

Szoftver- és rendszerellenőrzés

# A forráskód ellenőrzése

Majzik István

<http://www.inf.mit.bme.hu/>

# Tartalomjegyzék

- **Áttekintés**
  - Milyen a jó forráskód?
  - Kódolási szabályok
  - Forráskód metrikák
  - Milyen ellenőrzési módszerek vannak?
- **Statikus analízis eszközök**
  - Hibaminta keresők
  - Bővíthető eszközök
- **Dinamikus tulajdonságok vizsgálata statikus analízissel**
  - Absztrakt interpretáció

# Forráskód ellenőrzés

- Motiváció
  - Lehetséges hibák korai felderítése (még tesztelés előtt)
- Mikor jó egy forráskód?
  - Specifikációnak megfelelő funkciókat valósít meg
    - Dinamikus vizsgálata: **Tesztelés** (ld. később)
  - Programnyelv szintaxis szabályait betartja
    - **Fordító ellenőrzi**, betartása önmagában nem elég
  - A programozói (funkcionális) hibák kis esélyűek
    - Veszélyes, nehezen átlátható konstrukciók nincsenek
  - Újrafelhasználható, tesztelhető, módosítható
    - Áttekinthető, jól strukturált
    - Könnyen elemezhető
    - Megjegyzésekkel ellátott
    - ...

# Példa: Kritikus rendszerek fejlesztési szabványai

- Implementáció előírásai
  - Elemezhető programok: Műveleti szemantika van, követhető
  - Erősen típusos programnyelv: Típusellenőrzés van
  - Strukturált programozás: Átlátható vezérlési szerkezetek vannak
  - Objektum-orientált programozás (elfogadott)
- Programozási nyelv
  - SIL1-től nyomtatékosan ajánlott: Ada, Modula-2, Pascal
  - SIL1-től nem ajánlott: BASIC
  - SIL3-től nem ajánlott: BASIC, PLM, korlátozás nélküli C/C++
  - C és C++ ajánlott: kiegészítő kódolási szabályokkal!  
Nyelvi részalmaz definiálása szükséges
- Eszközök (fordítók, könyvtárak) használata
  - Tanúsított, validált, vagy gyakorlatban bevált eszközök

# Példa: Kódolási szabályok kritikus szoftverekben

- Irányelvek
  - Kód formázás és magyarázatok szükségesek
  - **Forráskód metrikákat** be kell tartani
- Korlátozott általános konstrukciók
  - Rekurzió
  - Automatikus típuskonverzió
  - Feltétel nélküli ugrás
- Korlátozott OO konstrukciók
  - Polimorfizmus, dinamikus kötés
  - Többszörös öröklődés
- Korlátozott dinamikus konstrukciók
  - Objektumok futásidejű létrehozása illetve törlése
  - Nem előre garantálható memória foglалás illetve felszabadítás

# Példa: SoHaR kódolási szabályok (nukleáris ipar)

Group	Number	Guideline
	1	Reliability
	1.1	Predictability of Memory Utilization
Specific	1.1.1	Minimizing Dynamic Memory Allocation
Outside	1.1.2	Minimizing Memory Paging and Swapping
	1.2	Predictability of Control Flow
Specific	1.2.1	Maximizing Structure
Specific	1.2.2	Minimizing Control Flow Complexity
Specific	1.2.3	Initialization of Variables before Use
Specific	1.2.4	Single Entry and Exit Points in Subprograms
Specific	1.2.5	Minimizing Interface Ambiguities
Specific	1.2.6	Use of Data Typing
General	1.2.7	Precision and Accuracy
Specific	1.2.8	Use of Parentheses rather than Default Order of Precedence
Specific	1.2.9	Separating Assignment from Evaluation
Outside	1.2.10	Proper Handling of Program Instrumentation
General	1.2.11	Control of Class Library Size
General	1.2.12	Minimizing Dynamic Binding
General	1.2.13	Control of Operator Overloading
	1.3	Predictability of Timing
Outside	1.3.1	Minimizing the Use of Tasking
Outside	1.3.2	Minimizing the Use of Interrupt Driven Processing

