UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Ana Đorđević

AUTOMATSKO GENERISANJE TEST PRIMERA UZ POMOĆ STATIČKE ANALIZE I REŠAVAČA Z3

master rad

Beograd, 2017.

Mentor:

dr Milena Vujošević Janičić, docent
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

dr Filip Marić, vanredni profesor
Univerzitet u Beogradu, Matematički fakultet

dr Milan Banković, docent
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: _____

Mami i tati

Naslov master rada: Automatsko generisanje test primera uz pomoć statičke analize i rešavača Z3

Rezime:

Ključne reči: verifikacija softvera, testiranje softvera, SMT rešavači, Z3 rešavač, automatsko pronalaženje grešaka u programu, računarstvo

Sadržaj

Glava 1

Uvod

Glava 2

Testiranje

2.1 Osnove testiranja

Testiranje predstavlja važan deo životnog ciklusa razvoja softvera. Omogućava lakše uočavanje grešaka i propusta nastalih prilikom imple- mentacije. Pored toga ono predstavlja i jedan od načina specifikacije problema. Kako i najmanji problem može uništiti uložen trud, u slučaju obimnih projekata nikada nije dovoljno testiranja. S porastom složenosti projekta raste i značaj testiranja i provera softvera kako bi se izbegli is- hodi koju mogu da unište ceo projekat.

Kako se greške ne mogu izbeći, potrebno ih je što je moguće ranije ot- kriti kako bi njihovo otklanjanje bilo brže i jeftinije. Zbog prednosti koje se dobijaju najzastupljenije i trenutno najpopularnije metodologije razvoja promovišu paralelno pisanje testova i implementacije softvera. Ekstremno programiranje kao jedan od predstavnika agilnih metodologija posebno ističe razvoj voden testovima i pisanje testova prihvatljivosti.

Testiranje se sastoji od planiranja, gde se odreduje predmet, cilj i ra- zlog testiranja, zatim sledi dizajn testova koji odreduje kako se sprovodi testiranje na osnovu test primera. Treća faza je implementacija testova iza koje sledi izvršavanje testova radi provere funkcionalnosti sistema. Kao poslednja faza se definiše evaluacija testova koja uključuje validnost izvršavanja testa, analizu izlaza i pregled dobijenih rezultata.

Testiranje služi kao provera da li program ima predvidjenu funkcionalnost i da otkrije greške u programu pre nego što je pušten u upotrebu. Kada testiramo program treba da koristimo i nespecifične ulazne podatke.

Testiranje softvera predstavlja proces analize elementa softvera kako bi se utvr-

dila razlika između postojećeg stanja i zahteva i kako bi se ustanovile karakteristike softvera. Pouzdanost je sposobnost sistema ili njegove komponente da izvrši zahtevane funkcije pod definisanim uslovima u određenom vremenskom periodu.

Glava 3

Rešavač Z3

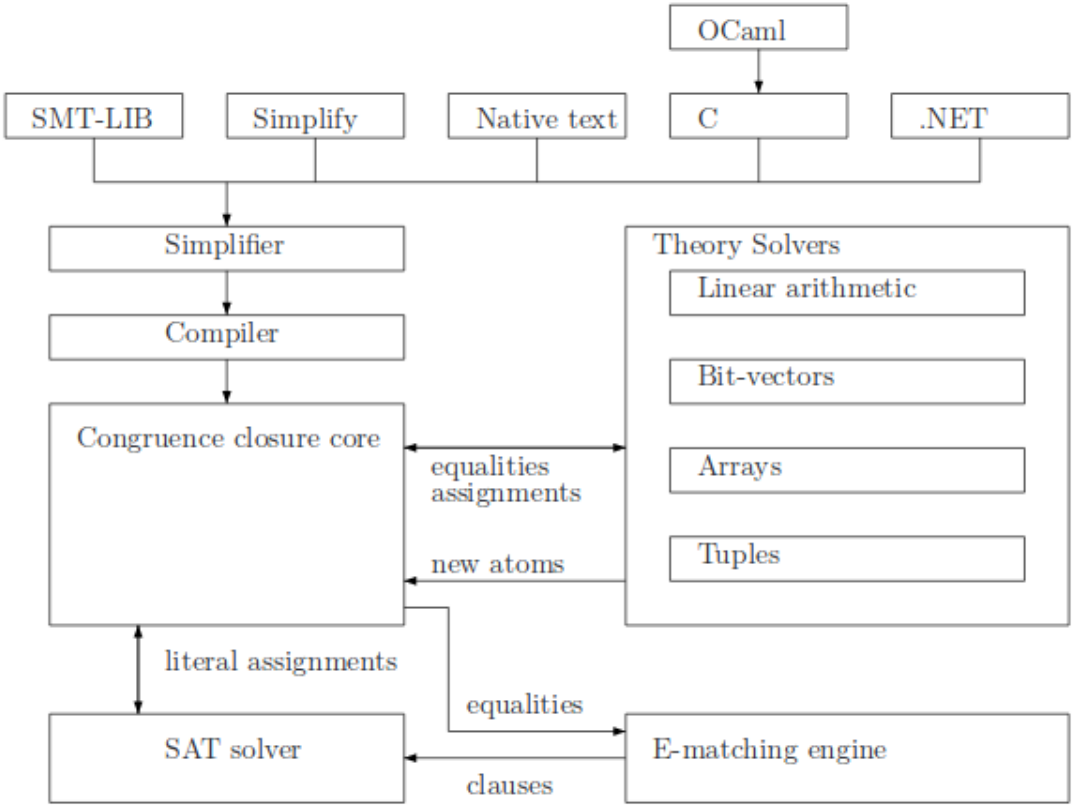
Sistemi za analizu i verifikaciju softvera su veoma kompleksni. Njihovu osnovu predstavlja komponenta koja koristi logičke formule za opisivanja stanja i transformacija između stanja sistema. Opisivanje stanja sistema često se svodi na proveravanje zadovoljivosti formula logike prvog reda. Proveravanje zadovoljivosti formula vrši se procedurama odlučivanja u odnosu na definisanu teoriju. Formalno, zadovoljivost u odnosu na teoriju (eng. *Satisfiability Modulo Theory*, skraćeno SMT) problem je odlučivanja zadovoljivosti u odnosu na osnovnu teoriju T opisanu u klasičnoj logici prvog reda sa jednakošću [1]. Alati koji se koriste za rešavanje ovog problema nazivaju se SMT rešavači.

