Dafny verification tool

Projekat na kursu iz Verifikacije sofvera

Matematički fakultet,Univerzitet u Beogradu

Nemanja Antić 1100/2017

Filip Lazić 1101/2017

Profesor: Milena Vujošević Janičić

Asistent: Ana Vulović

Sadržaj

Uvod

Dafni je programski jezik koji je da podržava statičku verifikaciju programa. To je imperativni, sekvencionalni programski jezik, koji podržava generičke klase i dinamičku alokaciju. Takođe sadrži preduslove i postuslove, invarijante petlje, klase… Neki od tipova koji su podržani u Dafniju su istinitosne vrednosti (bool), int vrednosti, skupovi (set). Kao što smo napisali Dafni sadrži klase, one mogu biti instancirane što daje neku vrstu objektno orjentisane prirode ali Dafni ne podržava podklase samim tim ni nasledivanje. Dafni kompajler proizvodi C# kod koji je sa svoje strane kompajliran do MSIL bajt koda za .NET platformu standardnim Microsoft C# kompajlerom. Dafni verifikator je deo kompajlera. Verifikator radi tako što prevodi program napisan u Dafniju u Boogie2 verifikacioni jezik na takav način da korektnost programa napisanog u Boogie jeziku implicira korektnost programa u Dafniju. Posle se alati Boogie jezika koriste da generišu uslove logike prvog reda koja se prenosi do Z3 SMT rešavaca. Najcešca upotreba Dafnija je u specifikaciji i verifikaciji nekih zahtevnijih algoritama.

Jezik „Dafny“

U ovom poglavlju ćemo objasniti neke osnove jezika. Dafni je jeyik koji je dizajniran da olakša pisanje ispravnog koda (Ispravnost u smislu da nema „runtime“ grešaka kao što su „index out of bounds“,“null dereferences“, deljenje nulom, ali i kada se radi tačno ono što je programer nameravao). Da bi se ovo postiglo Dafni se oslanja na anotacije na visokom nivou da bi se razjasnila i dokazala ispravnost koda. Efekat dela koda može biti dat apstraktno, koristeći „high-level“ izražavanje željenog ponašanja što čini pisanje koda lakšim i manje podložnim greškama. Dafni tada generiše dokaz da kod odgovara oznakama, ako su one tačne. Dafni tako podiže teret pisanja koda bez greške u pisanje napomena bez grešaka. Anotacije su često kraće i direktnije od koda, pa je mogućnost greške mnogo manja.

Metode

Jedna od osnovnih jedinica Danfni programa je metoda. Metoda je parče imperativnog, izvršivog koda. U ostalim programskim jezicima bi se moglo reći da je ovo funkcija, medjutim u Dafniju funkcija označava nešto drugo, ali o tome ćemo kasnije pričati.

```
method simpleMethod(x: int) returns (y: int)
{
   ...
}
```

Gore napisan kod deklariše metodu pod nazivom simpleMethod koja ima jedan ulazni parametar x tipa int, i vraća y takođe tipa int. Unutar zagrad ide telo metode koje sadrži naredbe koje Dafni podržava (dodele, uslovi, petlje, pozivi ostalih metoda, return naredbe…)

```
method MultipleReturns(x: int, y: int) returns (plus: int, minus: int)
{
   plus := x + y;
   minus := x - y;
   // comments: are not strictly necessary.
}
```

Ovaj kod daje primer vraćanja više promenljivih. Konkretno ova metoda vraća zbir i razliku dva broja

```
method Abs(x: int) returns (y: int)
{
   if x < 0
      { return -x; }
   else
      { return x; }
}
```

Ovo je primer metode koja za ulazni parametar tipa int vraća njegovu apsolutnu vrednost. U telu metode korišćena je standardna „if“ naredba za postavljanje uslova da li je broj negativan.