# C és C++ kódolási szabályok

- **MISRA C** (Motor Industry Software Reliability Association)
  - MISRA-C:2004: 142 szabály (122 szükséges jelölésű)
  - Példák:
    - Rule 33 (Required): The right hand side of a "&&" or "||" operator **shall not contain side effects**.
    - Rule 49 (Advisory): Tests of a **value against zero** should be **made explicit**, unless the operand is effectively Boolean.
    - Rule 59 (Required): The statement forming the body of an "if", "else if", "else", "while", "do ... while", or "for" statement **shall always be enclosed in braces**.
  - MISRA C:2012: 143 szabály + 16 direktíva
    - Szabályok: Statikus forráskód ellenőrzéshez
    - Direktívák: Folyamat, terv dokumentum alapján ellenőrizhető
- **MISRA C++ (2008): 228 szabály**
- **US DoD, JSF C++: 221 szabály (kód metrikák is)**
  - „Joint Strike Fighter Air Vehicle C++ Coding Standard”

# Példa: MISRA kódolási szabályok

- Kódsort nem szabad „kikommentezni”
  - Nehezedik a forráskód érthetősége
  - Áttekinthetetlenek az egymásba ágyazott kommentek
- Ciklusváltozót nem szabad a ciklus belsejében módosítani:

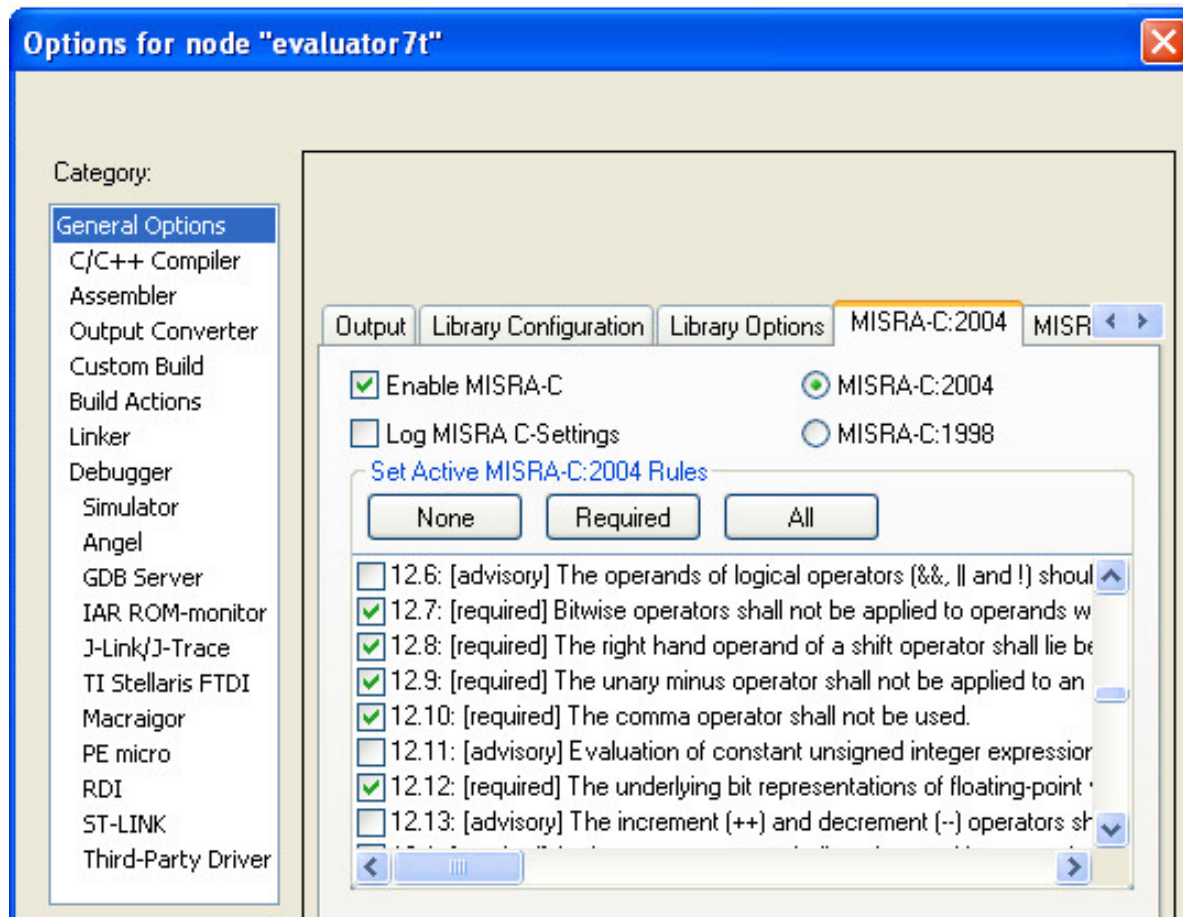
```
flag = 1;
for ( i = 0; (i < 5) && (flag == 1); i++ )
{
    /* ... */
    flag = 0; /* Compliant - allows early termination of loop */
    i = i + 3; /* Not compliant - altering the loop counter */
}
```