Jedan od najpoznatijih SMT rešavača je rešavač Z3 kompanije Microsoft koji se koristi za proveru zadovoljivosti logičkih formula u velikom broju teorija [7]. Z3 se najčešče koristi kao podrška drugim alatima, pre svega alatima za analizu i verifikaciju softvera. Pripada grupi SMT rešavača sa integrisanim procedurama odlučivanja.

U ovoj glavi biće opisane osnove rešavača Z3 u delu 3.1. U delu 3.2 biće opisane najvažnije teorije uključujući teoriju neinterpretiranih funkcija, teoriju linearne aritmetike, teoriju nelinearne aritmetike, teoriju bitvektora i teoriju nizova. U delu 3.3 opisan je format za komunikaciju sa Z3 rešavačem korišćenjem SMT-LIB standarda. Pored toga, rešavač Z3 nudi interfejse za direktnu komunikaciju sa programskim jezicima C, C++, Java i Python. U delu 3.4 opisan je interfejs rešavača Z3 za komunikaciju sa programskim jezikom C++. Više materijala o podržanim interfejsima za programske jezike C, C++, Java i Python može se pronaći u literaturi [8].

3.1 Osnove rešavača

Problem zadovoljivosti (eng. *Satisfiability problem*, skraćeno SAT) problem je odlučivanja da li za iskaznu formulu u konjunktivnoj normalnoj formi postoji valuacija u kojoj su sve njene klauze tačne [2]. Rešavači koji se koriste za rešavanje ovog problema nazivaju se SAT rešavači. Rešavač Z3 integriše SAT rešavač zasnovan na savremenoj DPLL proceduri i veliki broj teorija. Implementiran je u programskom jeziku C++. Šematski prikaz arhitekture rešavača [7] prikazan je na slici 3.1.



Slika 3.1: Arhitektura rešavača Z3

Formule prosleđene rešavaču se najpre procesiraju upotrebom simplifikacije. Simplifikacija primenjuje algebarska pravila redukcije kao što je `p` $\wedge$ `true` $\vdash$ `p`. Pored toga, ovim procesom se vrše odgovarajuće zamene kao što je `x=4` $\wedge$ `q(x)` $\vdash$ `x=4` $\wedge$ `q(4)`. Nakon simplifikacije, kompajler formira apstraktno sintaksno stablo formula čiji su čvorovi simplifikovane formule (klauze). Zatim se jezgru kongruentnog zatvorenja (eng. *Congruence closure core*) prosleđuje apstraktno sintaksno stablo.

Jezgro kongruentnog zatvorenja komunicira sa SAT rešavačem koji određuje istinitosnu vrednost klauza.

Glavni gradivni blokovi formula su konstante, funkcije i relacije. Konstante su specijalan slučaj funkcija bez parametara. Svaka konstanta je određene sorte. Sorta odgovara tipu u programskim jezicima. Relacije su funkcije koje vraćaju povratnu vrednost tipa Boolean. Funkcije mogu uzimati argumente tipa Boolean pa se na taj način relacije mogu koristiti kao argumenti funkcija.

Formula F je validna ako je vrednost valuacije *true* za bilo koje interpretacije funkcija i konstantnih simbola. Formula F je zadovoljiva ukoliko postoji bar jedna valuacija u kojoj je formula tačna. Da bismo odredili da li je formula F validna, rešavač Z3 proverava da li je formula $\neg F$ zadovoljiva. Ukoliko je negacija formule nezadovoljiva, onda je polazna formula validna.

3.2 Teorije

Teorije rešavača Z3 su opisane u okviru višesortne logike prvog reda sa jednakošću. Definisanjem specifične teorije, uvode se restrikcije pri definisanju formula kao i podržanih relacija i operatora koje se nad njima primenjuju. Na taj način, specijalizovane metode u odgovarajućoj teoriji mogu biti efikasnije implementirane u poređenju sa opštim slučajem. U nastavku će biti opisane teorija neinterpretiranih funkcija, teorija linearne aritmetike, teorija nelinearne aritmetike, teorija bitvektora i teorija nizova.

Teorija neinterpretiranih funkcija

Teorije obično određuju interpretaciju funkcijskih simbola. Teorija koja ne zadaje nikakva ograničenja za funkcijske simbole naziva se teorija neinterpretiranih funkcija (eng. *Theory of Equality with Uninterpreted Functions*, skraćeno EUF).

Kod rešavača Z3, funkcije i konstantni simboli su neinterpretirani. Ovo je kontrast u odnosu na funkcije odgovarajućih teorija. Funkcija + ima standardnu interpretaciju u teoriji aritmetike. Neinterpretirane funkcije i konstante su maksimalno fleksibilne i dozvoljavaju bilo koju interpretaciju koja je u skladu sa ograničenjima. Za razliku od programskih jezika, funkcije logike prvog reda su totalne, tj. definisane su za sve vrednosti ulaznih parametara. Na primer, deljenje 0 je dozvoljeno, ali nije specifikovano šta ono predstavlja. Teorija neinterpretiranih funkcija je odlu-

čiva i postoji procedura odlučivanja polinomijalne vremenske složenosti. Jedna od procedura odlučivanja za ovu teoriju zasniva se na primeni algoritma Nelson-Open (eng. *Nelson-Open algorithm*). O ovom algoritmu može se više naći u literaturi [6].

Teorija linearne aritmetike

Rešavač Z3 sadrži procedure odlučivanja za linearnu aritmetiku nad celobrojnim i realnim brojevima. Dodatni materijali o procedurama odlučivanja linearne aritmetike dostupni su u literaturi [5].

U okviru celobrojne linearne aritmetike, podržani funkcijski simboli su +, - i * pri čemu je kod množenja drugi operand konstanta. Nad funkcijskim simbolima, čiji su specijalni slučajevi konstante mogu se primenjivati relacijski operatori <, <=, > i >=.

U okviru realne linearne aritmetike, podržani funkcijski simboli su +, - i * pri čemu je kod operacije množenja drugi operand konstanta. Pored ovih podržane su operacije div i mod, uz uslov da je drugi operand konstanta različita od 0. Nad funkcijskim simbolima, čiji su specijalni slučajevi konstante mogu se primenjivati relacijski operatori <, <=, > i >=.

Teorija nelinearne aritmetike

Formula predstavlja formulu nelinearne aritmetike ako je oblika (* t s), pri čemu t i s nisu linearnog oblika. Nelinearna celobrojna aritmetika je neodlučiva, tj. ne postoji procedura koja za proizvoljnu formulu vraća zadovoljivost ili nezadovoljivost. U najvećem broju slučajeva, Z3 vraća nepoznat rezultat. Za nelinearne probleme, rešavač Z3 koristi posebne metode odlučivanja zasnovane na Grebnerovim bazama.

