Programok helyessége

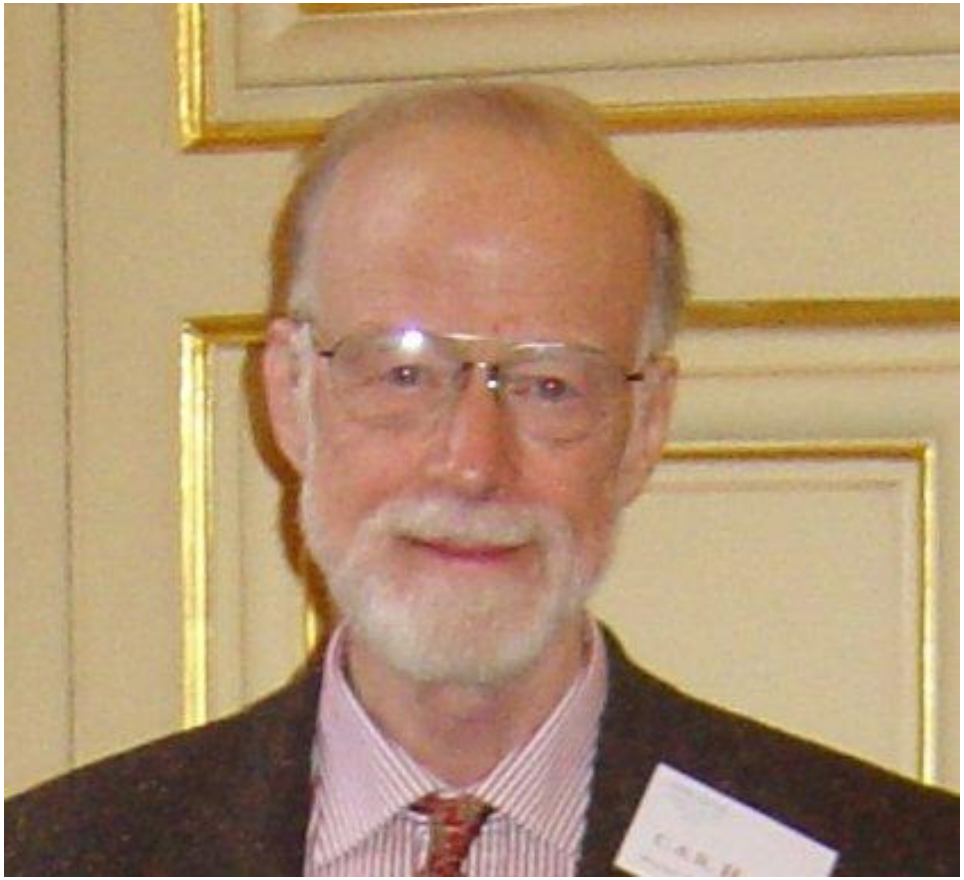
és a támogató nyelvi eszközök

Programok helyessége

- Feladat
- Algoritmust adok a megoldására
- Megoldja??
  - teszteljük…
    - fekete doboz, fehér doboz
    - nem biztos a megoldás ☹

Formális matematikai megközelítés kell!

Sir Charles Antony Richard Hoare (1934 -



brit tudós
- quicksort algoritmus
-„Hoare logic”
a programhelyesség igazolására
-strukturált programozás
-párhuzamos programozás
- Oxford
- most: Microsoft Research Lab. (Cambridge)

Feladat specifikációja

- **Szerződés** a felhasználó és az implementáló között
  - **előfeltétel:** egy állítás, ami leírja azt a feltételt, ami szükséges a feladat helyes működéséhez
  - **utófeltétel:** egy állítás, ami leírja azt a feltételt, amit a függvény teljesít a helyes végrehajtás után
- Helyesség a specifikáció figyelembevételével:
  - ha a függvény felhasználója teljesíti az előfeltételt, a függvény elkezd futni, s amikor befejezi, akkor az utófeltétel igaz lesz.
- (Mit kell az implementációnak teljesíteni, ha a felhasználó megsérti az előfeltételt?)

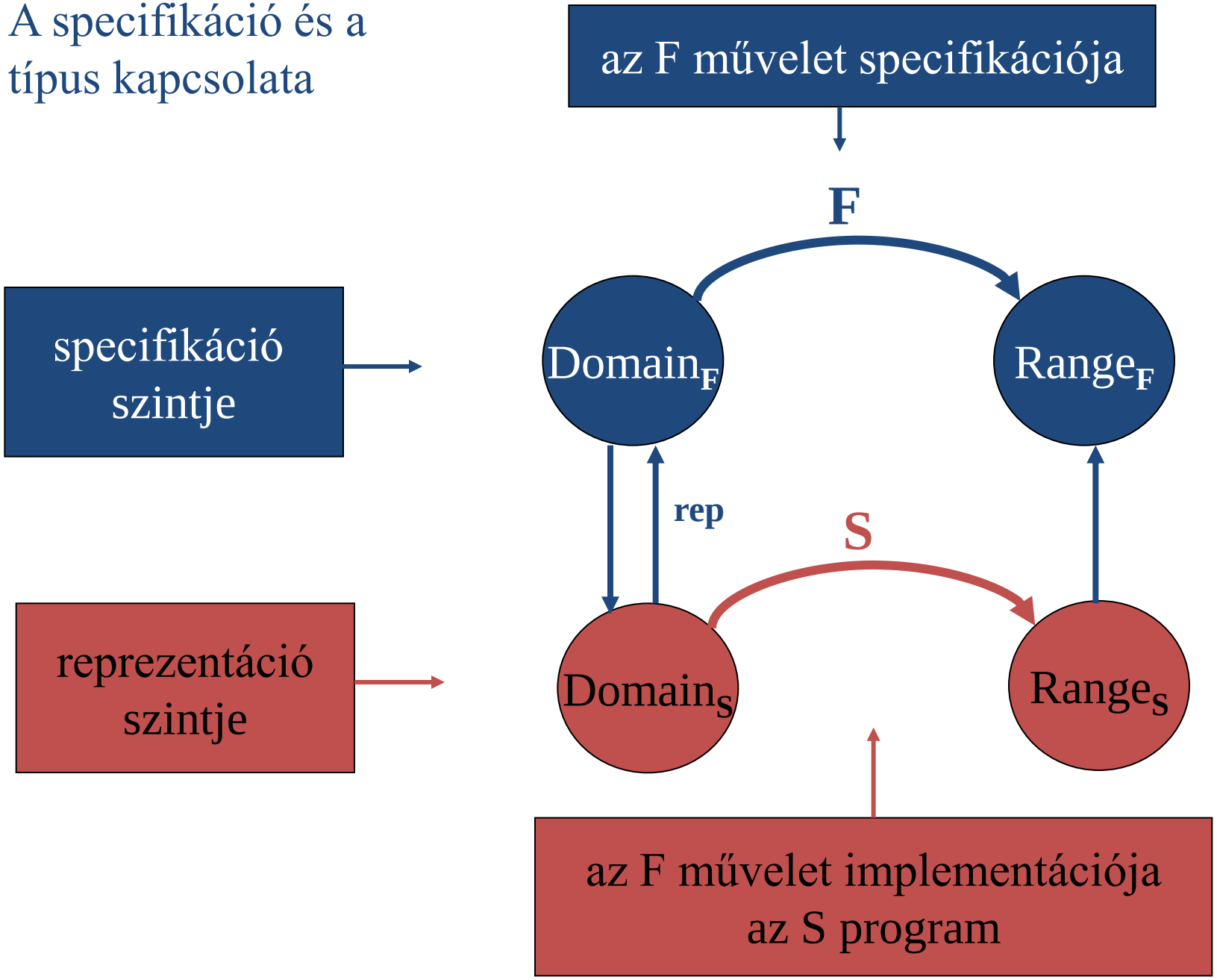
A strukturált és az objektumorientált programozási nyelvek központi fogalmává vált az absztrakció és az absztrakt adattípus.

