

Objedinjavanje Izvršavanja Imperativnog i Deklarativnog Koda

Seminarski rad u okviru kursa Verifikacija softvera
Matematički fakultet

Miloš Lončarević
milosloncarevic494@gmail.com

21. maj 2018.

Sažetak

U ovom radu je predstavljeno okruženje za izvršavanje deklarativnih specifikacija u kontekstu imperativnog objektno-orijentisanog programskog jezika. Specifikacije su napisane u relacionoj logici prvog reda sa tranzitivnim zatvorenjem, imperativni jezik je Java.

1 Uvod

U ovom radu je predstavljen framework pod nazivom Squander koji obezbeđuje okruženje za pisanje deklarativnih ograničenja i imperativnih naredbi u kontekstu jednog programa. Ovo je naročito korisno za implementaciju programa koji uključuju izračunavanja koja se relativno lako opisuju, ali veoma teško algoritamski rešavaju. Predstavljena tehnologija omogućava izvršavanje deklarativnih specifikacija bez potrebe za pisanjem bar jedne linije imperativnog koda. Podržan jezik za specifikacije je *JFSL*, koji podržava relacionu logiku prvog reda sa tranzitivnim zatvorenjem i standardne Java izraze. Zbog mogućnosti mešanja imperativnog koda i deklarativnih specifikacija, korisnik lako može da predstavi ograničenja terminima postojećih struktura podataka i objekata na hipu.

2 Primer - Sudoku rešavač

U ovom primeru se razmatra jednostavan Sudoku rešavač. Rešavaču je dat delimično popunjen Sudoku, od koga se očekuje da popuni prazna mesta u skladu sa već poznatim ograničenjima. Java model za ovaj problem je dat na slici. *CellGroup* sadrži niz sastavljen od *Cell* objekata, bez duplikata. *CellGroup* je definisan u klasi *Sudoku* za svaki red, kolonu i kockicu. Metod *init()* ima zadatak da kreira $n \times n$ *Cell* objekata, $n + n + n$ *CellGroup* objekata, i omogućava pravilno deljenje *Cell* objekata između *CellGroup* objekata. Ograničenje za *CellGroup* se može izraziti kroz jednu *JFSL* liniju (*@Invariant*). Kaže da za sve celobrojne vrednosti v različite od 0, slektuj sve *Cell* objekte iz *CellGroup.cells* sa vrednosti v i osiguraj da je njihov broj 0 ili 1 (lone). Ograničenja klase potvrđuju svojstva svih članova klase, ali se ne mogu izvršiti. Da bi uspostavili ograničenje koje može da se izvrši, definišemo standardan Java metod sa specifikacijom, koja sadrži:

```

static class Cell { int val = 0; } // 0 means empty

@Invariant("all v: int |
v != 0 => lone this.cells.elems.val.v")
static class CellGroup {
Cell[] cells;
public CellGroup(int n) { this.cells=new Cell[n];}

public class Sudoku {
CellGroup[] rows, cols, grids; int n;
public Sudoku(int n) { init(n); }
@Ensures("all c:Cell | c.val>0 && c.val<=this.n")
@Modifies("Cell.val [{c:Cell | c.val == 0}]")
public void solve() { Squander.exe(this); }}

```

- preduslov (*@Requires*) pre poziva metode, pretpostavljen ako nije naveden
- postuslov (*@Ensures*) posle izvršavanja metode
- frame uslov (*@Modifies*) koji pokazuje koji delovi su dozvoljeni da se menjaju

Specifikacija za metod *solve()* kaže da nakon izvršavanja sve ćelije moraju biti popunjene ne nula vrednostima. Frame uslov ograničava izmene na ona *Cell.value* polja koja su trenutno prazna. Ova ograničenja u kombinaciji sa prvim ograničenjem su dovoljna za rešavanje Sudoku-a. Telo metoda sadrži samo poziv uslužnog metoda *Squander.exe*, koji rešava problem u skladu sa specifikacijom. Ako je rešenje pronađeno, program može, na primer, da ispiše dobijen Sudoku, u suprotnom se izbacuje izuzetak koji kaže da rešenje ne može biti pronađeno.