Teorija bitvektora

Rešavač Z3 podržava bitvektore proizvoljne dužine. `(_ BitVec n)` je sorta bitvektora čija je dužina `n`. Bitvektor literali se mogu definisati koristeći binarnu, decimalnu ili heksadecimalnu notaciju. U binarnom i heksadecimalnom slučaju, veličina bitvektora je određena brojem karaktera. Na primer, literal *#b010* u binarnom formatu je bitvektor dužine 3. Kako konstanta *a* u heksadecimalnom formatu odgovara vrednosti 10, literal *#x0a* je bitvektor veličine 10. Veličina bitvektora mora biti specifikovana u decimalnom formatu. Na primer, reprezentacija *(_ bv10*

32) je bitvektor dužine 32 sa vrednošću 10. Podrazumevano, Z3 predstavlja bitvektore u heksadecimalnom formatu ukoliko je dužina bitvektora umnožak broja 4 a u suprotnom u binarnom formatu. Bitvektor literali mogu biti reprezentovani u decimalnom formatu. Više materijala o procedurama odlučivanja za teoriju bitvektora može se naći u literaturi [3].

Pri korišćenju operatora nad bitvektorima, mora se eksplicitno navesti tip operatora. Zapravo, za svaki operator podržane su dve varijante za rad sa označenenim i neoznačenim operandima. Ovo je kontrast u odnosu na programske jezike u kojima kompajler na osnovu argumenata implicitno određuje tip operacije (označena ili neoznačena varijanta).

U skladu sa prethodno navedenom činjenicom, teorija bitvektora ima na raspolaganju različite verzije aritmetičkih operacija za označene i neoznačene operande. Za rad sa bitvektorima od aritmetičkih operacija definisane su operacije sabiranja, oduzimanja, određivanje negacije (zapisivanja broja u komplementu invertovanjem svih bitova polaznog broja), množenja, izračunavanja modula pri deljenju, šiftovanje u levo kao i označeno i neoznačeno šiftovanje u desno. Podržane su sledeće logičke operacije: disjunkcija, konjunkcija, unarna negacija, negacija konjunkcije i negacija disjunkcije. Definisane su različite relacije nad bitvektorima kao što su <=, <, >=, >.

Teorija nizova

Osnovnu teoriju nizova karakterišu `select` i `store` naredbe. Funkcijom `(select a i)` vraća se vrednost na poziciji i u nizu a, dok se funkcijom `(store a i v)` formira novi niz, identičan nizu a pri čemu se na poziciji i nalazi vrednost v. Z3 sadrži procedure odlučivanja za osnovnu teoriju nizova. Dva niza su jednaka ukoliko su vrednosti svih elemenata na odgovarajućim pozicijama jednake.

Konstantni nizovi

Nizovi sa konstantnim vrednostima mogu se specifikovati koristeći `const` konstrukciju. Prilikom upotrebe const konstrukcije rešavač Z3 ne može da odluči kog tipa su elementi niza pa se on mora eksplicitno navesti. Interpretacija nizova je slična interpretaciji funkcija. Z3 koristi konstrukciju `(_ as-array f)` za određivanje interpretacije niza. Ako je niz a jednak rezultatu konstrukcije `(_ as-array f)`, tada za svaki indeks i, vrednost `(select a i)` odgovara vrednosti `(f i)`.

Primena map funkcije na nizove

Rešavač Z3 obezbeđuje primenu parametrizovane funkcije map na nizove. Funkcijom map omogućava se primena proizvoljnih funkcija na sve elemente niza.

Nad nizovima se mogu vršiti slične operacije kao i nad skupovima. Rešavač Z3 ima podršku za računanje unije, preseka i razlike dva niza. Ovi operatori se tumače na isti način kao i u teoriji skupova. Za nizove a i b, pomenuti operatori mogu se koristiti navođenjem funkcija:
`(union a b)` ; kreiranje unije dva niza kao skupa
`(intersect a b)` ; kreiranje preseka dva niza kao skupa
`(difference a b)` ; kreiranje razlike dva niza kao skupa

Komunikacija sa Z3 rešavačem može se ostvariti upotrebom SMT-LIB standarda i C++ interfejsa. Oba formata komunikacije imaju istu moć izražajnosti. Šta više, sintaksno se jako slično zapisuju. Ove sličnosti biće ilustrovane u nastavku.

3.3 Upotreba rešavača korišćenjem SMT-LIB standarda

Ulazni format rešavača Z3 je definisan SMT-LIB 2.0 standardom [4]. Standard definiše jezik logičkih formula čija se zadovoljivost proverava u odnosu na neku teoriju. Cilj standarda je pojednostavljivanje jezika logičkih formula povećavanjem njihove izražajnosti i fleksibilnosti kao i obezbeđivanje zajedničkog jezika za sve SMT rešavače.

Interno, Z3 održava stek korisnički definisanih formula i deklaracija. Formule i deklaracije jednim imenom nazivamo tvrđenjima. Komandom `push` kreira se novi opseg i čuva se trenutna veličina steka. Komandom `pop` uklanjaju se sva tvrđenja i deklaracije zadate posle push-a sa kojim se komanda uparuje. Komandom `assert` dodaje se formula na interni stek. Skup formula na steku je zadovoljiv ako postoji interpretacija u kojoj sve formule imaju istinitosnu vrednost tačno. Ova provera se vrši komandom `check-sat`. U slučaju zadovoljivosti vraća se `sat`, u slučaju nezadovoljivosti vraća se `unsat` a kada rešavač ne može da proceni da li je formula zadovoljiva ili ne vraća se `unknown`. Komandom `get-model` vraća se interpretacija u kojoj su sve formule na steku tačne.

Komandom `declare-const` deklariše se konstanta odgovarajuće sorte. Sorta može biti parametrizovana i u tom slučaju su specifikovana imena njenih parame-

tara. Specifikacija sorte vrši se naredbom `(define-sort [symbol] ([symbol]+)[sort])`. Komandom `declare-fun` deklariše se funkcija. U narednom primeru koristimo činjenicu da se validnost formule pokazuje ispitivanjem zadovoljivosti negirane formule.