- Ez a megközelítés azt jelenti, hogy a **típus**t a típusspecifikációval, a típusimplementációval és a közöttük levő kapcsolatot megadó reprezentációs függvénnyel definiáljuk.
- A helyességbizonyításhoz tehát olyan specifikációs módszerekre van szükségünk, amelyekkel a típusspecifikációt és a típusimplementációt adhatjuk meg. Ezt nevezzük kettős specifikációnak. Ebben az esetben a helyességbizonyítás során azt kell ellenőrizni, hogy az implementáció megfelel-e a specifikációnak.

A Hoare-féle specifikációban:

- A típusműveletekhez elő- és utófeltételeket rendelünk.
- A típusértékhalmazok leírása invariánsokat is tartalmazhat

A specifikáció és a típus kapcsolata



- Az elő- és utófeltételes specifikációt alkalmazza számos programozási nyelv
  - Közvetlenül pl.
    - az Alphard,
    - az Eiffel,
    - a D,
    - az Oxygene (korábban Chrome) (Object Pascal alapú)
    - a Cobra
    - stb.
  - Közvetve (valamilyen eszköz segítségével) pl.
    - C#, VB a *Code Contracts* használatával (ez egy Microsoft Research project, a .Net Framework 4.0-ba integrálva),
    - Java számos eszközzel, pl. iContract2, Contract4J, jContractor, Jcontract, Java Modeling Language (JML),
    - stb.

Alphard

- Az Alphardot a ’70-es évek második felében fejlesztették ki. A cél a Hoare-féle helyességbizonyításhoz egy megfelelő specifikációs eszköz kidolgozása volt, ezért sokan a “megváltó”-t látták benne. Végül azonban az implementálásig soha nem jutott el. ☹
- Mégis számos egyetemen oktatják a típusspecifikációs módszerek között, mert a specifikációs lehetőségei és a helyességbizonyításra való alkalmassága a legtöbb módszerhez képest sokkal kifinomultabbak.

Az Alphard a típusokat formoknak nevezi. Egy típust a következő módon kell megadni:

form típusnév (formális paraméterek) =
    beginform
    specifications
    ...
    representation
    ...
    implementation
        ...
    endform;

form istack (n: integer) =

beginform

specifications

requires n > 0;

let istack=<...,x_i,...> where x_i is integer;

-- egészekből álló sorozat

invariant 0 <= length (istack) <= n;

initially istack = nullseq;

-- az absztrakt objektum kezdeti tulajdonságai

```
function push (s: istack, x: integer)
                    pre 0 <= length (s) < n
                    post s = s' ~ x,          -- konkatenáció
            pop (s: istack)
                    pre 0 < length (s) <= n
                    post s = leader (s'),
            top (s: istack) returns x: integer
                    pre 0 < length (s) <= n
                    post x = last (s),
            isempty (s: istack) returns b: boolean
                    post b = (s = nullseq);
   representation
                ...
   implementation
                ...
endform;
```

- A requires a formális paraméterekre vonatkozó megszorításokat tartalmazza.
- A let kulcsszó után a típus leírására használt absztrakt adattípust adjuk meg (jelen esetben az integereket tartalmazó sorozatot).
- Az invariant a spec. típusinvariánst írja le. A példányosítás majd a konkrét térben fog megtörténni.
- A reprezentációs függvény, amelyet a representation-ben kell megadni, fog visszaképezni erre az absztrakt adattípusra.
- Minden konkrét térben inicializált objektumra alkalmazzuk a reprezentációs függvényt és az így kapott “absztrakt objektumoknak” ki kell elégíteniük az initiallyban megadott feltételt.
- Az initially az absztrakt kezdeti objektum tulajdonságait írja le.
- A function záradékban az absztrakt adattípus műveleteit specifikáljuk elő-utófeltételekkel.

```
form istack (n: integer) =
    beginform
      specifications
          ...
      representation
          unique   v: vector (integer, 1, n)
                     sp: integer init sp <- 0;
          rep (v, sp) = seq (v, 1, sp);
          invariant 0 <= sp <= n;
          states
                    empty when sp = 0,
                    normal when 0 < sp < n,
                    full when sp = n,
                    error otherwise;
```

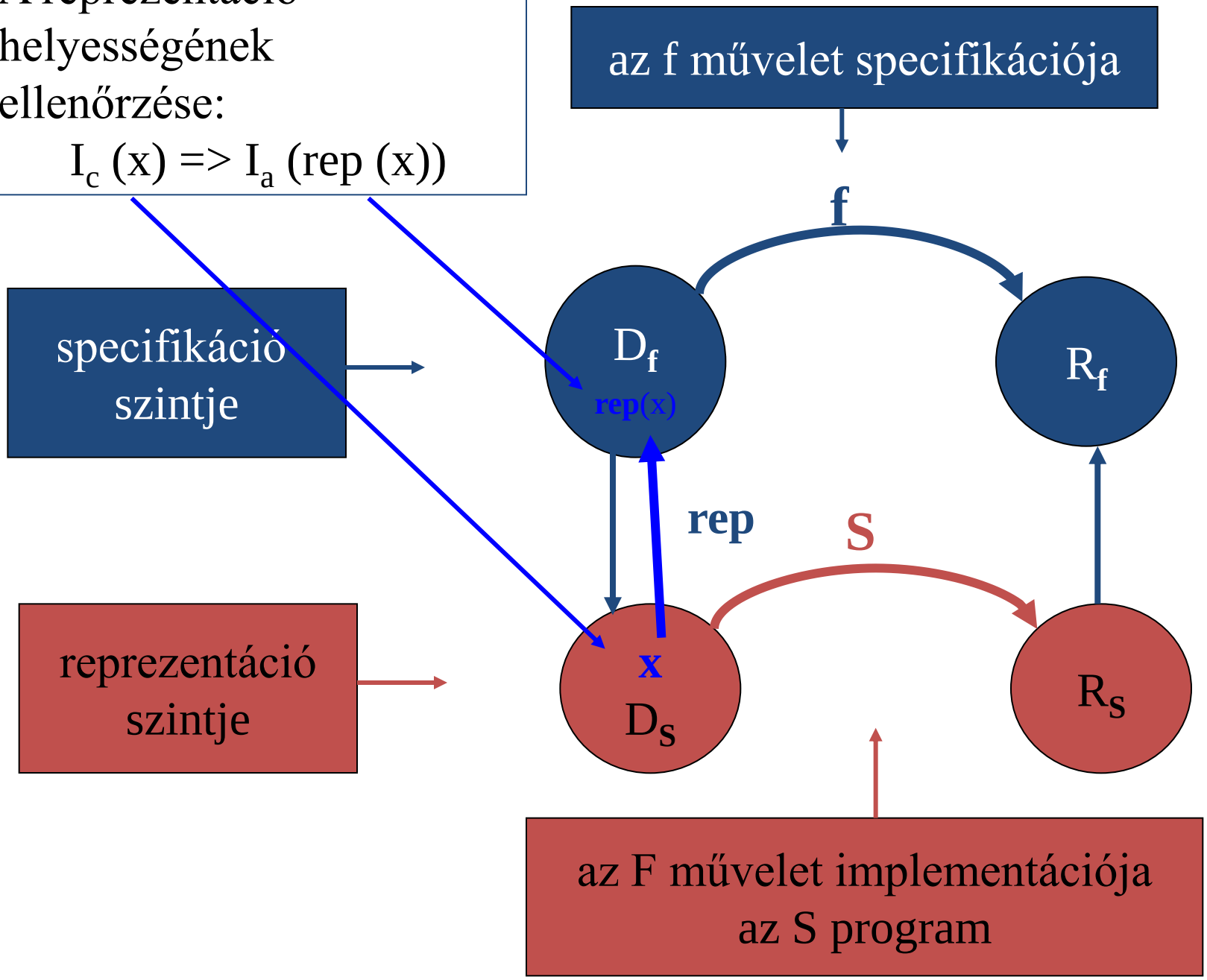
Implementation

```
body push out (s.sp = s.sp' + 1 ∧ s.v = α(s.v', s.sp, x)) =
        empty, normal:: (s.sp <- s.sp + 1; s.v [s.sp] <- x);
        otherwise:: FAIL;
    body pop out (s.sp = s.sp' – 1) =
        normal, full:: s.sp <- s.sp – 1;
        otherwise:: FAIL;
    body top out (x = s.v [s.sp]) =
        normal, full:: x <- s.v [s.sp];
        otherwise:: FAIL;
    body isempty out (b = (sp = 0)) =
        normal, full:: b <- false;
        empty:: b <- true;
        otherwise:: FAIL;
    endform;
```