3 Pozadina

3.1 Kodkod - rešavač za relacionu logiku

Kodkod je rešavač ograničenja u relacionoj logici. Uzima relacionu formulu koja se rešava, zajedno sa definicijama skupa netipiziranih relacija, koji se sastoje od granica za svaku relaciju i ograničen domen atoma od kojih su sastavljene relacije u rešenju. Kodkod prevodi formulu u SAT problem i primenjuje SAT rešavač u potrazi za zadovoljavajućim rešenjem, koje ako je pronađeno, prevodi nazad u relacioni domen. Relacije u Kodkod-u su netipizirane, što znači da svaka relacija može potencijalno da sadrži bilo koju konačnu torku. Stvarni skup torki koji relacija može da sadrži je definisan preko Kodkod granica. Dve granice moraju da budu definisane: donja granica koja definiše torke koje relacija mora da sadrži, i gornja granica koja definiše torke koje relacija može da sadrži. Veličina ovih granica utiče na vreme pretraživanja - što je manja razlika između gornje i donje granica, to je manji prostor za pretraživanje, pa je rešavanje brže.

3.2 JFSL - JForge Specification Language

JFSL je formalni jezik za pisanje specifikacija u Javi. JFSL pruža najčešće operatore relacione algebre: pridruživanje (*.*), transponovanje (*~*), konstruktor torki (*->*), tranzitivno zatvorenje (*^*) i refleksivno tranzitivno zatvorenje (***), kao i skupovne algebarske operacije: unija (*+*), razlika (*-*) i presek (*&*). Kao i u Javi domen se sastoji od Java objekata, null vrednosti i primitivnih vrednosti. Svaki tip odgovara skupu objekata tačno tog tipa (na primer *int* je skup svih primitivnih celih brojeva). Specifikacije su napisane kao Java klauzule. Pored već pomenutih *@Invariant*,

@Requires, *@Ensures* i *@Modifies* klauzule, *JFSL* pruža podršku specijalnom specifikaciji polju pomoću *@SpecField* napomene (definisanje tipa i opciono apstraktna funkcija koja određuje kako se računa vrednost polja u odnosu na druga polja).

4 Od objekata do relacija

Izvršavanje Squandera počinje kad se u klijentskom programu pozove funkcija *Squander.exe()* i sastoji se iz sledećih koraka:

- Sastavljanje relevantnih ograničenja, iz napomena klauzula koje sadrže specifikaciju metode, kao i iz klauzula klasa koje odgovaraju invarijantama svih relevantnih klasa
- Konstrukcija relacija koje predstavljaju vrednosti objekata i njihovih polja u pre-stanju, i konstrukcija dodatnih relacija za polja koja mogu da se menjaju kako bi se sačuvala njihove vrednosti u post-stanju, zajedno sa svojim Kodkod granicama
- Pravljenje jedinstvene relacije (koja se prosleđuje Kodkod-u)
- Ako je rešenje pronađeno, prevođenje Kodkod rezultata i ažuriranja Java hipa, modifikacijom polja objekata

4.1 Obilazak hipa i numerisanje objekata

Squander pronalazi dostupan deo hipa, koristeći standardan BFS algoritam. Interesantan deo je kako serijalizovati objekte. Squander pruža mehanizam koji obezbeđuje različite vrste serijalizacije u zavisnosti od klase objekta. Na primer default serijalizacija vraća samo vrednosti polja objekta. Squander je run-time mehanizam koji koristi JVM, tako da nema pristup compile-time informacijama, kao što su tipovi parametara koji su veoma bitni. Java refleksija pruža ove informacije koje su dostupne za sve objekte.

4.2 Čitanje, parsiranje i provera tipova - *JFSL*

Kada se pronađe nova klasa prilikom obilaska hipa, njena specifikacija se dobija refleksijom. Specifikacija klase se sastoji iz klasne invarijante (*@Invariant*) i polja specifikacije (*@SpecField*). Ove specifikacije se mogu definisati direktno u izvornom fajlu preko Java klauzula ili preko specijalnog spec fajla. Nakon toga se specifikacije parsiraju i vrši se provera tipova, i na kraju se prevode u relacione izraze.