- Tiltott nyelvi elemek:
  - goto,
  - continue
- Bitmanipuláló műveleteket (>>, <<, ~, &, ^) nem szabad signed, vagy floating típusokon végrehajtani



# Példa: MISRA megfelelés ellenőrzése

- Eszközök a MISRA megfelelés ellenőrzéséhez
  - LDRA, PolySpace, IAR Embedded Workbench, QA-C, ...



# Példa: Compiler sajátosságokra való felkészülés

- Egész osztások ellenőrzése és dokumentálása:
  - $(-5/3)$  eredménye lehet -1 ahol a maradék -2, illetve
  - $(-5/3)$  eredménye lehet -2 ahol a maradék +1
- Változók összeadásakor, szorzásakor kicsúszás az értéktartományból:

```
uint16_t u16a = 40000;          /* unsigned short / unsigned int ? */
uint16_t u16b = 30000;          /* unsigned short / unsigned int ? */
uint32_t u32x;                  /* unsigned int / unsigned long ? */

u32x = u16a + u16b;             /* u32x = 70000 or 4464 ? */
```

- Az összeadás eredményét nem az eredmény tárolási típusa, hanem a compiler belső aritmetikája (belső tárolás módja) határozza meg;  
pl. ha a belső aritmetika 16 bites, túlcsordulás történhet!
- „Compiler verification kit” készíthető

# Szoftver metrikák

- Célkitűzések
  - Mérhető forráskód jellegzetességek meghatározása
  - Visszajelzés a metrikák alapján a forráskód minőségéről
- Minőségi szempontok a metrikákhoz (pl. MISRA):
  - Komplexitás
  - Karbantarthatóság
  - Modularitás
  - Megbízhatóság
  - Strukturáltság
  - Tesztelhetőség
  - Érthetőség
  - Kíérleltség
- A minőség mellett költségek is becsülhetők
  - Fejlesztés, tesztelés, módosítás költsége

# Példa: MISRA metrikák

Software Attributes	Type of Technique	Area of Application	Technique or Metric
Structuredness	Method	Component Source Code	Interval Reduction
	Metrics	Component Source Code	Cyclomatic Number Essential Cyclomatic Complexity Number of Entry Points Number of Exit Points Number of Structuring Levels Number of Unconditional Jumps Number of Execution Paths
Testability	Metrics	Component Source Code	Cyclomatic Number Number of Distinct Operands Number of Unconditional Jumps Number of Execution Paths Number of Decision Statements IB Coverage DDP Coverage LCSAJ Coverage PPP Coverage
		System Source Code	Number of Calling Paths Number of Components IB Coverage DDP Coverage LCSAJ Coverage PPP Coverage

