke tipične operacije nad listom uključuju:

- brisanje elemenata liste
- dodavanje elemenata liste
- duljina liste - ili jednostavno, broj elemenata u listi

[3] https://en.wikipedia.org/wiki/Ordered_pair#Defining_the_ordered_pair_using_set_theory

4.2. Lista, formalni dokaz

Vratimo se sada na listu. Naša lista onda sadržava prirodne brojeve te ju možemo zamisliti kao listu koja je ograničena isključivo na prirodne brojeve. Pogledajmo njenu definiciju:

```
Inductive natlist : Type :=
 | nil : natlist
 | cons : nat → natlist → natlist.
```

Primjer jedne liste koja sadržava podatke (1, 2, 3) je ovo.

```
Definition mylist := cons 1 (cons 2 (cons 3 nil)).
```

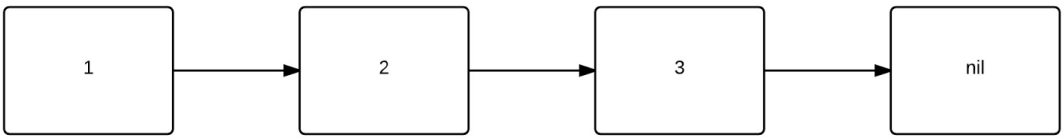
„Definition“ (*hr. definicija*) je ključna riječ koja predstavlja definiciju varijable, kao što naziv kaže. Slično je kao što je „let“ u funkcijskim programskim jezicima kao što su OCaml i Haskell. „Definition“ možemo čitati kao „dodijeli vrijednosti sa desne strane u varijablu sa lijeve strane (mylist)“.

Drugi dio priče definitivno je (funkcija) „cons“, kraće od „construction“ (*hr. sastavljanje*), koja prima dva parametra:

- prirodni broj
- listu

Kao što smo vidjeli u definiciji „natlist“, lista može biti prazna (*eng. nil*) ili sadržavati drugu listu, na neki način ju sastavljajući (*eng. construction*).
A njen izgled, izgled liste koja sadržava tri elementa (1, 2, 3) bi bio ovakav:

Slika 3*Skica liste*



Izvor: Izradio autor

Pogledajmo neku jednostavnu funkciju sa listom. Recimo duljinu liste i funkciju za spajanje dvaju listi.

```
Fixpoint length (l:natlist) : nat :=
 match l with
 | nil ⇒ O
 | h :: t ⇒ S (length t)
 end.

Fixpoint app (l1 l2 : natlist) : natlist :=
 match l1 with
 | nil ⇒ l2
 | h :: t ⇒ h :: (app t l2)
 end.
```

„Fixpoint“ je ključna riječ koja definira rekurzivnu funkciju koja se mora završiti. Drugim riječima osiguravamo da funkcija ne ostane u „petlji“. Isto tako, kao što možemo primijetiti, imamo neki novi simbol „::“ – taj simbol biti će poznat svima koji su se susreli sa funkcijskim programskim jezicima kao što je OCaml ili Haskell – taj operator zapravo je zamjena za prije definiranu funkciju „cons“.
I sada, kada imamo jednostavnu definiciju liste, možemo probati dokazati razna svojstva ovih dviju funkcija. Recimo dokažimo da je dužina dvije liste jednaka dužini spojene liste sastavljene od te dvije.

```
Theorem len_eq_app: forall n m :natlist,
 length(n ++ m) = length n + length m.
Proof.
 intros n m. induction n as [| n'].
 reflexivity.
 simpl. rewrite → IHn'. reflexivity.
Qed.
```

Novost koju vidimo na prijašnjem kodu je „++“. Kao što je „+“ kratica za poziv funkcije „plus“, tako je i „++“ kratica za poziv funkcije „append“ (*hr. spojiti*). Drugim riječima, ta funkcija dodaje jednu listu na drugu, spajajući ih u jednu novu listu.
I onda imamo garanciju da ćemo uvijek, neovisno koje dvije liste pokušamo spojiti, imati da je duljina spojene liste zbroj pojedinačnih. Pogledajmo kako to izgleda u samome kodu.

```
type nat =
| O
```

```
| S of nat


type natlist =
| Nil
| Cons of nat * natlist


(** val length : natlist -> nat **)


let rec length = function
| Nil -> O
| Cons (h, t) -> S (length t)


(** val app : natlist -> natlist -> natlist **)


let rec app l1 l2 =
 match l1 with
 | Nil -> l2
 | Cons (h, t) -> Cons (h, (app t l2))
```