Primer 1 *Dokazivanje de Morganovog zakona dualnosti ispitivanjem validnosti formule:* $\neg(a \wedge b) \Leftrightarrow (\neg a \vee \neg b)$ *tako što se kao ograničenje dodaje negacija polazne formule. Z3 pronalazi da je negacija formule nezadovoljiva, pa je polazna formula tačna u svim interpretacijama.*

Formula prosleđena rešavaču:	***Izlaz:***
(declare-const a Bool)	*unsat*
(declare-const b Bool)	
(define-fun demorgan () Bool	
(= (and a b) (not (or (not a) (not b))))	
)	
(assert (not demorgan))	
(check-sat)	
(get-model)	

Rešavač Z3 ima podršku za celobrojne i realne konstante. Prethodno pomenutom komandom `declare-const` deklarišu se celobrojne i realne konstante. Rešavač ne vrši automatsku konverziju između celobrojnih i realnih konstanti. Ukoliko je potrebno izvršiti ovakvu konverziju koristi se funkcija `to-real` za konvertovanje celobrojnih u realne vrednosti. Realne konstante treba da budu zapisane sa decimalnom tačkom.

Primer 2 *Naredni kod demonstrira upotrebu konstanti i funkcija. U primeru se deklariše konstanta a celobrojnog tipa i funkcija f sa parametrima tipa Int i Bool i povratnom vrednošću tipa Int. Zatim se dodaju odgovarajuća ograničenja za konstantu a i funkciju f korišćenjem operatora poređenja. Rešavač Z3 pronalazi da je ovo tvrđenje zadovoljivo i daje prikazani model.*

Formula prosleđena rešavaču:	***Izlaz:***
(declare-const a Int)	*sat*
(declare-fun f (Int Bool) Int)	*(model*
(assert (> a 10))	*(define-fun a () Int 11)*
(assert (< (f a true) 100))	*(define-fun f ((x!1 Int) (x!2 Bool)) Int*
(check-sat)	*(ite (and (= x!1 11) (= x!2 true)) 0 0))*
(get-model)	*)*

Primer 3 *Naredni kod ilustruje pronalaženje interpretacija celobrojnih i realnih konstanti. Interpretacija se svodi na pridruživanje brojeva svakoj konstanti. Ograničenja sadrže pomenute aritmetičke operatore. Rešavač vraća zadovoljivost tvrđenja i dobijeni model prikazujemo u nastavku.*

Formula prosleđena rešavaču:	***Izlaz:***
(declare-const a Int)	*sat*
(declare-const b Int)	*(model*
(declare-const c Int)	*(define-fun b () Int 0)*
(declare-const d Real)	*(define-fun a () Int 1)*
(declare-const e Real)	*(define-fun e () Real 4.0)*
(assert (> e (+ (to_real (+ a b)) 2.0)))	*(define-fun c () Int 0)*
(assert (= d (+ (to_real c) 0.5)))	*(define-fun d () Real (/ 1.0*
(check-sat)	*2.0))*
(get-model)	*)*

Takođe, postoji uslovni operator (if-then-else operator). Na primer, izraz (ite (and (= x!1 11) (= x!2 false)) 21 0) ima vrednost 21 kada je promenljiva x!1 jednaka 11, a promenljiva x!2 ima vrednost False. U suprotnom, vraća se 0.

U slučaju deljenja, može se koristiti ite (if-then-else) operator i na taj način se može dodeliti interpretacija u slučaju deljenja nulom.

Mogu se konstruisati novi operatori, korišćenjem `define-fun` konstruktora. Ovo je zapravo makro, pa će rešavač vršiti odgovarajuće zamene.

Primer 4 *Kod definiše operator deljenja tako da rezultat bude specifikovan i kada je delilac 0. Uvode se dve konstante realnog tipa i primenjuje se definisani operator. Z3 rešavač pronalazi nezadovoljivost tvrđenja s obzirom da operator mydiv vraća 0 pa relacija poređenja ne može biti tačna.*

Formula prosleđena rešavaču:	***Izlaz:***
(define-fun mydiv ((x Real) (y Real)) Real	*unsat*
* (if (not (= y 0.0)) (/ x y) 0.0))*	
(declare-const a Real)	
(declare-const b Real)	
(assert (>= (mydiv a b) 1.0))	
(assert (= b 0.0))	
(check-sat)	

Primer 5 *Primer ilustruje rešavanje nelinearnih problema sa celobrojnim i realnim konstantama. Kada su prisutna samo nelinearna ograničenja nad realnim konstantama, Z3 koristi posebne metode odlučivanja.*

Formula prosleđena rešavaču:	***Izlaz:***
(declare-const a Int)	*sat*
(assert (> (a a) 3))*	*(model*
(check-sat)	* (define-fun a () Int (- 8))*
(get-model)	*)*
(declare-const b Real)	
(declare-const c Real)	*unsat*
(assert (= (+ (b b b) (* b c)) 3.0))*	
(check-sat)	*sat*
(declare-const b Real)	*(model*
(declare-const c Real)	* (define-fun b () Real (/ 1.0 8.0))*
(assert (= (+ (b b b) (* b c)) 3.0))*	* (define-fun c () Real (/ 15.0 64.0))*
(check-sat)	*)*
(get-model)	

Primer 6 *Navodimo različite načine predstavljanja bitvektora. Ukoliko zapis počinje sa #b, bitvektor se zapisuje u binarnom formatu. Ukoliko zapis počinje sa #x, bitvektor se zapisuje u heksadecimalnom formatu. U oba slučaja, nakon specifikacije formata, zapisuje se dužina vektora. Drugi način zapisa počinje skraćenicom bv, navođenjem vrednosti i na kraju dužine. Komandom* `(display t)` *štampa se izraz* `t`*.*

Formula prosleđena rešavaču:	***Izlaz:***
(display #b0100)	*#x4*
(display (_ bv20 8))	*#x14*
(display (_ bv20 7))	*#b0010100*
(display #x0a)	*#x0a*
(set-option :pp.bv-literals false)	*(_ bv4 4)*
(display #b0100)	*(_ bv20 8)*
(display (_ bv20 8))	*(_ bv20 7)*
(display (_ bv20 7))	*(_ bv10 8)*
(display #x0a)	

Primer 7 *Ovaj primer ilustruje aritmetičke operacije nad bitvektorima. Podržane aritmetičke operacije su sabiranje (bvadd), oduzimanje (bvsub), unarna negacija (bvneg), množenje (bvmul), računanje modula (bvmod), šiftovanje ulevo (bvshl), neoznačeno (logičko) šiftovanje udesno (bvlshr) i označeno (aritmetičko) šiftovanje udesno (bvashr). Od logičkih operacija postoji podrška za disjunkciju (bvor), konjunkciju (bvand), ekskluzivnu disjunkciju (bvxor), negaciju disjunkcije (bvnor), negaciju konjunkcije (bvnand) i negaciju ekskluzivne disjunkcije (bvnxor). Komandom* `(simplify t)` *prikazuje se jednostavniji izraz ekvivalentan izrazu* `t` *ukoliko postoji.*