Kód strukturáltság

Tesztelhetőség  
(ld. később)

IB: Instruction blocks  
DDP: Decision to decision paths  
LCSAJ: Linear code sequences and jumps  
PPP: Procedure to procedure paths

# Példa: MISRA metrikák korlátai

Operátorok és operandusok átlagos száma utasításonként

Komponensek átlagos száma hívási szintenként a hívási gráfban

Software Metric	Area of Application	High Level Languages		Low Level Languages	
		Min	Max	Min	Max
Average Statement Size	Component	2	8	N/A	N/A
Comment Frequency	Component	0.5	1	1	1
Component Length	Component	3	250	3	250
Component Stress Complexity	Component	1	10000	1	10000
Cyclomatic Number	Component	1	15	1	15
DDP Coverage	Both	80%	100%	80%	100%
Essential Cyclomatic Complexity	Component	1	1	1	1
Hierarchical Complexity	System	1	5	1	5
IB Coverage	Both	100%	100%	100%	100%
LCSAJ Coverage	Both	60%	100%	60%	100%
Number of Calling Levels	System	1	8	1	8
Number of Calling Paths	System	1	250	1	250
Number of Components	System	1	150	1	150
Number of Decision Statements	Component	0	8	0	8
Number of Distinct Operands	Component	1	50	1	50
Number of Distinct Operators	Component	1	35	1	35

# OO metrikák kategóriák szerint

- **Méret:** Forráskód elemek leszámmlálása
  - Kódsorok, attribútumok, metódusok (private / public / protected)
- **Komplexitás:** Ciklomatikus számok
  - CK: Független utak maximális száma a vezérlési gráfban
  - Metódusok ciklomatikus komplexitásainak összege
- **Csatolás:** Egy-egy osztály hány más elemet használ
  - Közvetlenül hívott metódusok száma
  - Hívott metódussal vagy használt attribútummal rendelkező osztályok száma
- **Öröklés:** Öröklési gráf jellege
  - Adott osztály alatti, fölötti szintek száma, közvetlenül / összesen
  - Öröklött metódusok száma
- **Kohézió:** Osztály metódusai és attribútumai
  - Közös attribútumot használó metódusok száma
  - Egymást hívó saját metódusok száma

# OO metrikák és a hibára való „hajlam” összefüggése

- Mérések: **Metrika** és a tesztelés során tapasztalt **hibaszám** összefüggése osztályonként
  - Nyílt forráskódú projektek (Mozilla, 4500 osztály)
  - Hibakövető rendszerben rögzített hibák elemzése (Bugzilla, 230 000 hiba)

## Nem hatékonyak bizonyult hiba előjelző metrikák:

- **Öröklés kategória**
  - **NOA** (Number of Ancestors): Az ősosztályok száma
  - **NOC** (Number of Children): A közvetlen leszármazottak száma
- **Kohézió kategória**
  - **LCOM** (Lack of Cohesion in Methods): Metóduspárok száma, amelyek nem használnak közös attribútumot, mínusz amik használnak

# OO metrikák és a hibára való „hajlam” (folytatás)

Hatékonyan bizonyult hiba előjelző metrikák osztályokra:

- **Csatolás kategória:**
  - **CBO** (Coupling Between Objects): Osztályok száma, amelyekhez kapcsolódik (használja a metódusát vagy attribútumát, vagy öröklődik)
  - **NOI** (Number of Outgoing Invocations): A közvetlenül hívott metódusok
  - **RFC** (Response Set of a Class): Az osztály metódusai + közvetlenül hívott metódusok száma
  - **NFMA** (Number of Foreign Methods Accessed): Közvetlenül hívott idegen, azaz nem saját és nem örökölt metódusok száma
- **Méret kategória:**
  - **NML** (Number of Methods Local): Az osztály lokális metódusainak száma
  - **LLOC** (Logical Lines of Code): A nem üres és nem komment sorok száma