- Hogyan láthatjuk be az így specifikált adattípus (form) helyességét?
- Jelöljük a specifications záradék invariánsát I_a-val, míg a representation záradékbeli invariánst I_c-vel. Jelöljük a specifications záradékban levő requirest β_{req}-val, az initiallyt pedig β_{init}-tel. Jelöljük az inicializáló műveletet f_{init} –tel, a többit f-fel, míg egy művelet végrehajtását { } jelekkel.
- Jelöljük az f absztrakt műveletekhez tartozó elő- és utófeltételeket pre_a^f –vel és $post_a^f$ –vel, míg a konkrét szinten pre_c^f –vel és $post_c^f$ –vel.

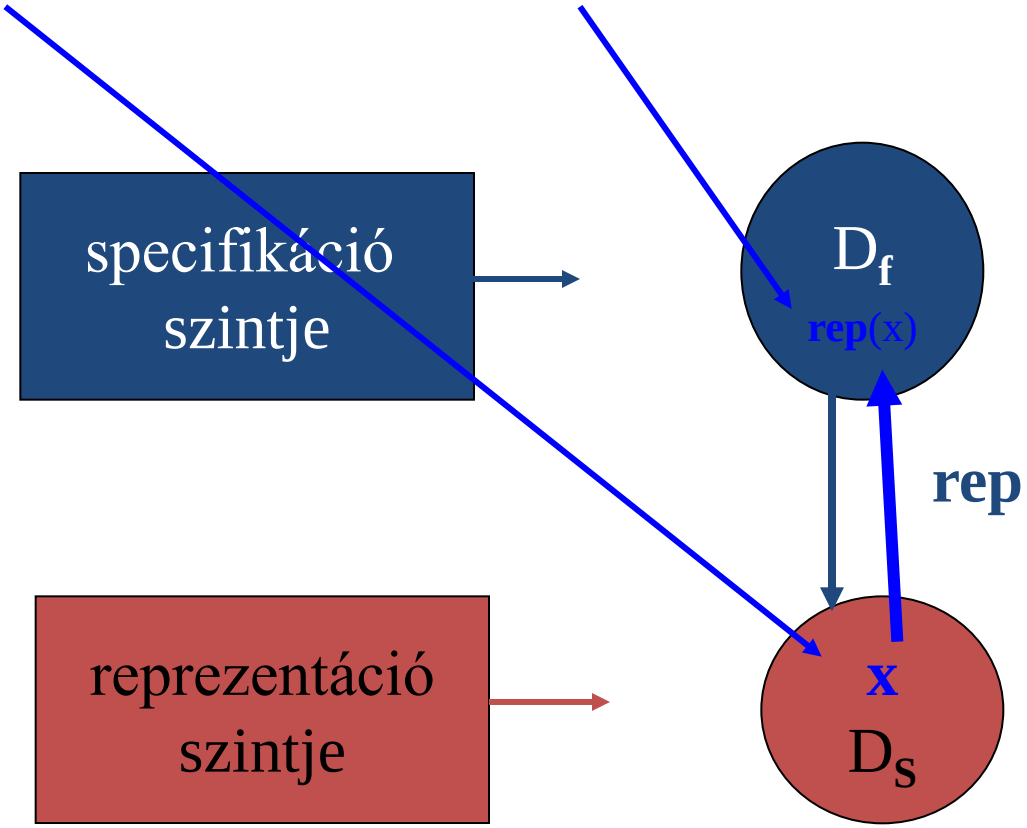
1. A reprezentáció helyességének ellenőrzése:

$$I_c(x) \Rightarrow I_a(rep(x))$$



2. Az objektum inicializálásának ellenőrzése:

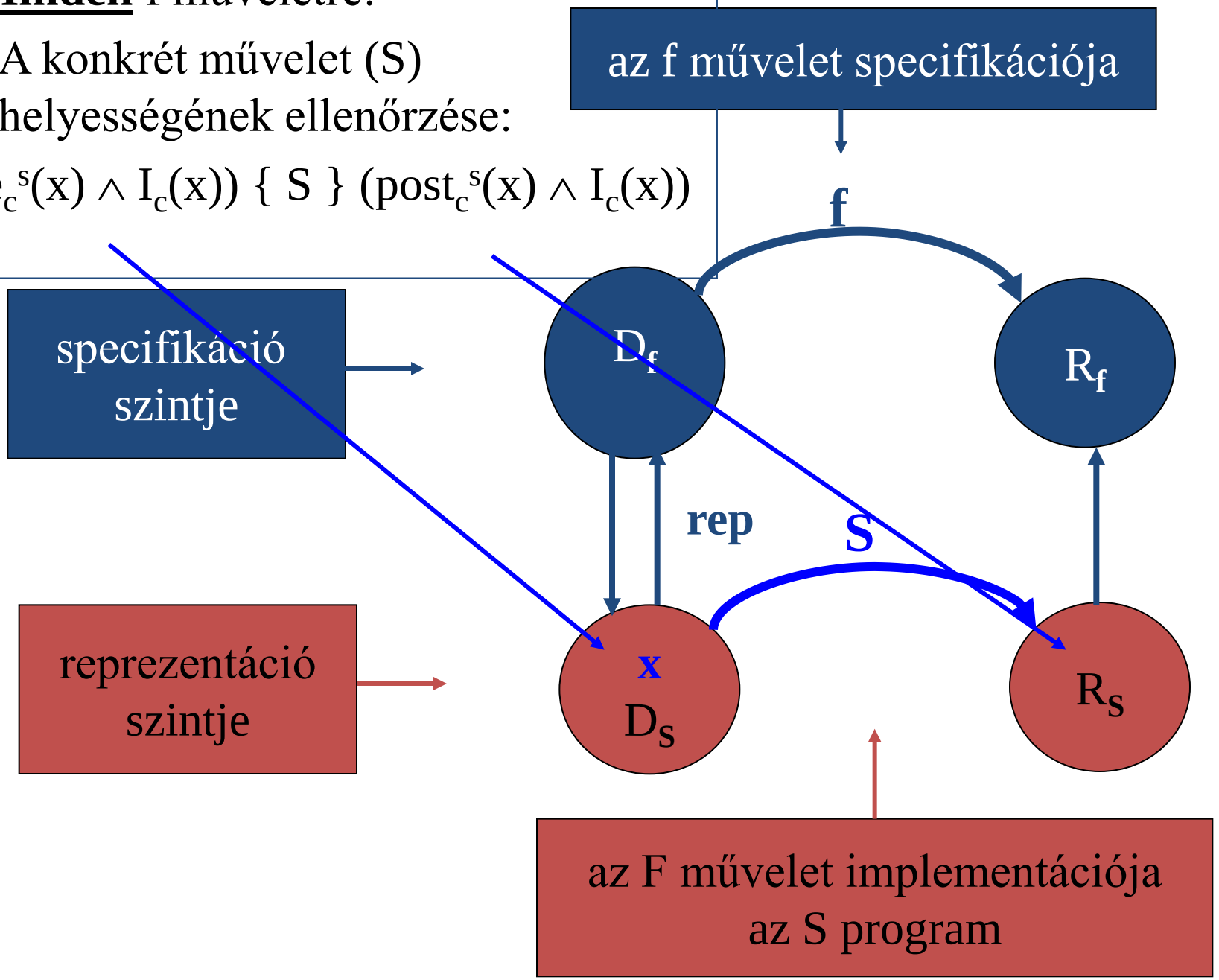
$\beta_{req}\ \{f_{init}\}\ (\beta_{init}\ (rep\ (x)) \wedge I_c\ (x))$