Formula prosleđena rešavaču:	***Izlaz:***
(simplify (bvadd #x07 #x03))	*#x0a*
(simplify (bvsub #x07 #x03))	*#x04*
(simplify (bvneg #x07))	*#xf9*
(simplify (bvmul #x07 #x03))	*#x15*
(simplify (bvsmod #x07 #x03))	*#x01*
(simplify (bvshl #x07 #x03))	*#x38*
(simplify (bvlshr #xf0 #x03))	*#x1e*
(simplify (bvashr #xf0 #x03))	*#xfe*
(simplify (bvor #x6 #x3))	*#x7*
(simplify (bvand #x6 #x3))	*#x2*
(simplify (bvxor #x6))	*#x9*
(simplify (bvnand #x6 #x3))	*#xd*
(simplify (bvnor #x6 #x3))	*#x8*
(simplify (bvxnor #x6 #x3))	*#xa*

Primer 8 *Postoji brz način da se proveri da li su brojevi fiksne dužine stepeni*

dvojke. Ispostavlja se da je bitvektor x stepen dvojke ako i samo ako je vrednost izraza x ∧ (x - 1) jednaka 0. Dodaje se negacija ove jednakosti kao tvrđenja i vrši se proveravanje za bitvektore vrednosti 0, 1, 2, 4 i 8. U svim slučajevima brojevi su stepeni dvojke pa Z3 rešavač vraća nezadovoljivost.

Formula prosleđena rešavaču:

```
(define-fun is-power-of-two
   ((x (_ BitVec 4))) Bool
   (= #x0 (bvand x (bvsub x #x1)))
)
(declare-const a (_ BitVec 4))
(assert
   (not (= (is-power-of-two a)
      (or (= a #x0)
         (= a #x1)
         (= a #x2)
         (= a #x4)
         (= a #x8)
      ))
   )
)
(check-sat)
```

Izlaz:

```
unsat
```

Primer 9 *Primer ilustruje upotrebu relacija nad bitvektorima. Podržane relacije uključuju neoznačene i označene verzije za operatore <, <=, > i >=. Neoznačene varijante počinju nazivom bvu, a u nastavku sledi ime relacije. Na primer, relacija <= nad neoznačenim brojevima zadaje se komandom bvule, a relacija > nad neoznačenim brojevima komandom bvugt. Označene varijante počinju nazivom bvs, a u nastavku ponovo sledi ime relacije. Na primer, relacija >= nad neoznačenim brojevima zadaje se komandom bvsge, a relacija < nad označenim brojevima komandom bvslt.*

Formula prosleđena rešavaču:	***Izlaz:***
(simplify (bvule #x0a #xf0))	*true*
(simplify (bvult #x0a #xf0))	*true*
(simplify (bvuge #x0a #xf0))	*false*
(simplify (bvugt #x0a #xf0))	*false*
(simplify (bvsle #x0a #xf0))	*false*
(simplify (bvslt #x0a #xf0))	*false*
(simplify (bvsge #x0a #xf0))	*true*
(simplify (bvsgt #x0a #xf0))	*true*

Rešavač Z3 nudi funkcije za promenu načina reprezentacije brojeva. Moguće su konverzije reprezentacije brojeva linearne aritmetike u reprezentaciju bitvektora i obrnuto. Ovaj rezultat može se postići naredbama:

```
(define b (int2bv[32] z))
(define c (bv2int[Int] x))
```

Primer 10 *Primer poredi bitvektore koristeći označene i neoznačene verzije operatora. Označeno poređenje, kao što je bvsle, uzima u obzir znak bitvektora za poređenje, dok neoznačeno poređenje tretira bitvektor kao prirodan broj. Z3 rešavač pronalazi da je tvrđenje zadovoljivo i daje prikazani model.*

Formula prosleđena rešavaču:	***Izlaz:***
(declare-const a (_ BitVec 4))	*sat*
(declare-const b (_ BitVec 4))	*(model*
(assert (not (= (bvule a b) (bvsle a b)))	*(define-fun b () (_ BitVec 4) #xe)*
(check-sat)	*(define-fun a () (_ BitVec 4) #x0)*
(get-model)	*)*

Primer 11 *Definišemo tri konstante x, y i z celobrojnog tipa. Neka je a1 niz celobrojnih vrednosti. Tada je ograničenje (and (= (select a1 x) x) (= (store a1 x y) a1)) zadovoljivo kada je element niza a na poziciji x jednak definisanoj konstanti x i u slučaju kada su konstante x i y jednake. Rešavač Z3 vraća zadovoljivost zadatog tvrđenja i odgovarajući model.*

Formula prosleđena rešavaču:

```
(declare-const x Int)
(declare-const y Int)
(declare-const z Int)
(declare-const a1 (Array Int Int))
(assert (= (select a1 x) x))
(assert (= (store a1 x y) a1))
(check-sat)
(get-model)
```

Izlaz:

```
sat
(model
   (define-fun y () Int 1)
   (define-fun a1 () (Array Int Int)
      (_ as-array k!0)
   )
   (define-fun x () Int 1)
   (define-fun k!0
      ((x!1 Int)) Int (ite (= x!1 1) 1 0)
   )
)
```

Primer 12 *Definišemo konstantni niz m celobrojnog tipa i dve celobrojne konstante a i i. Uvodimo ograničenje da niz m sadrži samo jedinice. Z3 pronalazi da je ovo tvrdjenje zadovoljivo i daje prikazani model.*

Formula prosleđena rešavaču:

```
(declare-const m (Array Int Int))
(declare-const a Int)
(declare-const i Int)
(assert (= m ((as const (Array Int Int)) 1)))
(assert (= a (select m i)))
(check-sat)
(get-model)
```

Izlaz:

```
sat
(model
   (define-fun m () (Array Int Int)
      (_ as-array k!0)
   )
   (define-fun i () Int 0)
   (define-fun a () Int 1)
   (define-fun k!0 ((x!0 Int))
      Int (ite (= x!0 0) 1 1)
   )
)
```

Primer 13 *Definišemo dva konstantna niza a i b tipa Boolean i dokazujemo da važi svojstvo* $\neg(a \wedge b) \Leftrightarrow (\neg a \vee \neg b)$ *primenom funkcije map na sve elemente nizova. Kao ograničenje dodajemo negaciju prethodno navedene formule. Rešavač Z3 vraća nezadovoljivost negirane formule, odakle zaključujemo da je polazna formula validna.*

Formula prosleđena rešavaču:

```
(define-sort Set (T) (Array T Bool))
(declare-const a (Set Int))
(declare-const b (Set Int))
(assert (not (= ((_ map and) a b) ((_ map not)
   ((_ map or) ((_ map not) b) ((_ map not) a)))))
)
(check-sat)
```

Izlaz:

unsat

Tipovi podataka

Algebarski tipovi podataka omogućavaju specifikaciju uobičajnih struktura podataka. Slogovi, torke i skalari (enumeracijski tipovi) spadaju u algebarske tipove podataka. Primena algebarskih tipova podataka može se generalizovati. Mogu se koristiti za specifikovanje konačnih listi, stabala i rekurzivnih struktura.