# Milyen technikái vannak a verifikációnak?

- **Ellenőrző lista** a kód átvizsgáláshoz (átolvasás)
  - Tipikus általános hibák keresése
  - Kódolási szabályok kézi ellenőrzése
  - Struktúra elemzése
    - Vezérlési folyamat elemzés
    - Adatáramlás elemzése
    - Hibabecslés
- **Statikus analízis eszközök alkalmazása**
  - **Hibaminta keresés:** Tipikusan szintaxis, részben szemantikai
    - Vezérlés: Elérhetetlen kódrészlet, ...
    - Adatáramlás: Inicializálatlanság, elmaradt felszabadítás, ...
  - Hibaminták bővítése specifikus **kódolási szabályokkal**
  - **Mértékek ellenőrzése**
- **Dinamikus tulajdonságok vizsgálata statikus analízissel** (kód interpretáció, absztrakt interpretáció)
  - **Változók értéktartományának vizsgálata**
  - **Teljesítményproblémák vizsgálata**

# Tartalomjegyzék

- **Áttekintés**
  - Milyen a jó forráskód?
  - Kódolási szabályok
  - Forráskód metrikák
  - Milyen ellenőrzési módszerek vannak?
- **Statikus analízis eszközök**
  - Hibaminta keresők
  - Bővíthető eszközök
- **Dinamikus tulajdonságok vizsgálata statikus analízissel**
  - Absztrakt interpretáció

# Automatikus statikus analízis eszközök típusai

- Korai eszközök: kód „jólformáltság” ellenőrzése
  - Lint (C-hez, 1979, Bell Labs)
  - JLint (Java) később
- Hibaminta keresők
  - **Beépített** hibaminták + **bővíthetők** újabb hibamintákkal
  - Nem adnak garanciát a hibamentességre
  - Nem biztonságosak (pl. kimaradó hibák, téves jelzések lehetnek)
  - Példák: FindBugs, PMD (Java), Gendarme (.Net CIL), ...
- Absztrakt kód interpretációt támogató eszközök
  - Túlcsordulás, túlcímzés, redundáns feltételek ellenőrzése
  - Példák:
    - CodeSurfer, CodeSonar (C/C++, template alapú)
    - Prevent: MS COM, Win32 API, PThreads támogatása
    - Klocworks, QA-C, QA-C++, ...

# Java alapú eszközök jellemzői

- JLint
  - Minták alapján azonosít jellegzetes hibákat
    - Szintaxis alapján felismerhető hibák
    - Adatfolyamhoz kötődő hibák (akár bájtkódban is)
  - Gyors, de nem bővíthető szabályokkal
- FindBugs
  - Beépített hibaminták nagy halmazával dolgozik (bájtkódon is)
  - **Bővíthető** saját szabályokkal (szűrés)
  - Parancssoros, GUI, vagy Eclipse plugin
  - Nagy memóriaigény
- PMD
  - Java forráskód analízise
  - Betölthető „rule-set”
  - **Bővíthető** Java vagy XPath szabályokkal
  - Sok környezetbe integrált: JDeveloper, Eclipse, NetBeans, ...

# Példa: FindBugs hibakategóriák és példák

- **Bad practice**
  - Random object created and used only once
- **Correctness**
  - Bitwise add of signed byte value
- **Malicious code vulnerability**
  - May expose internal static state by storing a mutable object into a static field
- **Multithreaded correctness**
  - Synchronization on Boolean could lead to deadlock
- **Performance**
  - Method invokes toString() method on a String
- **Security**
  - Hardcoded constant database password
- **Dodgy**
  - Useless assignment in return statement

# Példa: PMD szabályok bővítése

```
class Example {  
    void bar() {  
        while (baz)  
            buz.doSomething();  
    }  
}
```