3. **Minden** f műveletre:

a. A konkrét művelet (S) helyességének ellenőrzése:

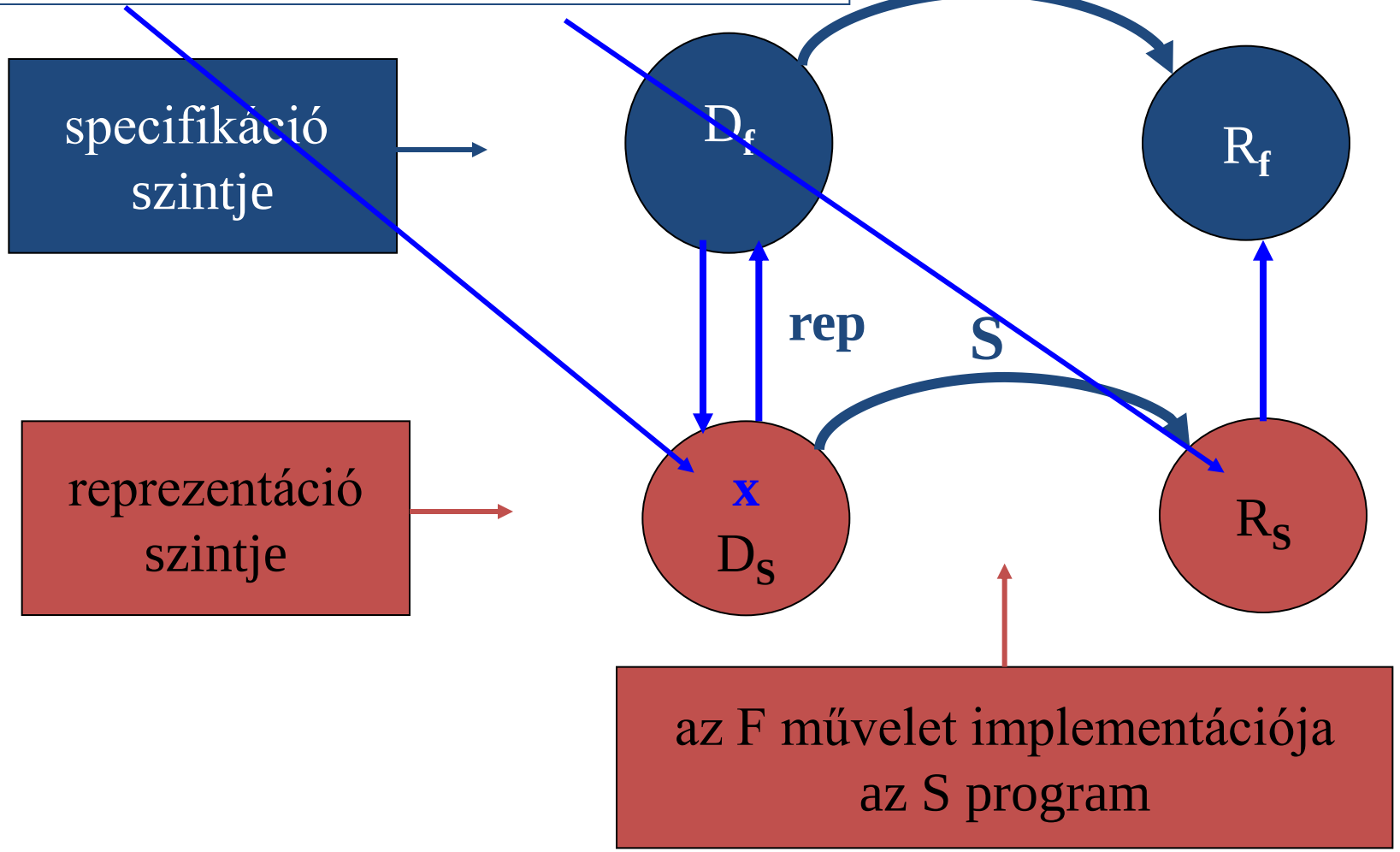
$(pre_c^s(x) \wedge I_c(x))\ \{\ S\ \}\ (post_c^s(x) \wedge I_c(x))$



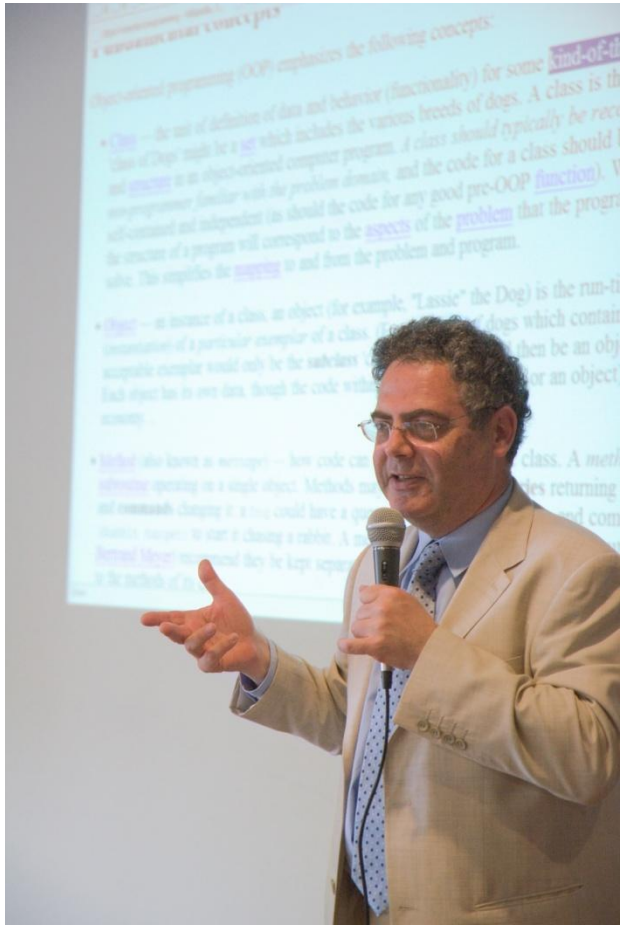
b. A konkrét és az absztrakt specifikáció közötti kapcsolat ellenőrzése:

(1) $I_c(x) \wedge pre_a^f(rep\ (x)) \Rightarrow pre_c^s(x)$

(2) $I_c(x) \wedge pre_a^f(rep\ (x)) \wedge post_c^s(x) \Rightarrow I_c(x) \wedge post_a^f(rep\ (x))$



Bertrand Meyer (1950-)



francia tudós
- „**Design by Contract**”
- „Object-Oriented Software Construction”
- Eiffel
- ETH Zürich

„Design by Contract”

- Programozás szerződéssel
  - Bertrand Meyer „Object-Oriented Software Construction”
  - 1986 óta
- Jogok és kötelezettségek
  - Az előfeltétel a megrendelő kötelezettsége
  - Az utófeltétel a teljesítést vállalóé

Eiffel

- Az Eiffel beépített nyelvi eszközökkel rendelkezik formális specifikációk megadására és az Eiffel futtató rendszer ellenőrizni is tudja, hogy az egyes programegységek nem sértik-e meg a specifikációt.
- A típusok műveleteinek elő-utófeltételes specifikációja fontos szerepet játszik az Eiffel módszertanában: felhasználják a program tervezése során a követelmények pontosabb megfogalmazására, de teszteléskor és dokumentációs célokra is. A nyelv tervezésekor ugyanakkor nem volt cél, hogy a fordítóprogram bizonyítsa a program helyességét, mert ez a gyakorlatban nem oldható meg.