Slogovi

Slog se specifikuje kao tip podataka sa jednim konstruktorom i proizvoljnim brojem elemenata sloga. Rešavač Z3 ne dozvoljava povećavanje broja argumenata sloga nakon njegovog definisanja. Važi svojstvo da su dva sloga jednaka samo ako su im svi argumenti jednaki.

Primer 14 *Pokazujemo svojstvo da su dva sloga jednaka ako i samo ako su im svi argumenti jednaki. Uvodimo parametarski tip Pair, sa konstruktorom mk-pair i dva argumenta kojima se može pristupiti koristeći selektorske funkcije first i second. Definišemo dva sloga p1 i p2, čija su oba podatka celobrojnog tipa. Dodajemo ograničenja da su slogovi p1 i p2 jednaki kao i ograničenje koje se odnosi na drugi element sloga. Rešavač Z3 u prvom slučaju vraća zadovoljivost formule i odgovarajući model. Dodavanjem ograničenja da prvi elementi slogova nisu jednaki korišćenjem selektorske funkcije first, tvrđenje postaje nezadovoljivo.*

Formula prosleđena rešavaču:

```
(declare-datatypes (T1 T2)
  (Pair (mk-pair (first T1) (second T2))))
(declare-const p1 (Pair Int Int))
(declare-const p2 (Pair Int Int))
(assert (= p1 p2))
(assert (> (second p1) 20))
(check-sat)
(get-model)
(assert (not (= (first p1) (first p2))))
(check-sat)
```

Izlaz:

```
sat
(model
   (define-fun p1 () (Pair Int Int)
      (mk-pair 0 21)
   )
   (define-fun p2 () (Pair Int Int)
      (mk-pair 0 21)
   )
unsat
```

Skalari (tipovi enumeracije)

Sorta skalara je sorta konačnog domena. Elementi konačnog domena se tretiraju kao različite konstante. Na primer, neka je S skalarni tip sa tri vrednosti A, B i C. Moguće je da tri konstante skalarnog tipa S budu različite. Ovo svojstvo ne može važiti u slučaju četiri konstante.

Primer 15 *Prilikom deklaracije skalarnog tipa podataka, navodi se broj različitih elemenata domena, u ovom primeru tri i pokazuje se nezadovoljivost tvrđenja sa četiri različita elementa domena.*

Formula prosleđena rešavaču:

```
(declare-datatypes () ((S A B C)))
(declare-const x S)
(declare-const y S)
(declare-const z S)
(declare-const u S)
(assert (distinct x y z))
(check-sat)
(get-model)
(assert (distinct x y z u))
(check-sat)
```

Izlaz:

```
sat
(model
   (define-fun z () S A)
   (define-fun y () S B)
   (define-fun x () S C)
)
unsat
```

Rekurzivni tipovi podataka

Deklaracija rekurzivnog tipa podataka uključuje sebe direktno kao komponentu. Standardni primer rekurzivnog tipa podataka je lista. Lista celobrojnih vrednosti sa imenom `list` može se deklarisati naredbom:

```
(declare-datatypes ((list (nil) (cons (hd Int) (tl list))))
```

Rešavaš Z3 ima ugrađenu podršku za liste korišćenjem ključne reči `List`. Prazna lista se definiše korišćenjem klučne reči `nil` a konstruktor `insert` se koristi za dodavanje elemenata u listu. Selektori `head` i `tail` se definišu na uobičajan način.

Primer 16 *Deklarišemo tri liste l1, l2 i l3 sa celobrojnim vrednostima, kao i celobrojnu konstantu x. Dodaju se ograničenja za prve elemente listi l1 i l2 korišćenjem selektora. Pored toga, dodaje se ograničenje da liste l1 i l2 nisu jednake, tj. da nisu svi elementi na odgovarajućim pozicijama u listama jednaki. U listu l2 dodaje se konstanta x. Rešavač Z3 vraća zadovoljivost tvrđenja i dobijeni model prikazujemo u nastavku.*

Formula prosleđena rešavaču:

```
(declare-const l1 (List Int))
(declare-const l2 (List Int))
(declare-const l3 (List Int))
(declare-const x Int)
(assert (not (= l1 nil)))
(assert (not (= l2 nil)))
(assert (= (head l1) (head l2)))
(assert (not (= l1 l2)))
(assert (= l3 (insert x l2)))
(assert (> x 100))
(check-sat)
(get-model)
```

Izlaz:

```
sat
(model
  (define-fun l3 () (List Int)
  (insert 101 (insert 0 (insert 1 nil)))
  (define-fun x () Int 101)
  (define-fun l1 () (List Int) (insert 0 nil))
  (define-fun l2 () (List Int) (insert 0
    (insert 1 nil))
  )
)
```

U prethodnom primeru, uvode se ograničenja da su liste l1 i l2 različite od nil. Vrši se uvođenje ovih ograničenja jer interpretacija selektora head i tail nije specifikovana u slučaju praznih lista.

3.4 Upotreba rešavača korišćenjem C++ interfejsa

C++ interfejs prema rešavaču Z3 obezbeđuje različite strukture podataka, klase i funkcije koje su potrebne za direktnu komunikaciju C++ aplikacije sa rešavačem. Neke od najbitnijih klasa biće opisane u nastavku, dok se kompletan opis interfejsa može naći na internetu [9].

Klasa `Z3_sort` koristi se za definisanje tipa izraza. Prilikom definisanja izraza navodi se tip kako bi bio poznat skup vrednosti koje mu se mogu dodeliti kao i skup dozvoljenih metoda. Sorte izraza definisane su tipom enumeracije. Neke od najvažnijih sorti su `Z3_BOOL_SORT`, `Z3_INT_SORT`, `Z3_REAL_SORT`, `Z3_BV_SORT` i `Z3_ARRAY_SORT`. Određivanje sorte izraza vrši se funkcijom `sort_kind()` sa povratnom vrednošću tipa enumeracije. Za proveru pripadnosti izraza sorti, koriste se funkcije `is_bool()`, `is_int()`, `is_real()`, `is_array()` i `is_bv()`. Sorte različitih izraza se mogu porediti korišćenjem operatora jednakosti.

