Dafny alat za verifikaciju

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

Dafny 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 *Dafny* jeziku su istinitosne vrednosti (eng. bool), celobrojne (eng. int) vrednosti, skupovi (eng. set). Kao što smo napisali *Dafny* sadrži klase, one mogu biti instancirane što daje neku vrstu objektno-orijentisane prirode ali *Dafny* ne podržava podklase samim tim ni nasleđivanje. *Dafny* kompajler proizvodi C# kod koji je sa svoje strane kompajliran do MSIL bajt koda za .NET platformu standardnim Microsoft C# kompajlerom. *Dafny* verifikator je deo kompajlera. Verifikator radi tako što prevodi *Dafny* jezik u *Boogie2* verifikacioni jezik na takav način da korektnost programa napisanog u *Boogie* jeziku implicira korektnost programa u *Dafny jeziku*. Posle se alati *Boogie* jezika koriste da generišu uslove logike prvog reda koja se prenosi do *Z3 SMT* rešavača. Najcešća upotreba *Dafny* jezika je u specifikaciji i verifikaciji nekih zahtevnijih algoritama.

Jezik *Dafny*

U ovom poglavlju ćemo objasniti neke osnove jezika. *Dafny* je jezik 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 *Dafny* 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. *Dafny* tada generiše dokaz da kod odgovara oznakama, ako su one tačne i 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 *Dafny* programa je metoda. Metoda je parče imperativnog, izvršivog koda. U ostalim programskim jezicima bi se moglo reći da je ovo funkcija, međutim ovde 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 zagrade je telo metode koje sadrži naredbe koje Dafny 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 (eng. ensures and requires)

Gore navedeni primeri mogu se napisati u bilo kom imperativnom jeziku. Prava moć *Dafny* jezika 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 prevođenja ove metode biće uspešan.

```
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 uspešan. 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;
}
```

Takođe 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 će ovde kompajler primetiti da postoji mogućnost da rezultat metode ne zadovoljava postuslov. Uopšte, postoje dva osnovna razloga greške u verifikaciji *Dafny* jezika:

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

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

U ovom slučaju možemo definisati preduslove koje *Dafny* takođe podržava. Preduslov bi bio: `0 < y.` U tom slučaju *Dafny* 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.

Tvrđenja (eng. assertions)

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

Pre nego što pogledamo kako funkcionišu tvrđenja navedimo primer definisanja i deklarisanja lokalnih promenljivih.

```
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 tvrđenje da proverimo korektnost „Abs“ metode.

Funkcije (eng. 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 tvrđenjima. Ova odlika nam čak dozvoljava da ne instanciramo novu lokalnu promenljivu kao i da ne moramo pisati preduslove i postuslove, već možemo direktno u tvrđenju 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 tvrđenje 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 (eng. loop invariants)

Petlje predstavljaju problem u *Dafny jeziku*. To je zato što *Dafny* 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 *Dafny* sadrži još jednu anotaciju koja daje invarijantu petlje. Podsećanja radi, invarijanta petlje omogućava da 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 (eng. 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 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 (eng. arrays)

Dafny podržava nizove kao strukturu podataka. Definišu se generički kao *Array<T>* gde je *T* neki proizvoljan tip elemenata u nizu. Nizovi podržavaju nedostajuće vrednosti(eng. *Nullable)* i imaju ugrađeno polje za dubinu. Pristup elementu je standardan kao u većini ostalih jezika preko uglastih zagrada u kojima se navodi indeks elementa. *Dafny* 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 možemo pristupiti elementima niza u for petlji npr. i vratiti
   // odgovarajući indeks
}
```

Primer osiguravanja granica niza

Kvantifikatori (eng. quantifiers)

Ako želimo da nam *Dafny* 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. *Dafny* 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 (eng. 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. *Dafny* nudi elegantnije rešenje: predikate.

Predikat je funkcija kroja vraća istinitosne vrednosti. 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 (eng. framing)

Predikat naveden u primeru nije mogao da se prevede, jer niz nije bio uključen u okvir za čitanje predikata. Razlog tome je što funkcija (ili predikat) nije sigurna da se niz a neće menjati u toku izvršavanja funkcije.

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

Ispravljen je predikat. Sada kompajler prevodi bez problema. Anotacijom „read“ smo dali niz a funkciji na čitanje.

„Read“ anotacija nije logički 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 niko drugi neće promeniti vrednost dok se ova funkcija ne završi. *Dafny* 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 takođe 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 *Dafny* vrlo precizno barata memorijom.

Okviri su primenljivi jedino na hip i pristupe memoriji preko referenci. Lokalne promenljive nisu smeštene na hipu tako da ne mogu biti navedene u *„read“* anotaciji. Tipovi kao što su skupovi, sekvence, multiskupovi se takođe ponašaju kao lokalne promenljive ili celobrojni tipovi. Nizovi i objekti su referentni tipovi i oni se čuvaju na hipu.