Elő- utófeltételek

```
forth is
    -- Move forward one position.
    require
    not_after: not after
    do
    ... Some appropriate implementation
    ensure
    position = old position + 1
    end -- forth
put_i_th (v: like first; i: INTEGER) is
    -- Put item v at i-th position.
    require
    index_large enough: i >= 1;
    index_small_enough: i <= count;
    deferred
    ensure
    not empty
end – put_i_th
```

Osztályinvariáns

```
deferred class CHAIN feature
    ....
    invariant
    --Definitions:
    empty = (count = 0);
    off = ((position = 0) or (position = count + 1 ));
    isfirst = (position = 1 );
    islast = (not empty and (position = count));
    -- Axioms:
    count >= 0;
    position >= 0; position <= count + 1;
    empty => (position = 0);
    (not off) => (item = i_th (position));
    -- Theorems:
    (isfirst or islast) implies not empty
end -- class CHAIN
```

A C osztály *konzisztens:*

- A `C` osztály minden `p` konstruktor eljárására:
  $\{pre_p\}\ do_p\ \{INV_C\}$
- A `C` osztály minden `r` metódusára, amit általában, vagy szelektíven exportál:
  $\{pre_r \wedge INV_C\}\ do_r\ \{post_r \wedge INV_C\}$

- Kvantorok megadására nincs lehetőség, de használhatjuk pl . az *old* operátort:
  - Az old operátorral egy objektum végrehajtandó metódusába való belépése előtti állapotát jelöljük; Utófeltételben: x = old x + 1

Ciklushelyesség:

```
from --INIT
   go_before
invariant -- INV
  0 <= child_position; child_position <= arity + 1
variant --VAR
    arity - child_position + 1
until --EXIT
    child after or else (j = i)
loop -- BODY
   child_forth;
    if (sought = child) then
      j:=j+1
    end
end
```

Ciklusinvariáns és terminátor függvény

Ciklushelyesség

- A C osztály r metódusa akkor és csak akkor *ciklushelyes*, ha minden ciklusra:
  1. {True} INIT {INV}
  2. {True } INIT {VAR>=0 }
  3. {INV and then not EXIT} BODY { INV }
  4. { INV and then not EXIT and then (VAR=v) }
     BODY { 0<=VAR<v }

Exception helyesség

- Egy C osztály egy r rutinja exception-helyes akkor és csak akkor, ha a rescue blokkjának minden '**b**' ágára a következő igaz:
  - Ha b egy retry-jal végződik :
    { true } b {INV_C and pre_r }
  - Ha b nem retry-jal végződik:
    { true } b { INV_C }

Öröklődés és helyesség

- Az öröklődés során az elő-utófeltételek újradeklarálhatóak, amelyre az Eiffel a *require else ...* alternatív előfeltételt és az *ensure then ...* extra utófeltételt adja.
- Legyenek pre_1, ...pre_n az ősök előfeltételei, $post_1$,...$post_n$ az ősök utófeltételei,
  ezek mind or else kapcsolatban lesznek az előfeltételnél és and then kapcsolatban az utófeltételnél
  (az esetek többségében n=1).
- Ekkor a metódus
  előfeltétele:

alternatív előfeltétel or else pre_1 or else ... or else pre_n
utófeltétele:

alternatív utófeltétel and then $post_1$ and then ... and then $post_n$

- A DbC ezt úgy fogalmazza meg, hogy egy rutint újradefiniálni azt jelenti, hogy *alszerződést* kötök a leszármazottal, amire a kliensek az ősével már kötöttek. Egy becsületes alvállalkozó legalább olyan jól teljesíti a követelményeket, mint az eredeti. Ez azt jelenti:
  - megtartja, vagy gyengíti az előfeltételt, vagyis nincs új követelménye a kliens felé,
  - megtartja vagy szűkíti az utófeltételt, vagyis az eredmény legalább olyan jó, mint az eredeti szerződésben volt.

- A program tetszőleges pontján elhelyezhetünk egy állítást, amelyet a futtató rendszer ellenőriz. Erre a *check* utasítás szolgál.
- A környezet számos opciója segíti, hogy melyik állítást értékeljük ki. A lehetséges szintek:
  no - nem ellenőrzi az állításokat,
  require - mindig az előfeltételeket ellenőrzi
  (ez a default!),
  ensure - előfeltételek és utófeltételek ellenőrzése,
  invariant - az előzőeken túl az osztályinvariáns is,
  loop - az előzőeken túl a ciklusok ellenőrzése,
  all - az előzőeken túl a *check* utasítások ellenőrzése.
  Az egyes szintek jelentősége főleg a tesztelésnél.

D nyelv

Assert:

legegyszerűbb feltétel: ha nem teljesül, AssertException-t dob

```
int i;
Symbol s = null;
for (i = 0; i < 100 ; i++)
{
    s = search(array[i]);
    if (s != null) break;
}
assert (s != null); //megtaláltuk-e a keresett elemet
```

D nyelv

Elő és utófeltételek

`in { … } out { … } body { … }`

- in és out elhagyható
- Ha csak body van, akkor a „body” tag is elhagyható
- in: a hívás előtt fut le
- out: a blokk elhagyásakor fut le, bárhogyan is lépünk ki belőle
- body: a függvény / blokk törzse
- Szabály: in és out bármilyen utasítást tartalmazhat, de a környezetet nem változtathatja meg (tipikusan assertekkel van tele)

D példa elő és utófeltételekre

```
long square_root(long x)
   in
   {
      assert(x >= 0);
   }

   out (result)
   {
     assert((result *result)<=x &&
                  (result+1) * (result+1) >x);
   }

   body
   {
     return cast(long)std.math.sqrt(cast(real)x);
   }
```

D - osztály invariáns

- Speciális tagfüggvény: invariant()
- Az osztály adattagjait nem módosíthatja
- A konstruktor elhagyásakor, a destruktor előtt, illetve minden egyes tagfüggvény hívása előtt és után lefut, hogy az objektum helyességét ellenőrizze

```
class Date {
    int day;
    int hour;
 invariant()
    {
        assert(1 <= day && day <= 31);
        assert(0 <= hour && hour < 24);
    }
}
```

Oxygene

-Object Pascal alapján .NET-re és Mono-ra.
-Elő és utófeltételek

Az előfeltételt a "require" kulcsszó után kell megadni. Igaznak kell lennie a metódusba való belépéskor. Például itt megkötéseket tehetünk a metódus paramétereire.

Az utófeltételnél megadott feltételek az előtt értékelődnek ki, mielőtt a metódus befejeződne és a vezérlés az őt hívó blokkba térne vissza. Az "ensure" kulcsszó után tehetjük

• Az elő és utófeltételeket egy osztály metódusánál lehet alkalmazni.

Oxygene

```
method MyObject.DoWork(aValue);
        require
          Assigned(fMyWorker);
          fMyValue > 0;
          aValue >0;
        begin
                  //… do the work here
        ensure
           Assigned(fMyResult);
           fMyResult.Value >= 5;
end;
```

Oxygene - invariáns

- "invariants" kulcsszó után lehet megadni.
-kétféle változata van:
    -"public invariants", - az összes publikus metódus végén leellenőrzi (az utófeltétel után).
    -"private invariants", ami az összes metódus végén leellenőrzi

Oxygene - invariáns

```
type
   MyClass = class;
      public
        … some methods or properties
                public invariants
                  fField1 > 35;
                  SomeProperty = 0;
                  SomeBoolMethod() and not (fField2 = 5);
                private invariants
                  fField > 0;
                end;
```

Cobra

- 2009. február 29-én, MIT licensz alatt publikált általános célú, open source programozási nyelv, még jelentős fejlesztés alatt áll.
- Chuck Esterbrook tervezte, a Python nyelvhez kapcsolódó fejlesztései miatt ismert.
- Ebből következik, hogy a Cobra jelentősen épít a Python alapjaira, a két nyelv szintaktikája majdnem teljesen megegyező,
- Microsoft .NET és Mono futtató környezetekhez készült megvalósítás, lesz majd más is.