Preduslovi i postuslovi (ensures and requires)

Gore navedeni primeri mogu se napisati u bilo kom imperativnom jeziku. Prava moć Dafni-ja leži u anotacijama koje određuju ponašanje ovih metoda. Na primer konkretno želimo da rezultat Abs metode bude veći ili jednak nuli. To su uslovi koje metoda mora da ispunjava. Jedan od najjednostavnijih načina su preduslovi i postuslovi metoda.

```
method Abs(x: int) returns (y: int)
   ensures 0 <= y
{
   if x < 0
      { return -x; }
   else
      { return x; }
}
```

Sada metoda Abs ima postuslov da kakav god da je kod u telu metode, ona mora da vrati pozitivnu vrednost. Pokušaj prevodjenja ove metode biće uspešna.

```
1 method Abs(x: int) returns (y: int)
2   ensures 0 <= y
3 {
4   if x < 0
5      { return -x; }
6   else
7      { return -x; }
8 }
```

Pokušaj prevođenja ove metode neće biti uspešna. Kompajler će prijaviti grešku tipa: „A postcondition might not hold on this return path.“ na liniji 7.

```
method MultipleReturns(x: int, y: int) returns (plus: int, minus: int)
   ensures minus < x
   ensures x < plus
{
   plus := x + y;
   minus := x - y;
}
```

Takodje možemo da navedemo više postuslova.

```
  ensures minus < x && x < plus
  ensures minus < x < plus
```

Možemo naravno i spojiti postuslove u jedan. Dva navedena uslova su ekvivalentna.

U primeru metode koja vraća zbir i razliku dva broja. Postuslovi su definisani da razlika mora biti manja od x a zbir mora biti veći od x. Napomenimo da ovde će ovde kompajler primetiti da postoji mogucnost da rezultat metode ne zadovoljavaju postuslov. Uopšte, postoje dva osnovna razloga greške u verifikaciji Dafni-ja :

- Specifikacije nisu u skladu sa kodom
- Situacije u kojima nije dovoljno „pametno“ da dokaže željena svojstva

U ovom slučaju Dafni je korektno prijavio grešku jer y je tipa int i može biti negativna vrednost. Onda zapravo važi: plus `<` `x` `<` `minus` `.`

U ovom slučaju možemo definisati preduslove koje Dafni takodje podržava. Preduslov bi bio: 0 `<` `y.` U tom slučaju Dafni će vratiti korektan rezultat.

```
method MultipleReturns(x: int, y: int) returns (more: int, less: int)
   requires 0 < y
   ensures less < x < more
{
   more := x + y;
   less := x - y;
}
```

Preduslov da je y mora biti pozitivna vrednost.

Tvrdjenja (assertions)

Za razliku od preduslova i postuslova tvrdjenja se navode u telu metode. Tvrdjenje nam omogućava da uslov koji se navodi važi uvek kada se dodje do tog mesta u kodu. Tvrdjenja su korisna kako u proveravanju jednostavnih matematičkih činjenica tako i mnogo kompleksnijim situacijama. Takodje, ovo je jedno moćno sredstvo za otkrivanje grešaka u anotacijama.

Pre nego što pogledamo kako funkcionišu tvrdjenja navedimo primer definisanja i deklarisanja lokalnih variabli.

```
var x: int := 5;
var y := 5;
var a, b, c: bool := 1, 2, true;
```

Prva linija je deklarisanje sa tipom, koji se može izostaviti u slučaju druge linije. Možemo deklarisati i više promenljivih (treća linija)

```
method Testing()
{
   var v := Abs(3);
   assert 0 <= v;
}
```

Ovde koristimo tvrdjenje da proverimo korektnost Abs metode.

Funkcije (funtions)

Za razliku od metode, koja može imati sve vrste naredbi u svom telu, telo funkcije se mora sastojati isključivo od jednog izraza sa ispravnim tipom. Funkcija ima korist jer može da se piše direktno u tvrdjenjima. Ova odlika nam čak dozvoljava da ne instanciramo novu lokalnu promenljivu kao i da ne moramo pisati preduslove i postuslove, već možemo direktno u tvrdjenju proveriti ispravnost.

```
function abs(x: int): int
{
   if x < 0 then -x else x
}
```

Funkcija abs isto kao i metoda Abs vraća apsolutnu vrednost od x.

```
assert abs(3) == 3;
```

Onda jednostavnom anotacijom tvrdjenja možemo proveriti korektnost funkcije abs.

Za razliku od metoda, funkcije se mogu pojaviti u izrazima. Videćemo to na jednostavnoj matematičkoj funkciji računanja Fibonačijevog niza.

```
function fib(n: nat): nat
{
   if n == 0 then 0 else
   if n == 1 then 1 else
                  fib(n - 1) + fib(n - 2)
}
```

Implementacija fibonačijevog niza. Tip nat se koristi za nenegativne cele brojeve. Funkcija, naravno ima eksponencijalnu složenost pa postoje i bolja rešenja.