Za upravljanje objektima interfejsa kao i za globalno konfigurisanje koristi se klasa `context`. Klasa sadrži konstruktor bez argumenata. Upotrebom klase `context`, mogu se detektovati različite vrste grešaka u korišćenju C++ API-ja. Greške su definisane tipom enumeracije `Z3_ERROR_CODE`. Neke od konstanti enumeracije su `Z3_OK`, `Z3_SORT_ERROR`, `Z3_INVALID_USAGE` i `Z3_INTERNAL_FATAL`. Kontekst omogućava kreiranje konstanti metodama `bool_const()`, `int_const()`, `real_const()` i `bv_const()`. Definisanje različitih sorti omogućeno je metodama `bool_sort()`, `int_sort()`, `real_sort()`, `bv_sort()` i `array_sort()`.

Izrazi koji se formiraju pripadaju klasi `expr`. Z3 izraz se koristi za predstavljanje formula i termova. Formula je proizvoljan izraz sorte `Boolean`. Sadrži konstruktor čiji je argument objekat klase `context`. Za dobijanje izraza na zadatoj poziciji u skupu izraza koristi se metoda `at(expr const &index)`. Provera da li podizraz predstavlja deo drugog izraza vrši se metodom `contains(expr const &s)`. Za dobijanje pojednostavljenog izraza ekvivalentnog polaznom koristi se metoda `simplify()` ukoliko takav izraz postoji. Za dobijanje pojednostavljenog izraza može se navesti i skup parametara koji se prosleđuju Z3 simplifikatoru. Zamenu vektora izraza drugim vektorom vrši se metodom `substitute(expr_vector const &source, expr_vector const &destination)`.

Postoji veliki broj metoda i operatora koji se koriste za izgradnju složenih izraza. Podržan je veliki broj aritmetičkih operatora za rad za izrazima uključujući operatore sabiranja, oduzimanja, množenja, deljenja, računanja stepena i mo-

dula. Svi aritmetički operatori kao argumente imaju izraze. Rezultat primene operatora je novi izraz. Pored toga, mogu se koristiti i različiti logički operatori. Neke od podržanih logičkih operacija su konjunkcija, disjunkcija, implikacija, negacija konjunkcije i negacija disjunkcije. Konjunkcija vektora izraza vrši se metodom `mk_and(expr_vector const &args)`. Disjunkcija vektora izraza vrši se metodom `mk_or(expr_vector const &args)`. Implikacija dva izraza vrši se metodom `implies(expr const &a, expr const &b)`. Negacija konjunkcije dva izraza vrši se metodom `nand(expr const &a, expr const &b)`. Negacija disjunkcije dva izraza vrši se metodom `nor(expr const &a, expr const &b)`. Nad izrazima se mogu primenjivati relacijski operatori `==`, `!=`, `<`, `<=`, `>`, `>=` pri čemu izrazi moraju biti odgovarajuće sorte kako bi poređenje bilo moguće. Nadovezivanje dva izraza vrši se metodom `concat(expr const &a, expr const &b)`. Može se vršiti i nadovezivanje vektora izraza. Kombinovanjem pomenutih metoda i operatora mogu se graditi izrazi proizvoljne složenosti.

Funkcije predstavljaju osnovne gradivne blokove. Definicija funkcije vrši se objektima klase `func_decl`. Korišćenjem ove klase definišu se interpretirane i neinterpretirane funkcije rešavača Z3. Povratne vrednosti funkcija određene su tipom enumeracije `Z3_decl_kind`. Neke od konstanti enumeracije su `Z3_OP_TRUE`, `Z3_OP_FALSE`, `Z3_OP_REAL`, `Z3_OP_INT` i `Z3_OP_ARRAY`. Dobijanje imena funkcijskog simbola vrši se metodom `name()`. Određivanje arnosti funkcijskog simbola vrši se metodom `arity()`. Određivanje sorte i-tog parametra funkcijskog simbola određuje se metodom `domain(unsigned i)`.

U okviru C++ interfejsa, teorije rešavača Z3 zadate su semantički navođenjem modela. Ova podrška implementirana je klasom `model`. Sadrži konstruktor čiji je argument objekat klase kontekst. Interpretacija izraza definisanog u modelu dobija se korišćenjem metode `eval(expr const &n)`. Metodom `get_func_decl(unsigned i)` dobija se i-ti funkcijski simbol modela. Metodom `get_const_decl(unsigned i)` dobija se interpretacija i-te konstante modela. Metodom `num_consts()` dobija se broj konstanti datog modela kao funkcijskih simbola arnosti 0. Metodom `num_funcs()` dobija se broj funkcijskih simbola arnosti veće od 0. Metodom `size()` vraća se broj funkcijskih simbola modela. Poređenje modela vrši se operatorom jednakosti. Dva modela su jednaka ukoliko su im jednake interpretacije svih funkcijskih simbola. Za ispisivanje modela, koristi se funkcija `Z3_model_to_string` čiji su argumenti objekti klasa `context` i `model`.

Sa Z3 rešavačem komunicira se korišćenjem objekta klase `solver`. Objekat klase

`solver` inicijalizuje se vrednostima objekta klase `context`. Osnovni metodi klase solver su `add`, `check` i `get_model`. Metodom `add(expr const &e)` dodaje se ograničenje koje se prosleđuje rešavaču. Metodom `check()` proverava se zadovoljivost ograničenja prosleđenih rešavaču. Metodom `get_model()` vraća se model definisan ograničenjima ukoliko postoji. Pre korišćenja metode `get_model()`, mora se pozvati metod `check()`. Metodom `assertions()` vraća se vektor ograničenja prosleđenih rešavaču. Ograničenja se mogu čitati iz fajla i iz stringa, korišćenjem metoda `from_file(char const *file)` i `from_string(char const *s)`. Uklanjanje svih ograničenja prosleđenih rešavaču vrši se metodom `reset()`.

Interfejs kroz primere

Naredni primeri ilustruju korišćenje najvažnijih klasa i metoda C++ interfejsa za komunikaciju sa Z3 rešavačem. Primer 17 ilustruje kreiranje bulovskih izraza i jednostavne formule i prikazuje kako se kreira i upotrebljava klasa `context` i klasa `solver`. U ovom primeru, ilustrovano je dodavanje ograničenja u solver metodom `add` i proveravanje njegove zadovoljivosti metodom `check`.