Szeretnénk, ha jelezné,  
amikor a while blokk körül  
nincsenek kapcsos zárójelek

```
public class WhileLoopsMustUseBracesRule extends AbstractRule {  
    public Object visit(ASTWhileStatement node, Object data) {  
        SimpleNode firstStmt = (SimpleNode)node.jjtGetChild(1);  
        if (!hasBlockAsFirstChild(firstStmt)) {  
            addViolation(data, node);  
        }  
        return super.visit(node,data);  
    }  
    private boolean hasBlockAsFirstChild(SimpleNode node) {  
        return (node.jjtGetNumChildren() != 0 && (node.jjtGetChild(0) instanceof ASTBlock));  
    }  
}
```

A szabály  
Java-ban  
kódolva

- AST reprezentáción dolgozik
- Beillesztendő a szabályok közé az AST adott helyén

# Példa futtatások\* 1.

```
public static void main(String[] args) {  
    String b = "bob";  
    b.replace('b', 'p');  
    if(b.equals("pop")){  
        System.out.println("Equals");  
    }  
}
```

A `String.replace()` függvény a megváltozott stringet a visszatérési értékében adja meg és nem módosítja a konkrét példányt, melynek tagfüggvényeként hívták.

## JLint:

java\lang\String.java:1: equals() was overridden but not hashCode().  
Verification completed: 1 reported messages.

## FindBugs:

Main.main(String[]) ignores return value of String.replace(char, char)

## PMD:

An operation on an Immutable object (String, BigDecimal or BigInteger) won't change the object itself

## Példa futtatások\* 2.

```
public class Main {  
    public static void chk(boolean s1, boolean s2){  
        if(s1 = s2) {System.out.println("foo");}  
        else {System.out.println("bar");}  
    }  
    public static void main(String[] args) {  
        boolean b1 = false;  
        boolean b2 = true;  
        Main.chk(b1, b2);  
    }  
}
```

'=' és '==' felcserélése  
Rossz kódrészlet fut le

### JLint:

Verification completed: 0 reported messages.

### FindBugs:

The parameter s1 to Main.chk(boolean, boolean) is dead upon entry but overwritten

Dead store to s1 in Main.chk(boolean, boolean)

### PMD:

No problems found!



## Példa futtatások\* 3.

```
public static void main(String[] args) {  
    int[] i = new int[3];  
    i[i.length] = 4;  
}
```

Tömb határának túllépése

**JLint:**

Main.java:1: Index [3,3] is out of array bounds.

**FindBugs:**

No errors.

**PMD:**

No problems found!

**A példa futtatások tanulságai:**

- Különbözők a „beépített” hibaminták, teljesség nem elvárható
- Adatfolyam analízise nehéz (nem mintakeresést igényel)

# Tartalomjegyzék

- **Áttekintés**
  - Milyen a jó forráskód?
  - Kódolási szabályok
  - Forráskód metrikák
  - Milyen ellenőrzési módszerek vannak?
- **Statikus analízis eszközök**
  - Hibaminta keresők
  - Bővíthető eszközök
- **Dinamikus tulajdonságok vizsgálata statikus analízissel**
  - Absztrakt interpretáció

# Dinamikus tulajdonságok statikus verifikációja

- Motiváció:

- Futás közben előkerülő **adathibák** megállapítása a program **végrehajtása nélkül** (a tesztelés előtt)
- Pl.: Nullával való osztás, tömb túlcímzés, ...

- Analógia:

## Optimalizáló fordítókban használatos elvek

- Élő (használatos) változók megállapítása
  - Egy regiszterbe kerülhetnek, amelyek nem egyszerre élnek
- Azonos értéket hordozó változók azonosítása
  - Konstansok használata változó olvasás helyett
- Sokszor használt rész-kifejezések keresése
  - Újrászámítás optimalizálható

# Statikusan detektálható futásidejű hibák

- Adatorientált hibák:
  - Null pointer
  - Tartományból kilógó pointer
  - Tartományból kilógó tömbindex
  - Inicializálatlan adat olvasása
  - Hozzáférési konfliktus megosztott változókon
  - Aritmetikai hiba
    - Nullával osztás
    - Negatív szám négyzetgyöke
    - Inverz szögfüggvények érvénytelen adatokon ...
  - Alulcsordulás, túlcsordulás
  - Veszélyes típuskonverzió
- Vezérlés orientált:
  - Kivételkezelés problémái
  - Nem elérhető kód