Cobra

- A Cobra a contractok fogalmát átörökítette az Eiffel nyelvből.
- Megkövetelhetjük, hogy bizonyos elő-, utófeltételek és invariáns tulajdonságok teljesüljenek egy metódus végrehajtásakor.
- require - előfeltételek
- ensure - utófeltételek
- A feltételeket egymás alatt felsorolva, logikai kifejezések listájaként adhatjuk meg
- invariant - osztályinvariáns
- assert használható
- öröklődnek, a „szokásos” szabályok szerint

Cobra

```
class Person
    def drive(v as Vehicle)
        require
            not v.hasDriver
            v.isOperable
        ensure
            v.miles > old v.miles
        body
            …
```

Cobra

```
class ContiguousList<of T>
  implements IList<of T>
    def insert(index as int, item as T)
    require
      index >=0 and index < count
    ensure
     .count = old .count + 1
      this[index] is item
    body ...
```

Cobra – elő-utófeltétel öröklődés

```
class NonContiguousList<of T>
  inherits ContiguousList<of T>
  """
    Allows insertions past the end of the list.
  """
    def insert(index as int, item as T)
     or require index >= 0
    body ...
```

- utófeltételnél: **and ensure**

Cobra - invariáns

```
class Player
invariant
 .name.length
 .score >= 0
 .isAlive implies .health > 0
```

Java - assert

assert utasítás:

(**Programming With Assertions)**

Az utasítással különböző **elő- és utófeltételeket, invariánsokat fogalmazhatunk meg**.

Előnye a feltételes utasításokkal szemben: használata ki- és bekapcsolható.

Ha használni szeretnénk, akkor a -ea opcióval kell meghívnunk a JVM-et.

Ezután az assert utasítások kimenete láthatóvá válik.

Az utasításnak két alakja van:

assert logikai_kifejezés;

assert logikai_kifejezés : üzenet;

Java -assert

Például két változóról azt feltételezzük, hogy az egyik kisebb, mint a másik. Ekkor az utasítás így nézhet ki:

assert x < y;

Ha a feltétel nem teljesül, akkor az utasításnak ez a formája AssertionError hibát dob.
Ha az üzenetet is megadjuk, akkor azt is kiírja.

Fontos szabály, hogy az assert utasítás **feltételének nem szabad befolyásolnia az alkalmazás működését**, ugyanis az előfeltevések használata kikapcsolható.

Nem érdemes például metódushívást szerepeltetni benne. Egyébként minden feltétel szóba jöhet.

Java –assert példa

- ellenőrzi, hogy a beolvasott érték 0 és 10 között van-e:

```
import java.util.Scanner;
    public class AssertTest {
      public static void main( String args[] ) {
           Scanner input = new Scanner( System.in );
           System.out.print( "Enter a number between 0 and 10: " );
           int number = input.nextInt();
            // assert that the value is >= 0
assert ( number >= 0 && number <= 10 ): "bad number: " + number;
           System.out.printf( "You entered %d\n", number );
       } // end main
   } // end class AssertTest
```

```
Enter a number between 0 and 10: 5
You entered 5
```

Java –assert példa –folyt.

```
Enter a number between 0 and 10: 50

Exception in thread "main" java.lang.AssertionError:
bad number: 50
at AssertTest.main(AssertTest.java:15)
```

Java - jContractor

- A Design by Contract 100%-os Java megvalósítása
- A szerződéseket egy elnevezési koncepció alapján, függvények formájában adhatjuk hozzá az osztályhoz.
- A jContractorhoz nem tartozik külön specifikációs nyelv, a feltételeket Java nyelven fogalmazhatjuk meg.
- Nagy előnye, hogy használatához magán a könyvtáron kívül nincs szükség további eszközökre, módosított compilerre, előfordításra.

Java – jContractor előfeltétel

- Az **előfeltételt** külön függvényben kell definiálnunk.
  A boolean visszatérési értékű függvényt úgy nevezzük el, hogy az eredeti függvénynév után a **_Precondition** utótagot írjuk:

```
protected boolean push_Precondition (Object o) {
   return o != null;
}
```

- Az előfeltételt az eljárásba belépés előtt ellenőrizzük.
- Konstruktor esetén az ellenőrzés az ősosztály konstruktorának meghívása után történik.

```
protected boolean Stack_Precondition
   (Object [] initialContents) {
return (initialContents != null)&&(initialContents.length>0);
}
```

Java - jContractor - előfeltétel

- Szabályok:
  - A Contract metódusoknak nem lehet előfeltétele
  - Natív metódusoknak nem lehet előfeltétele
  - A main(String [] args) metódusnak nem lehet előfeltétele.
  - Egy static metódus előfeltétele static kell legyen
  - Egy nem static metódus előfeltétele nem lehet static
  - Egy nem-private metódus előfeltétele protected kell legyen
  - Egy private metódus előfeltétele private kell legyen

Java – jContractor - utófeltétel

- Az utófeltételeket az előfeltételekhez hasonlóan egy-egy külön függvényben tudjuk definiálni. Az utótag itt **_Postcondition**.
- Az utófeltétel boolean visszatérési értékű függvénye azokat az argumentumokat várja, amiket az eredeti függvény is, kiegészítve egy RESULT elnevezésű argumentummal -- ennek típusa az eredeti függvény visszatérési értékének típusa.

  Konstruktor esetén a RESULT Void típusú változó lesz.

```
protected boolean push_Postcondition (Object o, Void RESULT) {
   return implementation.contains(o) && (size()==OLD.size()+1);
}
```

- Az utófeltételben az objektum a függvénybe belépéskor érvényes állapotának megfelelő értékeit az **OLD** példányváltozón keresztül érjük el. Az OLD használatához az osztálynak meg kell valósítania a Cloneable interfészt, azaz biztosítania kell egy clone() tagfüggvényt az objektum másolásához.

Java – jContractor - utófeltétel

- Szabályok:
  - A Contract metódusoknak nem lehet utófeltétele
  - Natív metódusoknak nem lehet utófeltétele
  - Egy static metódus utófeltétele static kell legyen
  - Egy nem static metódus utófeltétele nem lehet static
  - Egy nem-private metódus utófeltétele protected kell legyen
  - Egy private metódus utófeltétele private kell legyen
  - A konstruktorok utófeltételei nem hivatkozhatnak az OLD-ra

Java – jContractor - invariáns

- Az invariáns ellenőrzését egy boolean visszatérési értékű, **_Invariant** nevű, argumentum nélküli protected függvényben valósítjuk meg.
- Az invariáns ellenőrzése a publikus függvényekbe való belépéskor és a publikus függvények, illetve a konstruktor lefutása után történik meg.

```
protected boolean _Invariant () {
return size() >= 0;
}
```

Java – jContractor - invariáns

- Szabályok:
  - A contract, a static és a natív metódusokra az invariánst nem ellenőrzi.
  - Az invariánst konstruktor esetén csak kilépéskor ellenőrzi.
  - Az _Invariant() metódus protected és nem-static kell legyen

Java – jContractor

- A jContractorban definiált szerződések öröklődése biztosított a szokásos értelemben, az előfeltételek gyengülhetnek, az utófeltétel és az invariáns tulajdonság erősödhet a leszármazott osztály kontraktusainak bevezetésekor.
- A szerződések megsértése esetén a rendszer edu.ucsb.ccs.jcontractor.PreconditionViolationError,
  PostconditionViolationError vagy
  InvariantViolationError hibát generál,
  a program futása megáll.

Java – jContractor

- **Szerződés-osztályok használata**

Az előzőekben felsorolt szerződéseket egy külön osztályba is kigyűjthetjük. Ezeknek az osztályoknak _CONTRACT utótagot kell tenni a nevébe:

```
class Stack_CONTRACT extends Stack {
  private Stack OLD;
  private Vector implementation;
  protected boolean Stack_Postcondition (Object [] initialContents,
    Void RESULT) {
     return size() == initialContents.length;
    } …
```

Ennek segítségével interfészek számára is definiálhatunk kontraktusokat

A kontraktusosztály metódusait a jContractor az eredeti osztály metódusaiként fogja értelmezni.

Java – jContractor - Kvantorok

A jContractor szerződésekben a JaQuaL könyvtárat (Java Quantification Library) használhatjuk az egzisztenciális és univerzális kvantorokat használó feltételek megfogalmazásához.