4.3 Definisanje relacija i odnosa

Translacija ne obuhvata sva polja nego samo ona značajna, tj. samo ona koja su pomenuta u specifikacijama za konkretan metod. Takođe ni svi objekti nisu potrebni, nego samo oni čija su polja uključena u translaciji. Takvi objekti se nazivaju literali. Prvo se definise konačan domen literala u okviru granice. Za svaki od njih se kreira unarna konstantna relacija (gornja i donja granica se poklapaju). Za svaki Java tip se kreira relacija (sa odgovarajućim granicama tako da sadrži poznate literale). Za svako polje (uključujući i polja specifikacije) se kreira relacija $fld.declType \rightarrow fld.type$ (*dclType* je tip klase kojoj pripada). Ako polje može da se menja (pomenuto u *@Modifies* napomeni) kreira se dodatna

relacija sa sufiksom „pre“ koji označava pre-state vrednost. Relacijama za polja koja se menjaju su date iste granice koje odgovaraju trenutnom stanju hipa. Za modifikovane relacije, „pre“ relacija ima iste granice, a „post“ relacija je ograničena tako da može sadržati bilo koju torku koja je dozvoljena tipom polja.

4.4 Vraćanje stanja Java hipa

Nakon izvršavanja Kodkod-a, stanje hipa mora biti ažurirano tako da sadrži rezultat pretrage. Ako rešenje nije pronađeno, Squnader samo izbacuje izuzetak. Ako postoji više rešenja, jedno će biti vraćeno. Rešenje se vraća na hip putem modifikacije polja objekata refleksijom. Na kraju izvršavanja se vraća u klijentski kod. Proces vraćanja je jednostavan. Tokom prevođenja u Kodkod, Squnader čuva mapiranje Java objekata u Kodkod atome i mapiranje Java polja u Kodkod relacije. Tako da se Kodkod rešenje može direktno mapirati unazad u vrednosti Java polja.

5 Minimizacija veličine domena

Da bi se predstavila relacija r arnost k , Kodkod alocira matricu veličine n^k , gde je n broj elementata u domenu. Zbog preformansi, Kodkod koristi niz indeksiran Java celobrojnim vrednostima, što znači da je veličina matrice ograničena najvećom celobrojnomo vrednosti u Javi. Ako domen sadrži 1291 vredosti ili više, matrica za ternarne relacije sadrži 1291^3 ćelija, što prelazi limit. To može da predstavlja problem zato što se veoma često koriste ternarne relacije i hipovi sa više od 1290 vrednosti, pa onda neće bit moguće koristiti translaciju iz sekcije 4.4.

5.1 KodkodPart translacija

Umesto translacije koja se koristi u sekciji 4.4, dozvoljava se preslikavanje više literala u jedan Kodkod element (atom), tako da može biti više literala nego atoma. Ali ključan uslov je da postoji i inverzna funkcija, zbog ažuriranja hipa nakon pronalaska rešenja. Podsetimo da su tipovi polja predstavljeni kao unije baznih tipova (zovemo ih particije). Osnovna ideja ove translacije je: svi literali unutar blio kojoj particije moraju bit mapirani u različite atome, dok literali koji ne pripadaju zajedničkoj particiji mogu da dele atome.

5.2 Algoritam particionisanja

Zadatak ovog algoritma je da za skup baznih domena, literala, particija i za datu funkciju koja preslikava domene u skup literala koji mu pripadaju pronađe skup atoma i funkciju koja za svaku particiju vraća različite atome za svaki litaral koji pripada toj particiji. Osnovni cilj ovog algoritma je da minimizuje broj atoma kako taj broj ne bi bio isti kao i broj literala. Algoritam radi na sledeći način:

1. Zavisnotsti izmedju domena su izračunate
2. Najveća particija je pronađena
3. Za svaki literal l iz te particije je kreiran atom a
4. Za svaku drugu particiju p , za sve literala l koji pripadaju p , a koji već nemaju pridružen atom, skup mogućih atoma je izračunat i prva vrednost iz tog skupa je pridružena literalu l

