Graduiertenkolleg „Werkzeuge zum effektiven Einsatz paralleler und verteilter Rechnersysteme“

LOOP – Ein Werkzeug zur Spezifikation und Verifikation von Klassen

Workshop Rathen ’97

Hendrik Tews – TU Dresden

Logic of **O**bject–**O**riented **P**rogramming

Spezifikation:

```
#include "uint.defs.h"
#include "uint.int.VHMap.h"

class Register
  {
    private:
      uintintVHMap contents;

    public:
      Register( void );

      int   read( uint address );
      void write( uint address, int data );
      void erase( uint address );
  };
```

Problem:

```
Register r;
int d;
...
r.write( 0, 10 );
r.write( 1, 20 );
d = r.read( 0 );
```

Inhalt

1. Modellierung von Klassen mittels Coalgebren
2. Konzeption von LOOP
3. CCSL: Designentscheidungen
4. Gegenwärtiger Entwicklungsstand
5. Ausblick

Coalgebraic **C**lass **S**pecification **L**anguage

explizite Typisierung der Methoden:

$$
\begin{aligned}
\mathsf{read} &: \mathsf{Self} \times \mathsf{Address} \longrightarrow \mathsf{Lift}[\mathsf{Data}] \\
\mathsf{write} &: \mathsf{Self} \times \mathsf{Address} \times \mathsf{Data} \longrightarrow \mathsf{Self} \\
\mathsf{erase} &: \mathsf{Self} \times \mathsf{Address} \longrightarrow \mathsf{Self}
\end{aligned}
$$

Zusammenfassung der Klasse zu einer Operation:

$$
\mathsf{next} : \mathsf{Self} \quad \longrightarrow \quad \left(\begin{array}{cl} & \big(\mathsf{Address} \Longrightarrow \mathsf{Lift}[\mathsf{Data}]\big) \\ \times & \big(\mathsf{Address} \times \mathsf{Data} \Longrightarrow \mathsf{Self}\big) \\ \times & \big(\mathsf{Address} \Longrightarrow \mathsf{Self}\big) \end{array} \right)
$$

Formalisierung bietet:

- ursprüngliche Operationen, z.B. für $r \in \mathsf{Self}$: $\quad r.\mathsf{read}(x) \;\cong\; \big(\mathsf{proj}_1\ (\mathsf{next}(r))\big)\ (x)$
- Axiome, Erzeugungsbedingungen
- Konzept „Verhaltensgleichheit"
- Beweisprinzip „Coinduktion"

Diese Folie wurde handgemalt und steht maschinenlesbar nicht zur Verfügung ;–(

- streng getypt, ungetypt
- Klassenbasiert — Objektbasiert
- Vererbungsrelation — Typrelation
- binäre Methoden
- parametrischer Polymorphismus, generische Klassen
- Überladen — Überschreiben
- Konstruktoren
- Umbenennen von Methoden

type ::=	
| *identifikator*	konstanter Typ
| **Self**	
| **[** *type* ⟶ *type* **]**	Funktionstyp
| **[** *typelist* **]**	Produkt (Record)
| *identifikator* **[** *arglist* **]**	Adt– oder Klasseninstanz
| **Lift [** *type* **]**	Partialität
| **List [** *type* **]**	
| **Coproduct [** *typelist* **]**	Variantentyp

typelist ::=
| *type*
| *typelist* **,** *type*

arglist ::=
| *argument*
| *arglist* **,** *argument*

argument ::=	
| *type*	Typparameter
| *expression*	Wertparameter
| *identifikator*	

```
type nat : Basetype

begin Register[ Data : Type, Address : Type ] : ClassSpec

    method
        read   : [Self, Address] -> Lift[Data];
        write  : [Self, Address,Data] -> Self;
        erase  : [Self, Address] -> Self
    Assertion
      read_write : PVS FORALL (a,b: Address, d: Data) :
            read(write(x, a, d), b) =
            IF a = b THEN up(d) ELSE read(x, b) ENDIF ENDPVS
      read_erase : PVS FORALL (a,b: Address) :
            read(erase(x, a),b) =
            IF a = b THEN bot ELSE read(x , b) ENDIF ENDPVS
    Creation
      read_new : PVS FORALL (a: Address) : read(new,a) = bot ENDPVS

end Register
```

```
begin BoundedRegister[ Data : Type, Address : Type, n:nat ] : ClassSpec
    Inherit from Register[Data,Address]
    method
        bounded_write : [Self,Address,Data] -> Self;
        count : Self -> nat
    Assertion
      bounded_write_def : PVS FORALL (a:Address, d: Data) :
            bisim?(bounded_write(x, a, d),
                   IF count(x) < n OR up?(read(x,a))
                   THEN write(x, a, d)
                   ELSE x ENDIF)     ENDPVS
      count_write : PVS FORALL (a: Address, d: Data) :
            count(write(x, a, d)) =
            IF bot?(read(x,a))
            THEN count(x) + 1 ELSE count(x) ENDIF ENDPVS
      count_erase : PVS FORALL (a: Address) :
            count(erase(x,a)) =
            IF bot?(read(x,a)) THEN count(x)
            ELSE max(0, count(x) -1) ENDIF ENDPVS

    Creation
      count_new : PVS count(new) = 0 ENDPVS
end BoundedRegister
```

Entwicklungsstand **LOOP/CCSL**

- Mitarbeiter in Nijmegen: Marieke Huisman, Bart Jacobs, Joachim van de Berg (Diplomant), in Dresden: Ulrich Hensel, Hendrik Tews
- Veröffentlichung eingereicht bei **E**uropean **S**ymposium **O**n **P**rogramming '98 (ETAPS '98) (`http://wwwtcs.inf.tu-dresden.de/~hensel/publications.html`)
- im Moment nur Entwicklung am CCSL Compiler
  - implementiert in ocaml
  - Kommentare
  - lokale Deklarationen
  - Typdeklarationen
  - Klassenspezifikationen
  - Überladen von Methoden, binäre Methoden
  - unvollständige Typüberprüfung
  - keine Fehlerbehandlung
- funktionierende Unterstützung von PVS
- erste Arbeiten an Java
- keine Ergebnisse zu Isabelle

theoretischer Hintergrund:

- neue Modellierung von Objekten in PVS
- „Implementation“ in Isabelle

Arbeit an **CCSL**

- „öffentliche“ Release
- „großes“ Beispiel
- primitive Typen, Ausdrücke
- richtiges Typchecking (Teiltypen?)
- Behandlung von Namenskonflikten
- Generierung von „leesbaren“ PVS Variablen