

# IML – Enhancements for Robust Programming

Implementing Design-by-Contract for IML



# Inhalt

- Was? / Warum?
  - Spracherweiterung / Beispiel
  - Vergleich
  - Checking
  - Code Generation
  - Demo
  - Tooling
  - Ausblick
-

## Was / Warum

- Unterstützung von Design by Contract
    - Mittels pre-postconditions
  - Robuste Software
  - Wachsender Support in anderen Sprachen/Umgebungen
    - .NET: Code contracts
    - Ada 2012
    - Eiffel, usw.
-

# Spracherweiterung - Beispiel

```
proc incrementPositive(inout var a: int32)
requires [ a > 0 : ParameterShouldBePositive]
ensures  [ a = old(a + 1), a > 0 : ResultShouldBePositive]
{
    a := a + 1
}
```

---

# Spracherweiterung – Lexikalische Syntax

|          |                   |
|----------|-------------------|
| [        | (Token: LBRACKET) |
| ]        | (Token: RBRACKET) |
| requires | (Token: REQUIRES) |
| ensures  | (Token: ENSURES)  |

# Spracherweiterung – Grammatikalische Syntax (1/3)

`requires ::= REQUIRES conditionList`

`ensures ::= ENSURES conditionList`

`conditionList ::= LBRACKET [condition {COMMA condition}] RBRACKET`

`condition ::= expr [COLON IDENT]`

---

# Spracherweiterung – Grammatikalische Syntax (2/3)

```
procDecl ::= PROC IDENT paramList  
          [GLOBAL globImpList]  
          [LOCAL cpsDecl]  
          [requires]  
          [ensures]  
          blockCmd
```

---

# Spracherweiterung – Grammatikalische Syntax (3/3)

```
funDecl ::= FUN IDENT paramList  
        RETURNS storeDecl  
        [GLOBAL globImpList]  
        [LOCAL cpsDecl]  
        [requires]  
        [ensures]  
        blockCmd
```

---



# Vergleich

- Eiffel sehr ähnlich wie unser Ansatz
- Abwärtskompatibel mit Standard IML
- In Sprache integriert (Java, C#)

```
procedure Update_Person (P : in out Person)
    with Post => P.Sex = P.Sex'Old
           and P.Birth_Date = P.Birth_Date'Old;

function Inc(X: Integer) return Integer
    with Pre  => X /= Integer'Last,
        Post => Inc'Result = X'Old+1;
```

---

# Checking - Erweiterungen

- Context
    - old nur in Postconditions
    - Keine Deklaration von old Funktion
    - Kein rekursiver Aufruf in Condition
  - Type
    - Conditions müssen boolean zurückliefern
-

## Checking - Erweiterungen

- Scope - Preconditions
    - Alle initialisierten Variablen(Parameter, Imports, in/inout)
  - Scope - Postconditions
    - Alle die in Precondition verfügbar sind
    - Return-Wert und out Parameter verfügbar
    - Zugriff auf old funktion
  - Scope – In Parameter der old-Funktion
    - Wie in Precondition
  - keine lokalen Variablen
-

# Code Generation

- Zielplattform JVM
  - Abbildung IML Programm -> JVM Klasse
  - Herausforderung IML Parameterhandling
    - Flowmodes / Mechmodes
  - Kompatibilität mit anderen JVM Sprachen
-

# Code Generierung – Mapping IML -> JVMKlasse

```
program outtest
```

```
global
```

```
  proc loadTwice(ref i: int32, out ref a: int32, out ref b: int32)
```

```
    requires[ i > 0 : exampleCondition ]
```

```
    ensures[ a = old(i), b = old(i) ]
```

```
  {
```

```
    a init := i;
```

```
    b init := i
```

```
  };
```

```
  a: int32;
```

```
  b: int32;
```

```
  var v: int32
```

```
{
```

```
  v init := 12;
```

```
  call loadTwice(v, a init, b init);
```

```
  ! a;
```

```
  ! b
```

```
}
```



```
public final class outtest {
```

```
  public void loadTwice(int, int[], int[]);
```

```
  public outtest();
```

```
  public void outtest();
```

```
  public static void main(java.lang.String[]);
```

```
}
```

# Code Generierung – JVM Bytecode

```
public void loadTwice(int, int[], int[]);
```

```
  flags: ACC_PUBLIC
```

```
  Code:
```

```
    stack=3, locals=4, args_size=4
```

```
      0: iload_1          // i auf Stack laden
      1: bipush           0    // 0 auf Stack laden
      3: if_icmple         10    // conditional jump (less equals)
      6: iconst_1          // true laden
      7: goto             11    // ans Ende springen
     10: iconst_0          // false laden
     11: ifeq              17    // falls Ergebnis false zu AssertionError springen
     14: goto             27    // Weiter mit normalem Code
     17: new                #9    // class java/lang/AssertionError
     20: dup
     21: ldc              #11    // Laden der String Konstante "exampleCondition"
     23: invokespecial     #15    // Method java/lang/AssertionError."<init>"
     26: athrow
     27: aload_2
     28: iconst_0
     29: iload_1
     30: iastore
     31: aload_3
     32: iconst_0
     33: iload_1
     34: iastore
     35: return
```

---

# Demo - Implementierung

- Kompletter IML-Sprachumfang
    - Funktionen / Prozeduren
    - Flowmodes / Mechmodes
    - Conditionals / Loops
    - Globals
  - Komplette Erweiterung
  - (fast) alle Context-Checks
-

Demo !





# Tooling

- Scala
  - Parser Kombinatoren
- ASM Bytecode Library
  - Schreiben der class-Files (z.B. checksumming)

```
def factor : Parser[Expr] = positioned( literal
    | (ident ~ Init)
    | (ident ~ exprList)
    | ident
    | (monadicOpr ~ factor)
    | (LParen ~> expr <~ RParen)

    ^^ {case l => l}
    ^^ {case i ~ init => StoreExpr(i, true)}
    ^^ {case i ~ list => FunCallExpr(i, list)}
    ^^ {case i => StoreExpr(i, false) }
    ^^ {case opr ~ fac => MonadicExpr(opr, fac)}
    ^^ {case rest => rest} )
```

---

# Ausblick

- Statische Verifikation zur Compilezeit
- Labels mit String
- Invarianten

# IML – Enhancements for Robust Programming

Implementing Design-by-Contract for IML