Invarijante petlji (Loop invariants)

Petlje predstavljaju problem u Dafniju. To je zato što Dafni ne može unapred da zna koliko će iteracija biti kroz petlju, ali mora da uzme sve puteve kroz koje se program kreće da bi dokazao korektnost. Da bi se ovo omogućilo u Dafniju postoji još jedna anotacija koja daje invarijantu petlje. Podsećanja radi, invarijanta petlje omogućava uslov važi nakon svake iteracije.

```
method m(n: nat)
{
   var i: int := 0;
      while i < n
         invariant 0 <= i
      {
         i := i + 1;
      }
   assert i == n;
}
```

Primer jednostavne invarijante petlje

Sada možemo definisati elegantniji algoritam za računanje Fibonačijevog niza.

```
function fib(n: nat): nat
{
   if n == 0 then 0 else
   if n == 1 then 1 else
                  fib(n - 1) + fib(n - 2)
}

method ComputeFib(n: nat) returns (b: nat)
   ensures b == fib(n)
{
   if n == 0 { return 0; }
   var i: int := 1;
   var a := 0;
       b := 1;
   while i < n
      invariant 0 < i <= n
      invariant a == fib(i - 1)
      invariant b == fib(i)
   {
      a, b := b, a + b;
      i := i + 1;
   }
}
```

Zaustavljanje (Termination)

Problem kod pisanja invarijanti petlje je što oni proveravaju da li je uslov tačan u svakoj iteraciji ali ne I da li se petlja završava. Tome služi još jedna Dafnijeva anotacija, anotacija smanjivanja. Ona ima efekat uslova da se vrednost smanjuje svakom iteracijom ili rekurzivnim pozivom.

```
method m()
{
   var i := 20;
   while 0 < i
      invariant 0 <= i
      decreases i
   {
      i := i - 1;
   }
}
```

Primer jednostavne petlje sa zaustavljanjem

```
method m()
{
   var i, n := 0, 20;
   while i < n
      invariant 0 <= i <= n
      decreases n - i
   {
      i := i + 1;
   }

}
```

Umesto smanjivanja vrednosti, ako nam je potrebno da se povećava vrednost I možemo iskoristi jednostavan aritmetički trik

Nizovi (Arrays)

Dafni podrzava nizove kao strukturu podataka. Definišu se generički kao Array<T> gde je T neki proizvoljan tip elemenata u nizu. Nizovi su null-abilni, I imaju ugrađeno polje za dubinu. Pristup element je standardan kao u većini ostalih jezika preko uglastih zagrada u kojima se navodi indeks elementa. Dafni se stara o pristupu niza u okviru njegovih granica, te tu proveru ne treba navoditi.

```
method Find(a: array<int>, key: int) returns (index: int)
   ensures 0 <= index ==> index < a.Length && a[index] == key
{
   // Ovde mozemo pristupiti elementima niza u for petlji npr. I vratiti
   // odgovarajući indeks
}
```

Primer osiguravanja granica niza

Kvantifikatori (Quantifiers)

Ako želimo da nam Dafni ne javi grešku ako ne nađe element u nizu, možemo to raditi takozvanim kvantifikatorom. Najčešće je to univerzalni kvantifikator, koji postavlja uslov za sve elemente niza.

```
assert forall k :: k < k + 1;
```

Primer univerzalnog kvantifikatora

Sada možemo videti primer traženja elementa u nizu i ako ga ne nađe vratiće -1.

```
method Find(a: array<int>, key: int) returns (index: int)
   ensures 0 <= index ==> index < a.Length && a[index] == key
   ensures index < 0 ==> forall k :: 0 <= k < a.Length ==> a[k] != key
{
   index := 0;
   while index < a.Length
   {
      if a[index] == key { return; }
      index := index + 1;
   }
   index := -1;

}
```

Ova metoda prima listu int vrednosti i traži vrednost key promenljive. Ako je nađe, vraća indeks elementa u niz, inače vraća -1.