Négyféle JaQuaL kifejezést használhatunk:

- A **ForAll**.in(collection).ensure(assertion) kifejezéssel meggyőződhetünk arról, hogy egy kollekció minden elemére teljesül az assertionben megadott feltétel:

```
Assertion connected = new Assertion () {
   boolean eval (Object o) {
         return ((Node) o).connections >= 1;
   }
};
return ForAll.in(nodes).ensure(connected);
```

Java – jContractor - Kvantorok

- Az **Exists**.in(collection).suchThat(assertion) kifejezéssel az egzisztenciális kvantort tudjuk kiváltani, működése a ForAll-hoz hasonló.
- Az **Elements**.in(collection).suchThat(assertion) függvény egy Vectort ad vissza azokból az elemekből, melyek kielégítik az assertionben megadott feltételt.
- A Logical.**implies**(a, b) függvény a logikai következést valósítja meg: igazat ad vissza, ha az a hamis vagy ha a és b mindegyike igaz volt.

Java – jContractor - Kvantorok

- A JaQual rendelkezik beépített assertionökkel, ezek a következők:
  - InstanceOf: ellenőrzi, hogy egy objektum a megadott osztály példánya-e
  - Equal: ellenőrzi, hogy a két objektum megegyezik-e
  - InRange: ellenőrzi, hogy a megadott érték egy intervallumba esik-e
  - Not: egy másik kifejezés negálásához használható

Java - JML

Java Modelling Language (JML)

- A JML egy formális viselkedési interfész-specifikációs nyelv. A viselkedési specifikáció alatt azt értjük, hogy a Java osztályok interfészének szokásos szintaktikus leírásán (függvénynevek, láthatóság, visszatérési értékek stb.) túl az egyes függvények viselkedését is megpróbáljuk specifikálni a rendszerben.
- A JML nyelvben a szerződéseket megfogalmazó feltételeket annotációs megjegyzések formájában helyezhetjük el a programkódban.
- Ez lehet egy //*-gal kezdődő sor vagy egy /*@ ... @*/ blokk.
- A kifejezésekben a Java jelölésrendszerét használhatjuk, kibővítve néhány speciális elemmel.
- Az elkészült kódot a JML saját compilerével, a jmlc-vel kell lefordítanunk, de mivel a kontraktusokat megjegyzések formájában írtuk fel, a kód fordítható marad az eredeti Java fordítóval is.
- A kontraktusok ellenőrzése kikapcsolható.

Java - JML

```
public class IMath {
  /*@ requires (* x is positive *);
    @ ensures \result >= 0 &&
    @    (* \result is an int approximation to square root of
  x *)
    @*/
   public static int isqrt(int x) { … }
```

Java - JML

Bizonyos kiterjesztések használhatók:

Szintaxis	Jelentés
\result	eredmény
a ==> b	a –ból következik b
a <== b	b –ből következik a
a <==> b	a iff b
a <=!=> b	!(a <==> b)
\old(E)	E eredeti, belépéskori értéke

- Minősítők
  - Univerzális és egzisztenciális (\forall és \exists)
  - Általános (\sum, \product, \min, \max)
  - Numerikus (\num_of)

Függvény specifikációja

```
/*@ requires len >= 0
    @ ensures \result ==
    @                (\sum int j; 0 <= j && j < len; v[j])
    @*/
float sum (int v[], int len) {
   float s = 0.0;
   int i = 0;
   while (i < len) {
         s = s + v[i];
         i = i + 1;
   }
   return s;
}
```

JML - példák

```
public class IntegerSetf
...
  byte[] a; /* The array a is sorted */
  /*@ invariant
  (\forall int i; 0 <= i && i < a.length-1;
   a[i] < a[i+1]);
  @*/
```

JML - példák

```
public class BankAccount {
   final static int MAX_BALANCE = 1000;
   int balance;
   int debit(int amount) {
      balance = balance - amount;
      return balance; }
   int credit(int amount) {
      balance = balance + amount;
      return balance; }
  public int getBalance(){ return balance; }
...
```

JML - példák

- Elő- és utófeltétel:

```
/*@ requires amount >= 0;
      ensures
        balance == \old(balance)-amount &&
       \result == balance;
@*/
public int debit(int amount) {
...
}
```

JML - példák

- Invariáns:

```
public class BankAccount {
final static int MAX_BAL = 1000;
int balance;
/*@ invariant 0 <= balance &&
      balance <= MAX_BAL;
@*/
...
```

Invariánsok implicite hozzáadódnak az elő- és utófeltételekhez

Contracts for Java (cofoja)

- A Google –nál fejlesztették 2011-ben
- Annotációkat lehet használni
  - Típus invariáns – **Invariant** kulcsszó – minden publikus és csomagszintű láthatóságú metódus be- és kilépésnél, és a konstruktorok végén ellenőrzi, öröklődésnél **and** kapcsolat
  - Előfeltételek – **Requires** kulcsszó - metódus belépésnél ellenőrzi, öröklődésnél **or** kapcsolat
  - Utófeltételek – **Ensures** kulcsszó - metódus normál kilépésnél ellenőrzi, öröklődésnél **and** kapcsolat
  - Kivételes utófeltételek - **ThrowEnsures Checked** - ha exception lépett fel, ezt ellenőrzi
  - **old** és **result** lehetősége

Contracts for Java (cofoja) példa

```
@Invariant("size() >= 0")
interface Stack<T> {
  public int size(); …

  @Requires("size() >= 1")
  @Ensures({ "size() == old(size()) - 1",
             "result == old(peek())" })
  public T pop();
…
}
```

C# és VB

Code Contracts Library a .NET-ben:

http://research.microsoft.com/en-us/projects/contracts/

ingyen letölthető:

http://visualstudiogallery.msdn.microsoft.com/1ec7db13-3363-46c9-851f-1ce455f66970

- Code Contracts – utolsó verzió: 2013. augusztus 14.

Code Contracts Library

- Előfeltételek: Contract.Requires(…)
  - általában paraméterek ellenőrzésére
  - az előfeltételben szereplő összes tagnak elérhetőnek kell lennie az adott helyen (különben az előfeltételt nem tudja értelmezni a hívó metódus)
  - a megadott feltételeknek nem lehet mellékhatásuk.

Code Contracts Library

- Előfeltételek: Contract.Requires(…)
  - Példák:

- az x paraméter nem lehet null:

  Contract.Requires (x ! = null);

- ha az adott feltétel nem teljesül, milyen kivétel váltódjék ki:

  Contract.Requires<ArgumentNullException>
  (x != null);

Code Contracts Library

- „Örökölt” követelmények:

A legtöbb kódban a paraméterek ellenőrzésére if-then-throw szerkezet, ha ezek a metódus elején, át lehet alakítani előfeltételekké, de egy explicit contract metódushívás kell utána (pl.: Requires, Ensures, EnsuresOnThrow vagy EndContractBlock)

```
if ( x == null ) throw new ...
Contract.EndContractBlock();
// minden megelőző if-et előfeltételnek tekint
```

Code Contracts Library

- Utófeltételek: ellenőrzése az adott metódus végrehajtása után
  - Közönséges utófeltétel: Contract.Ensures()

    Contract.Ensures(this .F > 0);
  - Kivételes utófeltételek: Ha a metódus végrehajtása során kivétel váltódik ki, akkor is lehetőség van az utófeltétel ellenőrzésére. Ekkor a kivétel típusától (T) függően is lehet megadni feltételt:

    Contract.EnsuresOnThrow<T>(this.F > 0);

Code Contracts Library

- Speciális metódusok az utófeltételben:

(csak utófeltételek belsejében lehet használni)

Contract.Result<T>()

hivatkozik a T típusú visszatérési értékre (void típusú függvényeknél nem használható)

```
Contract.Ensures(0 < Contract.Result<int>());
```

Code Contracts Library

- **Contract.OldValue<T>(e)**
  az e kifejezés metódus hívása előtt értékét adja vissza.
  Megszorítások:
  - az e kifejezés nem tartalmazhat másik régi értéket lekérdező függvényt
  - csak olyan kifejezésre hivatkozhat, aminek létezett értéke a metódus meghívása előtt.