# Példa: PRQA QA-C, QA-C++ eszközök



## Security Issues:

- ✓ Buffer under- and overflow
- ✓ Arithmetic overflow and wraparound
- ✓ Format string mis-use

## Crash-Inducing Defects:

- ✓ Null pointer operations, invalid pointer values, operations on unrelated pointers
- ✓ Divide-by-zero
- ✓ Uncaught exceptions, throw-catch specification mismatches, improper exception use

## Flawed Logic Issues:

- ✓ Invariant (always true/false) logic and unreachable code
- ✓ Unset variables
- ✓ Redundant expressions, initializations and assignments
- ✓ Infinite loops
- ✓ Return value mismatches

## Memory Issues:

- ✓ Memory allocation mismatches
- ✓ Memory leaks

## API Mis-use:

- ✓ Standard library pre- and post-condition verification

„A combination of SMT solver and in-house language and parsing expertise result in exceptionally accurate dataflow and semantic modelling of C and C++ code – a foundation for a set of unique analysis checks.”

# Detektálható futásidejű hiba (1. példa)

```
10: int ar[100];  
11: int *p=ar;  
12: int i;  
13: for (i=0; i<100; i++; p++)  
14:     { *p=0; }  
15: *p=5;
```

Out-of-bound pointer in line 15

## Detektálható futásidejű hiba (2. példa)

```
20: int ar[10];
21: int i,j;
22: for (i=0; i<10; i++)
23: {
24:     for (j=0; j<10; j++)
25:     {
26:         ar[i-j] = i+j;
27:     }
28: }
```

Out-of-bound array access in line 26

# Hogyan működik a statikus analízis?

Vizsgált forráskód:

```
0: k=ioread32 ();
1: i=2;
2: j=k+5;
3: while (i<10) {
4:     i=i+1;
5:     j=j+3;
6: }
7: // end of loop
8: k = k / (i-j);
```

Kockázatos a  
0-val való osztás.  
Előfordulhat-e?  
Milyen k esetén?



# Mit tudunk az $(i, j, k)$ változók értékeiről?

$$X_0 = \{(0, 0, k) \mid k \in [-2^{31}, 2^{31} - 1]\}$$

$$X_1 = \{(2, j, k) \mid (i, j, k) \in X_0\}$$

$$X_2 = \{(i, k+5, k) \mid (i, j, k) \in X_1\}$$

$$X_3 = X_2 \cup X_6$$

$$X_4 = \{(i+1, j, k) \mid (i, j, k) \in X_3, i < 10\}$$

$$X_5 = \{(i, j+3, k) \mid (i, j, k) \in X_4\}$$

$$X_6 = X_5$$

$$X_7 = \{(i, j, k) \mid (i, j, k) \in X_3, i = 10\}$$

$$X_8 = \{(i, j, k) \mid (i, j, k) \in X_7\}$$

Mik lehetnek  
 $(i, j, k)$  értékei

Előző lépés alapján

Két helyről jöhet ide a  
program

Ciklus belseje

Ciklusból kilépés

# A változók értéktartományainak kiszámítása I.

- $X_0 = \{(0,0,k) \mid k \in [-2^{31}, 2^{31}-1]\}$

$$X_0 = \{(0,0,k) \mid k \in [-2^{31}, 2^{31}-1]\}$$

- $X_1 = \{(2,j,k) \mid (i,j,k) \in X_0\}$

$$X_1 = \{(2,0,k) \mid k \in [-2^{31}, 2^{31}-1]\}$$

- $X_2 = \{(i,k+5,k) \mid (i,j,k) \in X_1\}$

$$X_2 = \{(2,k+5,k) \mid k \in [-2^{31}, 2^{31}-1]\}$$

- $X_3 = X_2 \cup X_6$

$$X_3 = \{(i,j,k) \mid k \in [-2^{31}, 2^{31}-1], i \in [2,10], j = k + 3i - 1\}$$