Navedeni primer kompajler neće validirati kao tačan, jer smo izostavili invarijante petlje. Dafni ne zna da petlja zapravo pokriva sve elemente, moramo mu to eksplicitno reći preko invarijanti.

```
method Find(a: array<int>, key: int) returns (index: int)
   ensures 0 <= index ==> index < a.Length && a[index] == key
   ensures index < 0 ==> forall k :: 0 <= k < a.Length ==> a[k] != key
{
   index := 0;
   while index < a.Length
      invariant 0 <= index <= a.Length
      invariant forall k :: 0 <= k < index ==> a[k] != key
   {
      if a[index] == key { return; }
      index := index + 1;
   }
   index := -1;
}
```

Ovaj kod će se kompajlirati bez problema. Napomena: ako bismo izostavili invarijantu na indeksu, kompajler bi prijavio „Index out of bounds“ grešku.

Predikati (Predicates)

Ako bismo hteli da vršimo binarnu pretragu niza, što je mnogo efikasnije. Neophodan uslov nam je da niz bude sortiran. Ovo možemo rešiti direktno kvantifikatorom unutar naše metode. Dafni nudi elegantnije rešenje: predikate.

Predikat je funkcija kroja vraća boolean vrednost. Ideja je da uzememo niz koji je sortiran, dakle treba nam funkcija koja za niz vraća true ako je niz sortiran, inače vraća false. Predikati čine kod kraćim I razumljivijim.

```
predicate sorted(a: array<int>)
   requires a != null
{
   forall j, k :: 0 <= j < k < a.Length ==> a[j] <= a[k]
}
```

Ovo je primer predikata za sortiranje, nema svoju povratnu vrednost jer će vratiti true u slučaju da lista ispunjava uslove ovog kvantifikatora. Napomena: ovaj kod se ne kompajlira jer zahteva read annotaciju.

Okviri (Framing)

Predikat naveden u primeru nije mogao da se prevede jer niz nije bio uključen u okvir za čitanje predikata. Razlog možemo da šta funkcija(predikat) može da pročita je da kada pišemo u memoriju, možemo biti sigurni da funkcije koje su čitale taj deo memorije, taj deo memorije ima istu vrednost kao i ranije.

```
predicate sorted(a: array<int>)
   requires a != null
   reads a
{
   forall j, k :: 0 <= j < k < a.Length ==> a[j] <= a[k]
}
```

Ispravljeni predikat. Sada kompajler prevodi bez problema. Anotacijom reads smo dali niz a funkciji na čitanje.

Read anotacija nije boolean izraz, kao ostale anotacije I može se pojaviti bilo gde zajedno sa preduslovom ili postuslovom. Ona određuje skup memorijskih lokacija kojima funkcija može da pristupi za čitanje, I garantuje da neko drugi neće promeniti vrednost, dok ova funkcija ne završi izvršavanje. Dafni proverava da li pristupamo u okviru funkcije nekoj memoriji koja nije dozvoljena, što znači da bilo koja funkcija unutar neke može čitati samo memorijski podskup te funkcije. Parametri funkcije koji nisu memorijske lokacije ne moraju biti navedeni u anotaciji.

Okviri takodje utiču na metode. Metodama je dozvoljeno da čitaju bilo koju memoriju, ali su u obavezi da navedu koji deo memorije hoće da menjaju. Na taj način Dafni vrlo precizno barata memorijom.

Okviri su primenljivi jedino na hip i pristupe memoriji preko referenci. Lokalne variable nisu smeštene na hipu tako da ne mogu biti navedene u read anotaciji. Tipovi kao što su setovi, sekvence, multisetiovi se takođe ponašaju kao lokalne variable ili integer-i. Nizovi I objekti su referentni tipovi i oni se čuvaju na hipu.

Primeri jednostavnih Dafni implemmentacija

U nastavku je dato nekoliko primera jednostavnih Dafni implementacija.

Sabiranje I množenje brojeva

Prvi primer je verifikacija jednostavnog algoritma sabiranja 2 broja ali inkrementalno. Verifikujemo i algoritam množenja 2 broja , množenje vršimo tako što dodajemo prvi operand neophodan broj puta, to jest koristimo napisanu metodu sabiranja. Takodje obratimo pažnju da je metoda Mul napisana rekurzivno, dok je Add napisana iterativno, uz pomoć invarijante petlje.

```
method Add(x: int, y: int) returns (r: int)
  ensures r = x+y;
{
  r := x;
  if (y < 0) {
    var n := y;
    while (n ≠ 0)
      invariant r = x+y-n ∧ 0 ≤ -n;
    {
      r := r - 1; n := n + 1;
    }
  } else {
    var n := y;
    while (n ≠ 0)
      invariant r = x+y-n ∧ 0 ≤ n;
    {
      r := r + 1; n := n - 1;
    }
  }
}
method Mul(x: int, y: int) returns (r: int)
  ensures r = x*y;
  decreases x < 0, x;
{
  if (x = 0) {
    r := 0;
  } else if (x < 0) {
    call r := Mul(-x, y); r := -r;
  } else {
    call r := Mul(x-1, y); call r := Add(r, y);
  }
}
```