Code Contracts Library

Példák lehetséges hibákra:

1. A metódus visszatérési értékének a régi értékére nem lehet hivatkozni.

```
Contract.OldValue(Contract.Result<int>() + x) // ERROR
```

2. A régi érték nem függhet a visszatérési értéktől.

```
Contract.ForAll(0,Contract.Result<int>(),
  i => Contract.OldValue(xs[i]) > 3 ); // ERROR
```

Code Contracts Library

Contract.ValueAtReturn<T>(out T t)

az out paraméter értékének ellenőrzésére az utófeltételben.

```
public void OutParam(out int x){
Contract.Ensures(Contract.ValueAtReturn(out x) == 3);
                x = 3;
}
```

Code Contracts Library

Invariánsok (Invariants)

- Az összes invariáns void típusú függvény kell legyen, ha az adott osztályból lehet származtatni, akkor a láthatóságnak protected-et kell megadni.
- A [ContractInvariantMethod] attribútummal jelöljük, hogy az adott metódus egy invariáns
- Az invariáns ellenőrzése az összes publikus metódus végrehajtása után megtörténik. Ha az invariánson belül hivatkozunk az osztály egy másik publikus metódusára, akkor csak a legkülső függvénynél történik ellenőrzés.

Code Contracts Library

```
[ContractInvariantMethod]
        protected void ObjectInvariant()
        {
                Contract.Invariant( this.y >= 0 );
                Contract.Invariant( this.x > this.y );
                ...
        }
```

Code Contracts Library

- Contract.Assert - a program egy bizonyos pontján tudunk ellenőrizni:

```
Contract.Assert( this.privateField > 0 );
Contract.Assert( this.x == 3,
                 "Why isn't the value of x 3?" );
```

- Contract.Assume - Egy feltevést ír le, működése megegyezik az Assert-tel futási időben, de itt fordítási idejű ellenőrzés is van!

```
Contract.Assume( this.privateField > 0 );
Contract.Assume( this.x == 3,
                 "Static checker assumed this");
```

Code Contracts Library

Contract.EndContractBlock

ha az előfeltételek if-then-throw formában vannak leírva, akkor ez jelzi, hogy ezek előfeltételek, itt van az ellenőrző blokk vége:

```
if ( x == null ) throw
                        new ArgumentNullException("x");
if ( y < 0 ) throw
                    new ArgumentOutOfRangeException(...);
Contract.EndContractBlock( );
```

Code Contracts Library

- Contract.ForAll – ellenőrző ciklus, contract-on belül használható

Két paraméteres változat:

```
public int Foo<T>(IEnumerable<T> xs){
    Contract.Requires(
        Contract.ForAll( xs, (T x) => x != null) );
```

első paraméter egy kollekció, a második egy predikátum, ha ez igaz a gyűjtemény minden elemére, akkor igazat ad vissza, ha van olyan elem, amire hamis, akkor megáll, és hamisat ad vissza.

Code Contracts Library

Három paraméteres változat:

```
public int[] Bar(){
Contract.Ensures(
Contract.ForAll(0, Contract.Result<int[]>().Length,
    index => Contract.Result<int[]>()[index] > 0));
```

első paraméter az alsó határ,
a második a felső határ,
a harmadik a predikátum, aminek van egy index argumentuma
A két határ között bejárja a predikátumban adott kollekció elemeit, ellenőrzi, hogy igaz-e mindegyikre.

Code Contracts Library

- Contract.Exists - ugyanolyan paraméterei vannak, mint a ForAll-nak,
  akkor tér vissza igaz értékkel, ha a kollekció legalább egy elemére teljesül az adott predikátum és hamis értékkel tér vissza, ha egyre sem.

Code Contracts Library

Interfész contract-ok

- itt nem írhatunk függvény törzset, ezért egy külön contract osztály kell, amit az interfésszel attribútumok kapcsolnak össze.

```
[ContractClass(typeof(IFooContract))]
interface IFoo {
        int Count { get; }
        void Put(int value );
}
```

Code Contracts Library

- Interfész contract-ok (folyt.)

```
[ContractClassFor(typeof(IFoo))]
abstract class IFooContract : IFoo {
        int IFoo.Count {
            get {
                Contract.Ensures( 0 <= Contract.Result<int>() );
                        return default( int ); // dummy return
                        // lehet egy exc.-t is dobni itt
                        }
                }
        void IFoo.Put(int value){
                        Contract.Requires( 0 <= value );
        }
}
```

Code Contracts Library

- Absztrakt metódus szerződések

  – itt se lehet függvény törzset írni, ezért itt is egy külön contract osztály kell, amit az absztrakt osztállyal attribútumok kapcsolnak össze:

```
[ContractClass(typeof(FooContract))]
abstract class Foo {
        public abstract int Count { get; }
        public abstract void Put(int value );
}
```

Code Contracts Library

- Absztrakt metódus szerződések (folyt.):

```
[ContractClassFor(typeof(Foo))]
   abstract class FooContract : Foo {
      public override int Count {
            get {
         Contract.Ensures( 0 <= Contract.Result<int>() );
                        return default( int ); // dummy return
                        }
                }
     public override void Put(int value){
                        Contract.Requires( 0 <= value );
                }
      }
```

Code Contracts Library

Contract metódusok túlterhelése:

Mindegyik metódusnak lehet egy string típusú paramétere is.

Ez kiíródik, ha a feltétel nem teljesül. A string értékének fordítási időben ismertnek kell lennie.

```
Contract.Requires( x ! = null,
        " If x is null , then the missiles are fired ! " );
```

Code Contracts Library

Contract öröklődés

- Egy szerződés a típus altípusára is öröklődik.
- Az altípus előfeltételének gyengébbnek kell lennie ezért, ha az őstípusnál nincs megadva semmilyen előfeltétel, akkor az azt jelent, hogy az azonosan igaz az előfeltétele ezért az altípusokhoz nem lehet előfeltételt írni.
- Az utófeltételnél erősebbet kell / lehet megadni.

```
using System;
 using System.Diagnostics.Contracts;
 namespace ContractExample1 {
 class Rational {
   int numerator, denominator;
   public Rational( int numerator, int denominator) {
     Contract.Requires( denominator ! = 0 );
     this.numerator = numerator;
     this.denominator = denominator;
      }
   public int Denominator {
      get {
              Contract.Ensures( Contract. Result<int>() != 0 );
               return this.denominator;
              }
      }
 [ContractInvariantMethod]
protected void ObjectInvariant () {
   Contract.Invariant ( this.denominator ! = 0 );
      }
   }
}
```

Ada 2012

elő- és utófeltételek:

```
procedure Push(S: in out Stack; X: in Item)
  with
    Pre => not Is_Full(S),
    Post => not Is_Empty(S);
```

- lehet az eredeti értékre is hivatkozni az utófeltételben:
- írhatunk pl. Post => I = I'Old-ot vagy
- A(I)'Old-t, vagy A(I'Old)-t , stb.

Ada 2012

típusinvariáns:

```
type Stack is private
   with  Type_Invariant => Is_Unduplicated(Stack);


 type Disc_Pt is private
        with Type_Invariant => Check_In(Disc_Pt);
```

Az elő- és utófeltételes specifikációt alkalmazó programozási nyelvekkel kapcsolatos kérdések:

- Alprogramoknak megadhatunk-e, és ha igen, milyen formában elő- és utófeltételeket?
- A típushelyesség ellenőrzéséhez megadhatunk-e specifikációs- és típusinvariánst?
- Kezeli-e a specifikációs elő- és utófeltételek és az implementált program elő - és utófeltételeinek különbségét, és így ellenőrzi-e a megfelelés helyességét?
- Vannak-e eszközei a ciklushelyesség ellenőrzéséhez?
- Exception-helyességet támogatja-e? (Exception kezelésnél legalább a típusinvariánst helyre kell állítani.)
- Milyen az öröklődés és az elő- és utófeltételek, valamint a típusinvariáns kapcsolata?