Sada imamo jednostavne funkcije koje mogu raditi jednostavne stvari ali garantirati da one rade ispravno. Ostale funkcije trebalo bi napisati, a bile bi to neke funkcije koje ubacuju i izbacuju elemente sa liste. Iako već za ubacivanje možemo koristiti „++“ iliti „append“.

5. Zaključak

Formalni dokazi trebaju se koristiti češće. Prolaskom vremena sve više se javljaju velike greške u industriji u kojoj koristimo računala, a ta je industrija svake godine sve veća i veća. Samim time povećavaju se i mogućnosti fatalnih grešaka koje je potrebno eliminirati. Naravno, ako vam je potrebna web stranica, ili Internet trgovina s malom količinom prometa, onda "normalno" programiranje ima prednost - da bismo formalno dokazali tako velike programe potrebno je jako puno vremena - a samim time i novca. Ali koji puta je potrebna kvaliteta.

Popis literature

KNJIGE:

1. Morash, Ronald P. (1987): *Bridge to Abstract Mathematics: Mathematical Proof and Structures*
2. Bornat, R. (2005): *Proof and Disproof in Formal Logic: An Introduction for Programmers*
3. Manger R. (2014): *Strukture podataka i algoritmi*

INTERNETSKI IZVORI:

1. Scott W. Ambler (2003-2014): Examining the Agile Cost of Change Curve;
(dostupno na: http://www.agilemodeling.com/essays/costOfChange.htm)
2. Paul Bordeaux (2009): Top Ten Most Infamous Software Bugs Of All Time
(dostupno na: https://www.sundoginteractive.com/blog/top-ten-most-infamous-software-bugs-of-all-time/)
3. Benjamin C. Pierce (2015): Software Foundations
(dostupno na: http://www.cis.upenn.edu/~bcpierce/sf/current/index.html)
3. Robert Manger (2013): Softversko inženjerstvo, skripta
(dostupno na: http://web.studenti.math.pmf.unizg.hr/~manger/si/SI-skripta-v2.pdf)

Popis slika

Slika 1 Traditional cost of change curve 2
Slika 2 Struktura liste..... 24
Slika 3 Skica liste..... 26

Student: Kristijan Šarić
JMBAG: 0145022678, izvanredni student
Studijski smjer: Informatika
Predmet: Programiranje
Mentor: Prof. dr. sc. Krunoslav Puljić

FORMALNI DOKAZI U PROGRAMIRANJU

Sažetak završnog rada

Programiranje u moderno doba znači programiranje sa velikom količinom kompleksnosti. Velika kompleksnost znači veliku mogućnost greške. Programiranje u moderno doba znači veliku mogućnost greške. U ovom radu navodimo slučajeve u kojima bi se trebali koristiti formalni dokazi, odnosno verifikacija napisanog programa, kako bi program mogao raditi bez greške. Time se postiže velika sigurnost u radu kritičnih dijelova sustava i samim time, sigurnost za sve koji se takvom sustavu izlažu.

Ključne riječi: programiranje, greške, formalni dokazi, verifikacija

Student: Kristijan Šarić
Student identification code: 0145022678, part-time student
Field of study: Informatics
Subject: Programming
Mentor: Krunoslav Puljić, Ph.D. Associate Professor

FORMAL PROOFS IN PROGRAMMING

Summary of thesis

Programming in the modern era means programming with a large amount of complexity. Large amount of complexity means a significant possibility of error. Programming in the modern era means a significant possibility of error. In this paper we present the cases when the use of formal methods, and verification of the written program, so the program could operate without failure should be used. This achieves a high degree of certainty in the operation of the critical parts of the system and therefore, the safety of all who are exposed to a such system.

Key words: programming, errors, formal methods, verification