Binarna pretraga niza

Sledeći primer je verifikacija ovog dobro poznatog algoritma koji u sortiranom nizu brojeva nalazi tražen broj u linearnoj složenosti. Pored samog algoritma binarne predrage u Dafniju moramo verifikovati i ulaz, to jest da su svi brojevi u sortiranom poretku. Za ovo kao što smo videli možemo koristiti predikatsku funkciju.

```
ıethod BinarySearch(a: array<int>, key: int) returns (result: int)
  requires a ≠ null;
  requires (∀ i, j • 0 ≤ i ∧ i < j ∧ j < |a| ⟹ a[i] ≤ a[j]);
  ensures -1 ≤ result ∧ result < |a|;
  ensures 0 ≤ result ⟹ a[result] = key;
  ensures result = -1 ⟹ (∀ i • 0 ≤ i ∧ i < |a| ⟹ a[i] ≠ key);
```

```
invariant 0 ≤ low ∧ low ≤ high ∧ high ≤ |a|;
invariant (∀ i • 0 ≤ i ∧ i < low ⟹ a[i] < key);
invariant (∀ i • high ≤ i ∧ i < |a| ⟹ key < a[i]);
```

Implementirani algoritam binarne pretrage napisani su u fajlu DafnyVerification.dfy. Fajl je preveden i konvertovan u .cs ekstenziju i pokrenut u C# konzolnoj aplikaciji u materijalima.

Implementacija mape

Dafni kao što je navedeno podržava i concept klasa što omogućava i verifikaciju u duhu Objektno-orijentisanog programiranja.

U ovom primeru ćemo verifikovati generičku klasu Map, koja nije deo Dafnija već smo je implementirali. Ona je predstavljena povezanom sekvencom ključeva I vrednosti, gde ključ k na poziciji I u sekvenci ključeva ima odgovarajuću vrednost v koje je takođe na poziciji i u sekvenci vrednosti. Generička mapa je implementirana povezanom listom čvorova gde svaki čvor sadrži ključ, vrednost i pokazivač na sledeći čvor.

```
class Map<Key,Value> {
  ghost var Keys: seq<Key>;
  ghost var Values: seq<Value>;
  ghost var Repr: set<object>;

  var head: Node<Key,Value>;
  ghost var nodes: seq<Node<Key,Value>>;

  function Valid(): bool
    reads this, Repr;
  { this in Repr ∧
    |Keys| = |Values| ∧ |nodes| = |Keys| + 1 ∧ head = nodes[0] ∧
    (∀ i • 0 ≤ i ∧ i < |Keys| ⟹
        nodes[i] ≠ null ∧ nodes[i] in Repr ∧
        nodes[i].key = Keys[i] ∧ nodes[i].key ∉ Keys[i+1..] ∧
        nodes[i].val = Values[i] ∧ nodes[i].next = nodes[i+1]) ∧
                  nodes[|nodes|-1] = null
  }

  method Init()
    modifies this;
    ensures Valid() ∧ fresh(Repr - {this}) ∧ |Keys| = 0;
  {
    Keys := [];
    Values := [];
    Repr := {this};
    head := null;
    nodes := [null];
  }

 method Add(key: Key, val: Value)
    requires Valid();
    modifies Repr;
    ensures Valid() ∧ fresh(Repr - old(Repr));
    ensures (∀ i • 0 ≤ i ∧ i < |old(Keys)| ∧ old(Keys)[i] = key ⟹
                |Keys| = |old(Keys)| ∧ Keys[i] = key ∧ Values[i] = va
                (∀ j • 0 ≤ j ∧ j < |Values| ∧ i ≠ j ⟹
                    Keys[j] = old(Keys)[j] ∧ Values[j] = old(Values)[
    ensures key ∉ old(Keys) ⟹ Keys = [key] + old(Keys);
         ensures Values = [val] + old(Values);
  {
    call p, n, prev := FindIndex(key);
    if (p = null) {
      var h := new Node<Key,Value>;

        h.key := key;  h.val := val;  h.next := head;
        head := h;
        Keys := [key] + Keys;  Values := [val] + Values;
        nodes := [h] + nodes;
        Repr := Repr + {h};
      } else {
        p.val := val;
        Values := Values[n := val];
      }
    }
   //...
  }
```

```
class Node<Key,Value> {
  var key: Key;
  var val: Value;
  var next: Node<Key,Value>;
}
```