- $X_4 = \{(i+1,j,k) \mid (i,j,k) \in X_3, i < 10\}$

$$X_4 = \{(i,j,k) \mid k \in [-2^{31}, 2^{31}-1], i \in [3,10), j = k + 3i - 4\}$$

$X_0$  által nyújtott információ figyelembevétele

Értékkadás a ciklusig és ciklus belseje

$j = k + 5 + 3(i-2)$   
a ciklusban

$i$  már nőtt, de  $j$  még nem kapott értéket

## A változók értéktartományainak kiszámítása II.

- $X5 = \{(i, j+3, k) \mid (i, j, k) \in X4\}$   
 $X5 = \{(i, j, k) \mid k \in [-2^{31}, 2^{31}-1], i \in [3, 10), j = k + 3i - 1\}$
- $X6 = X5$   
 $X6 = X5$
- $X7 = \{(i, j, k) \mid (i, j, k) \in X3, i = 10\}$   
 $X7 = \{(10, j, k) \mid k \in [-2^{31}, 2^{31}-1], j = k + 29\}$
- $X8 = \{(i, j, k) \mid (i, j, k) \in X7\}$   
 $X8 = \{(10, j, k) \mid k \in [-2^{31}, 2^{31}-1], j = k + 29\}$
- **X8-nál hiba, ha  $i - j = 0$**   
Mivel  $i = 10$ , ez akkor következhet be, ha  $k = -19$   
Itt  $X8_{\text{error}} = \{(10, 10, -19)\}$

$$j = k + 3i - 1, \\ \text{itt } i = 10$$

Teszteléssel nehezen lenne kideríthető.

# Az ellenőrzés alapelvei

- Adatfolyam alapú ellenőrzés
  - Tartományokkal (kényszerekkel) történő műveletek
  - Ciklus invariánsok felhasználása
- Általános célú nyelvekre az **invariánsok** számítása nem triviális
  - A megállási probléma erre visszavezethető lenne
  - Közelítő számítások szükségesek
- Absztrakció: A valódinál **bővebb tartományok** (lefedő tartományok) **biztonságosan** használhatók
  - **Lehetséges hibahelyek nem maradnak ki**: Hibahely esetén nem jelez hibamentességet (bővebb halmazból nem szökik meg a hiba)
  - **Téves hibajelzés lehetséges**: Ezek részletes analízisére van szükség (pl. teszteléshez tippet ad)
  - **Kód színezés**: „biztosan jó”, „biztosan hibás”, „gyanús” helyek

# Kód interpretációt támogató eszközök

- Absztrakt interpretációt támogató eszközök:
  - PolySpace C/Ada
    - Ariane 5 (70k kódsor), Flight Management System (500k kódsor)
  - Astrée
    - Airbus flight control software
  - C Global Surveyor
    - NASA Mars PathFinder, Deep Space One
- Annotáció alapú eszközök (design by contract):  
Ciklus invariánsok, elő- és utófeltételek explicit bevitele
  - ESC/Java (JML alapján)
  - Microsoft PreFix, PreFast, Boogie (Spec#, BoogiePL)
- Használatuk előnyei:
  - Statikusan detektált futásidejű hibák (tesztelés előtt)
  - Robusztussági problémák felderítése
  - Annotáció alapján monitor kód, teszt oracle is generálható
    - Pl. jmlc+jmlrac, jmlunit

# Miről volt szó?

- **Áttekintés**
  - Milyen a jó forráskód?
  - Kódolási szabályok
  - Forráskód metrikák
  - Milyen ellenőrzési módszerek vannak?
- **Statikus analízis eszközök**
  - Hibaminta keresők
  - Bővíthető eszközök
- **Dinamikus tulajdonságok vizsgálata statikus analízissel**
  - Absztrakt interpretáció