Primer 17 *Primer demonstrira važenje De Morganovog zakona dokazivanjem formule iz primera 1. Pokazuje se nezadovoljivost negirane formule. U zavisnosti od rezultata štampa se odgovarajuća poruka.*

```
void demorgan() {
    context c;
    expr x = c.bool_const("x");
    expr y = c.bool_const("y");
    expr e = (!(x && y)) == (!x || !y);

    solver s(c);
    s.add(!e);

    switch (s.check()) {
    case unsat:   std::cout << "Formula je validna"; break;
    case sat:     std::cout << "Formula nije validna"; break;
    case unknown: std::cout << "Rezultat je nepoznat"; break;
    }
}
```

Primer 18 ilustruje kreiranje celobrojnih konstanti i jednostavne neinterpretirane funkcije upotrebom klase `func_decl`. Prilikom definisanja funkcije navodi se njeno

ime kao i sorte argumenta i povratne vrednosti. Ilustruje se kreiranje složenijeg izraza upotrebom metode `implies` koji odgovara logičkom operatoru implikacije. Složeniji izraz prosleđuje se solveru i proverava se njegova zadovoljivost.

Primer 18 *Primer demonstrira upotrebu neinterpretiranih funkcija dokazivanjem formule x = y => g(x) = g(y). Dodaje se negacija prethodno navedene formule. U zavisnosti od rezultata, štampa se odgovarajuća poruka.*

```
void primer_sa_neinterpretiranim_funkcijama() {
    context c;
    expr x       = c.int_const("x");
    expr y       = c.int_const("y");
    sort I       = c.int_sort();
    func_decl g = function("g", I, I);

    solver s(c);
    expr e = implies(x == y, g(x) == g(y));
    s.add(!e);
    if (s.check() == unsat)
        std::cout << "dokazano";
    else
        std::cout << "nije dokazano";
}
```

U primeru 19 korišćenjem metode `get_model` pristupa se modelu koji je solver vratio. Vrši se evaluacija izraza dobijenih iz modela primenom metode `eval`.

Primer 19 *Rešavaču se prosleđuju jednostavna ograničenja nad konstantama. Zatim se vrši evaluacija jednostavnih izraza nad konstantama definisanih u modelu.*

```
void eval_primer() {
    context c;
    expr x = c.int_const("x");
    expr y = c.int_const("y");
    solver s(c);

    s.add(x < y);
    s.add(x > 2);
    std::cout << s.check();

    model m = s.get_model();
    std::cout << "Model:" << m;
```

```
13      std::cout << "x+y = " << m.eval(x+y);
14  }
```

Primer 20 ilustruje pronalaženje interpretacija konstanti modela za problem linearne aritmetike uvođenjem ograničenja. Korišćenjem klase `func_decl`, za svaku od konstanti kao funkcijskih simbola arnosti 0 ispisuje se njeno ime i dodeljena vrednost.

Primer 20 *Primer ispisuje imena i interpretacije konstanti modela korišćenjem funkcije* `arity` *klase* `func_decl`.

```
1  void primer_linearne_aritmetike() {
2      context c;
3      expr x = c.int_const("x");
4      expr y = c.int_const("y");
5      solver s(c);
6      s.add(x >= 1);
7      s.add(y < x + 3);
8      model m = s.get_model();
9
10     for(unsigned i = 0; i < m.size(); i++) {
11         func_decl v = m[i];
12         assert(v.arity() == 0);
13         std::cout << v.name() << "=" << m.get_const_interp(v);
14     }
15 }
```

Primer 21 ilustruje pronalaženje interpretacija realnih konstanti modela za problem nelinearne aritmetike uvođenjem ograničenja. Interpretacija realnih konstanti ispisuje se u celobrojnom i realnom formatu korišćenjem opcija za konfigurisanje formata ispisa klase `context`.

Primer 21 *Primer ispisuje imena i interpretacije realnih konstanti modela korišćenjem funkcije* `arity` *klase* `func_decl`.

```
1  void primer_nelinearne_aritmetike() {
2      context c;
3      expr x = c.real_const("x");
4      expr y = c.real_const("y");
```

```
5       expr z = c.real_const("z");
6
7       solver s(c);
8       s.add(x*x + y*y == 1);
9       s.add(x*x*x + z*z*z < c.real_val("1/2"));
10
11      std::cout << s.check();
12      model m = s.get_model();
13      std::cout << m;
14
15      for(unsigned i = 0; i < m.size(); i++) {
16          func_decl v = m[i];
17          assert(v.arity() == 0);
18          std::cout << v.name() << "=" << m.get_const_interp(v);
19      }
20
21  }
```

Primer 22 ilustruje pronalaženje interpretacija konstanti koje imaju bitvektorsku reprezentaciju korišćenjem metode `bv_const` klase `context`. Parametri ove metode su ime i broj mesta za zapisivanje konstante. Za svaku od konstanti ispisuje se njeno ime i dodeljena vrednost.

Primer 22 *Primer ispisuje imena i interpretacije konstantni predstavljenih bitvektorom dužine 32.*

```
1   void primer_sa_bitvektorima() {
2       context c;
3       expr x = c.bv_const("x", 32);
4       expr y = c.bv_const("y", 32);
5
6       solver s(c);
7       s.add((x ^ y) - 103 == x * y);
8       std::cout << s.check();
9       std::cout << s.get_model();
10
11      for(unsigned i = 0; i < m.size(); i++) {
12          func_decl v = m[i];
13          assert(v.arity() == 0);
14          std::cout << v.name() << "=" << m.get_const_interp(v);
15      }
16  }
```

Glava 4

Zaključak

Literatura

[1] C. Barrett i R. Sebastiani. *Satisfiability Modulo Theories, Frontiers in Artificial Intelligence and Applications*. 1987, str. 825–885.

[2] Armin Biere i Marijin Heule. *Handbook of Satisfiability, volume 185 of Frontiers in Artificial Intelligence and Applications*. 2009.

[3] J. Levitt C. Barrett D. Dill. *A decision procedure for bit-vector arithmetic*. 1998, str. 522–527.

[4] Aaron Stump Clark Barrett. *The SMT-LIB Standard - version 2.0*. 2013.

[5] B. Dutertre i L. de Moura. *A Fast Linear-Arithmetic Solver for DPLL(T)*. 2006.

[6] R. W. House i T. Rado. *A Generalization of Nelson-Open's Algorithm for Obtaining Prime Implicants*.

[7] Leonardo de Moura i Nikolaj Bjorner. *Z3 - An Efficient SMT Solver, Microsoft Research*. 2008, str. 337–340.

[8] Microsoft Research. *Automatically generated documentation for the Z3 APIs*. `http://z3prover.github.io/api/html/index.html`. 2016.

[9] Microsoft Research. *Automatically generated documentation for the Z3 C++ API*. `https://z3prover.github.io/api/html/group__cppapi.html`. 2016.

Biografija autora