Iteratori

Ovde ćemo verifikovati program koji koristi iteratore za neki kolekcijski tip. Za ovaj problem neophodno je bilo implementirati i klasu Iterator i klasu Collection. Kolekcijska klasa je implementirana kao sekvenca generičkih tipova sa metodama za inicijalizovanje kolekcije, vraćanje vrednosti, dodavanje vrednosti I dohvatanje iteratora iz kolekcije.

Program skladišti elemente kolekcije sekvencionalno i proverava da li iterator vraća ispravan element. Takođe proveravamo da iterator ne uništi kolekciju preko koje iterira.

```
class Collection<T> {
  var Repr: set<object>;
  var elements: seq<T>;

  function Valid():bool
    reads this, Repr;
  { this in Repr }

  method GetCount() returns (c:int)
    requires Valid();
    ensures 0 ≤ c;
  { c := |elements|; }

  method Init()
    modifies this;
    ensures Valid() ∧ fresh(Repr - {this});
  {
    elements := []; Repr := {this};
  }

  method GetIterator() returns (iter:Iterator<T>)
    requires Valid();
    ensures iter ≠ null ∧ iter.Valid();
    ensures fresh(iter.Repr) ∧ iter.pos = -1;
    ensures iter.c = this;
  {
    iter := new Iterator<T>;
    call iter.Init(this);
  }

  //...
}
```

```
class Iterator<T> {
  var c: Collection<T>;
  var pos: int;
  var Repr: set<object>;

  function Valid():bool
    reads this, Repr;
  { this in Repr ∧ c ≠ null ∧ -1 ≤ pos ∧ null ∉ Repr }

  method Init(coll:Collection<T>)
    requires coll ≠ null;
    modifies this;
    ensures Valid() ∧ fresh(Repr - {this}) ∧ pos = -1;
    ensures c = coll;
  {
    c := coll;
    pos := -1;
    Repr := {this};
  }

  method MoveNext() returns (b:bool)
    requires Valid();
    modifies Repr;
    ensures fresh(Repr - old(Repr)) ∧ Valid();
    ensures pos = old(pos) + 1 ∧ b = HasCurrent() ∧ c = old(c
  {
    pos := pos+1;
    b := pos < |c.elements|;
  }

  //...
}
```

Rezultati

Vremena izvršavanja verifikacije su redom : 3,5, 3,7, 4,9 I 3,9 sekundi. Rezultati su vrlo zadovoljavajući. Jedan veliki problem u ovim implemenacijama je to što Dafni koristi matematičke int-ove koje većina programskih ne koristi. Tako da u Dafni programima ne postoji mogućnost prekoračenja što je inače vrlo realan problem. Briga o prekoračenjima prevazilazi okvire ovog teksta. Za ovaj kao i druge probleme rešenja se mogu naći u zvaničnoj dokumentaciji Dafni jezika.

Zaključak

Dafni je jedan od alata koji se trenutno koristi u akademskim i industrijskim krugovima za konstrukciju pouzdanog softvera. Sa jedne strane Dafni je lak za učenje zato što sadrži većinu funkcionalnosti koje sadrže i najpopularniji programski jezici današnjice. Sa druge strane Dafni sadrži dokaze koji moraju da se ispune kao deo programskog koda, što umnogome olakšava upotrebu i povećava čitljivost programa, jer baš oni objašnjavaju korektnost algoritma. Dafni se trenutno ne koristi dovoljno ni u industriji ni u teorijskim okvirima ali imajući u vidu da je ovo relativno novi alat koji drugi programski jezici ne podržavaju dovoljno, može se u budućnosti očekivati veća upotreba ovog zanimljivog alata.