6 Korisnički definisane apstakcije za bibliotečke tipove

Squnader obezbeđuje generičko rešenje dozvoljavajući korisnicima da pišu funkcije apstrakcije i konkretizacije za bibliotečke klase. Potrebno je napisati *.jfspec* (u *JFSL*) fajl sa definicijom apstraktnih polja i serijalizator objekta kao implementaciju *IObjSer* interfejsa koji obezbeđuje funkcije apstrakcije i konkretizacije za apstraktna polja. Date su specifikacije za listu, skup i mapu.

```
interface List<E> {
  @SpecField("elts: int -> E")
  @SpecField("size: one int | this.size = #this.elts")
  @SpecField("prev: E -> E |
    this.prev = (this.elts).DEC . (this.elts)")
  @Invariant("all i: int | (i >= 0 && i < this.size)
    ? one this.elts[i] : no this.elts[i]")
}

interface Set<K> {
  @SpecField("elts: set K")
  @SpecField("size: one int | this.size = #this.elts")
}

interface Map<K,V> {
  @SpecField("elts: K -> V")
  @SpecField("size: one int | this.size = #this.elts")
  @SpecField("keys: set K | this.keys = this.elts.V")
  @SpecField("vals: set V | this.vals = this.elts[K]")
  @Invariant({
    "all k: K | k in this.elts.V => one this.elts[k]"
  })
}
```

7 Rešavanje teških problema

Ako je problem rešiv u polinomijalnom vremenu, dovoljno dobra manualna implementacija će najverovatnije da radi brže od Squnader implementacije. Ali ako je problem težak, Squnader rešenje može ispasti (u zavisnosti od efikasnosti SAT rešavača) bolje nego rešenje koje se dobija manuelnim algoritmom. Squnader neće uvek dati najefikasnije rešenje, ali interesantno je to da je Squnader rešenje konkurentno ručno pisanom rešenju (čak i uz potrebna kodiranja i dekodiranja).

Problem *N* kraljica uključuje postavljanje *N* kraljica na šahovsku tablu veličine $N \times N$ tako da se kraljice ne ugrožavaju međusobno. Metod *nqueens* uzima ceo broj *n* i skup koji već sadrži *n* *Cell* objekata. Od ovog metoda se očekuje da modifikuje dati skup tako da sadrži validne pozicije *N* kraljica. Frame uslov govori da se mogu menjati samo koordinate ćelije. U post uslovu, poslednja *all* klauzula postavlja ograničenje da dve kraljice ne mogu biti u istom redu, koloni ili dijagonali. Prve dve *all* klauzule kažu da svaki red i kolona mora da sadrži tačno jedan *Cell* objekat. Ova dva uslova su redundantna ali poboljšavaju brzinu rešavanja. U datoj tabeli prikazani su rezilati za različite vrednosti broja *N*.

```
@Requires("result.length == n")
@Ensures({
  "all k: int | k >= 0 && k < n => lone (Cell@i) . k",
  "all k: int | k >= 0 && k < n => lone (Cell@j) . k",
  "all q1: result.elts | no q2: result.elts - q1",
  "q1.i = q2.i || q1.i - q1.j = q2.i - q2.j || "+
  "q1.j = q2.j || q1.i + q1.j = q2.i + q2.j"
})
@Modifies({
  "Cell.i ||| {k: int | k >= 0 && k < n}",
  "Cell.j ||| {k: int | k >= 0 && k < n}"
})
public static void nqueens(int n, Set<Cell> result)
```

	n =	16	28	32	34	36	68
Mamual	0.01	0.49	15.94	428.94	t/o	t/o	
Squander	0.64	4.88	10.32	11.58	16.02	269.09	
tTransl	0.18	0.57	0.93	1.1	1.34	17.57	
tKodkod	0.38	1.45	2.59	3.08	3.54	32.71	
tSAT	0.08	2.86	6.8	7.4	11.14	218.81	

Literatura

- [1] Unifying Execution of Imperative and Declarative Code. Aleksandar Milicevic, Derek Rayside, Kuat Yessenov, Daniel Jackson. International Conference on Software Engineering (ICSE), 2011. <http://people.csail.mit.edu/kuat/papers/squander.pdf>