Primeri jednostavnih *Dafny* implemmentacija

U nastavku je dato nekoliko primera jednostavnih *Dafny* 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. Takođe 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) {
    r := Mul(-x, y);
    r := -r;
  } else {
    r := Mul(x-1, y);
    r := Add(r, y);
  }
}
```

Binarna pretraga niza

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

```
method BinarySearch(a: array<int>, key: int) returns (result: int)
    requires forall i, j :: 0 <= i < j < a.Length ==> a[i] <= a[j];
    ensures -1 <= result < a.Length;
    ensures 0 <= result ==> a[result] == key;
    ensures result == -1 ==> forall i :: 0 <= i < a.Length ==> a[i] != key;
  {
    var low := 0;
    var high := a.Length;

    while (low < high)
      invariant 0 <= low <= high <= a.Length;
      invariant forall i :: 0 <= i < low ==> a[i] < key;
      invariant forall i :: high <= i < a.Length ==> key < a[i];
    {
      var mid := low + (high - low) / 2;
      var midVal := a[mid];

      if (midVal < key) {
        low := mid + 1;
      } else if (key < midVal) {
        high := mid;
      } else {
        result := mid; // key found
        return;
      }
    }
    result := -1;  // key not present
  }
```

Implementirani algoritam binarne pretrage napisan je u fajlu **DafnyVerification.dfy**. Fajl je preveden i konvertovan u *„.cs“* ekstenziju i pokrenut u C# konzolnoj aplikaciji u materijalima.

Maksimum niza

Sada ćemo prikazazi algoritam koji pronalazi maksimum niza. Dodali smo dodatni uslov da nizu mora biti dodeljena memorija to jest da pokazivač mora biti različit od NULL. Takođe ako je nizu dodeljena memorija ali je trenutno prazan maksimalan element će biti 0.

```
method max(a:array?<int>) returns(max:int)
  requires a!=null;
  ensures (forall j :int :: (j >= 0 && j < a.Length ==> max >= a[j]));
  ensures (a.Length > 0)==>(exists j : int :: j>=0 && j < a.Length && max==a[j]);
 {
  if (a.Length == 0)  { max := 0;}
  else {
    max:=a[0];
    var i:int :=1;
    while(i < a.Length)
      invariant (i<=a.Length) && (forall j:int :: j>=0 && j<i ==> max >= a[j])
         && (exists j:int :: j>=0 && j<i && max==a[j]);
      decreases (a.Length-i);
    {
      if(a[i] > max){max := a[i];}
      i := i + 1;
    }
  }
}
```

Obrtanje niza

Metod “Reverse” vrši modifikaciju polaznog niza, tako što ga obrće.

```
method max(a:array?<int>) returns(max:int)
  requires a!=null;
  ensures (forall j :int :: (j >= 0 && j < a.Length ==> max >= a[j]));
  ensures (a.Length > 0)==>(exists j : int :: j>=0 && j < a.Length && max==a[j]);
 {
  if (a.Length == 0)  { max := 0;}
  else {
    max:=a[0];
    var i:int :=1;
    while(i < a.Length)
      invariant (i<=a.Length) && (forall j:int :: j>=0 && j<i ==> max >= a[j])
         && (exists j:int :: j>=0 && j<i && max==a[j]);
      decreases (a.Length-i);
    {
      if(a[i] > max){max := a[i];}
      i := i + 1;
    }
  }
}
```

Stepenovanje

U ovom slučaju verifikujemo metod *"Power1"* koji vrši stepenovanje dok funkciju *"power"* koristimo kao funkciju koja osigurava tačnost naše implementacije.

```
function power(n:int, e:nat):int
{
   if (e==0) then 1 else n * power(n,e-1)
}

method Power1(n:int, e:nat) returns (p:int)
ensures p==power(n, e);
{   var i:= 0;
    p:= 1;
    while (i!=e)
    invariant i<=e;
    invariant p==power(n, i);
    {
       i := i+1;
       p := p*n;
    }
}
```

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>
{
  ghost 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>;
    iter.Init(this);
  }
}

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

  function Valid(): bool
    reads this, Repr;
  {
    this in Repr && c != null && -1 <= pos
  }

  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 && c == old(c)
  {
    pos := pos + 1;
    b := pos < |c.elements|;
  }
}
```

Quick Sort

Sada ćemo predstaviti efikasniji algoritam sortiranja, takozvani quick sort. On u najgorem slučaju takođe ima kvadratnu složenost ali je u prosečnom logaritamska i smatra se jednim od najefikasnijih algoritama sortiranja. Takođe kao i kod *"Bubble sort-a"* imamo pomoćne metode. Metod *"sorted"* kao i u prethodnom primeru, predstavlja u stvari predikat koji samo proverava da li je deo niza sortiran. Dok metod "*partitioned "* vrši particionisanje niza a ne proverava samo da li je particionisan kao predikat u *"Bubble sort-u"*.

```
method QuickSort(a: array<int>, start: int, end: int)
   requires a.Length >= 1;
   requires 0 <= start <= end <= a.Length;
   requires 0 <= start <= end < a.Length ==>
    forall j :: start <= j < end ==> a[j] < a[end];
   requires 0 < start <= end <= a.Length ==>
    forall j :: start <= j < end ==> a[start - 1] <= a[j];
   modifies a;
   ensures sorted(a, start, end);
   ensures forall j :: 0 <= j < start || end <= j < a.Length ==> old(a[j]) == a[j];
   ensures 0 <= start <= end < a.Length ==>
    forall j :: start <= j < end ==> a[j] < a[end];
   ensures 0 < start <= end <= a.Length ==>
    forall j :: start <= j < end ==> a[start - 1] <= a[j];
   decreases end - start;
{
   if(end - start > 1)
   {
     var pivot := partition(a, start, end);
     QuickSort(a, start, pivot);
     QuickSort(a, pivot + 1, end);
   } else {
     return;
   }
}
predicate sorted (a: array<int> ,low: int , high :int )
  requires 0 <= low <= high <= a.Length;
  reads a;
{
```

```
  forall j,k ::low <= j < k < high ==> a[j] <= a[k]
}

method partition(a: array<int>, start: int, end: int) returns (pivot: int)
   requires a.Length > 0;
   requires 0 <= start < end <= a.Length;
   requires 0 <= start <= end < a.Length ==>
    forall j :: start <= j < end ==> a[j] < a[end];
   requires 0 < start <= end <= a.Length ==>
    forall j :: start <= j < end ==> a[start - 1] <= a[j];
   modifies a;
   ensures 0 <= start <= pivot < end <= a.Length;
   ensures forall j :: start <= j < pivot ==> a[j] < a[pivot];
   ensures forall j :: pivot < j < end ==> a[pivot] <= a[j];
   ensures forall j :: 0 <= j < start || end <= j < a.Length ==> old(a[j]) == a[j];
   ensures 0 <= start <= end < a.Length ==>
    forall j :: start <= j < end ==> a[j] < a[end];
   ensures 0 < start <= end <= a.Length ==>
    forall j :: start <= j < end ==> a[start - 1] <= a[j];
{
    pivot := start;
    var index := start + 1;
      while(index < end)
         invariant start <= pivot < index <= end;
         invariant forall j :: start <= j < pivot ==> a[j] < a[pivot];
         invariant forall j :: pivot < j < index ==> a[pivot] <= a[j];
         invariant forall j :: 0 <= j < start || end <= j < a.Length ==>
            old(a[j]) == a[j];
         invariant 0 <= start <= end < a.Length ==>
            forall j :: start <= j < end ==> a[j] < a[end];
         invariant 0 < start <= end <= a.Length ==>
            forall j :: start <= j < end ==> a[start - 1] <= a[j];
     {
         if(a[index] < a[pivot])
         {
          assert 0 < start <= end <= a.Length ==>
          forall j :: start <= j < end ==> a[start - 1] <= a[j];
           var counter := index - 1;
           var temp := a[index];
           a[index] := a[counter];
           while(counter > pivot)
              invariant forall j :: start <= j < pivot ==> a[j] < a[pivot];
              invariant forall j :: pivot < j < index + 1 ==> a[pivot] <= a[j];
              invariant a[pivot] > temp;
              invariant forall j :: 0 <= j < start || end <= j < a.Length ==> old(a[j]) == a[j];
              invariant 0 <= start <= end < a.Length ==>
                 forall j :: start <= j < end ==> a[j] < a[end];
              invariant 0 < start <= end <= a.Length ==>
                 forall j :: start <= j < end ==> a[start - 1] <= a[j];
           {
              a[counter + 1] := a[counter];
              counter := counter - 1;
           }
           a[pivot + 1] := a[pivot];
           pivot := pivot + 1;
           a[pivot - 1] := temp;
       }
       index := index + 1;
     }
}
```

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 *Dafny* koristi matematičke celobrojne vrednosti koje većina programskih ne koristi. Tako da u *Dafny* 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 *Dafny* jezika.

Zaključak

Dafny je jedan od alata koji se trenutno koristi u akademskim i industrijskim krugovima za konstrukciju pouzdanog softvera. Sa jedne strane *Dafny* je lak za učenje zato što sadrži većinu funkcionalnosti koje sadrže i najpopularniji programski jezici današnjice. Sa druge strane *Dafny* 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. *Dafny* 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.

Literatura

- Leino, K. R. M. (2010, April). Dafny: An automatic program verifier for functional correctness. In International Conference on Logic for Programming Artificial Intelligence and Reasoning (pp. 348-370). Springer, Berlin, Heidelberg. ISO 690
- Leino, K. R. M., & Wüstholz, V. (2014). The Dafny integrated development environment. arXiv preprint arXiv:1404.6602.
- Leino, K. R. M., & Monahan, R. (2010, August). Dafny meets the verification benchmarks challenge. In International Conference on Verified Software: Theories, Tools, and Experiments (pp. 112-126). Springer, Berlin, Heidelberg.
- Lucio, P. (2017). A Tutorial on Using Dafny to Construct Verified Software. arXiv preprint arXiv:1701.04481.