Technische Universität
München

Fakultät für
Informatik

fortiss An-Institut an der Technischen
Universität München

Support Slides

Adding C++ Support to mbeddr

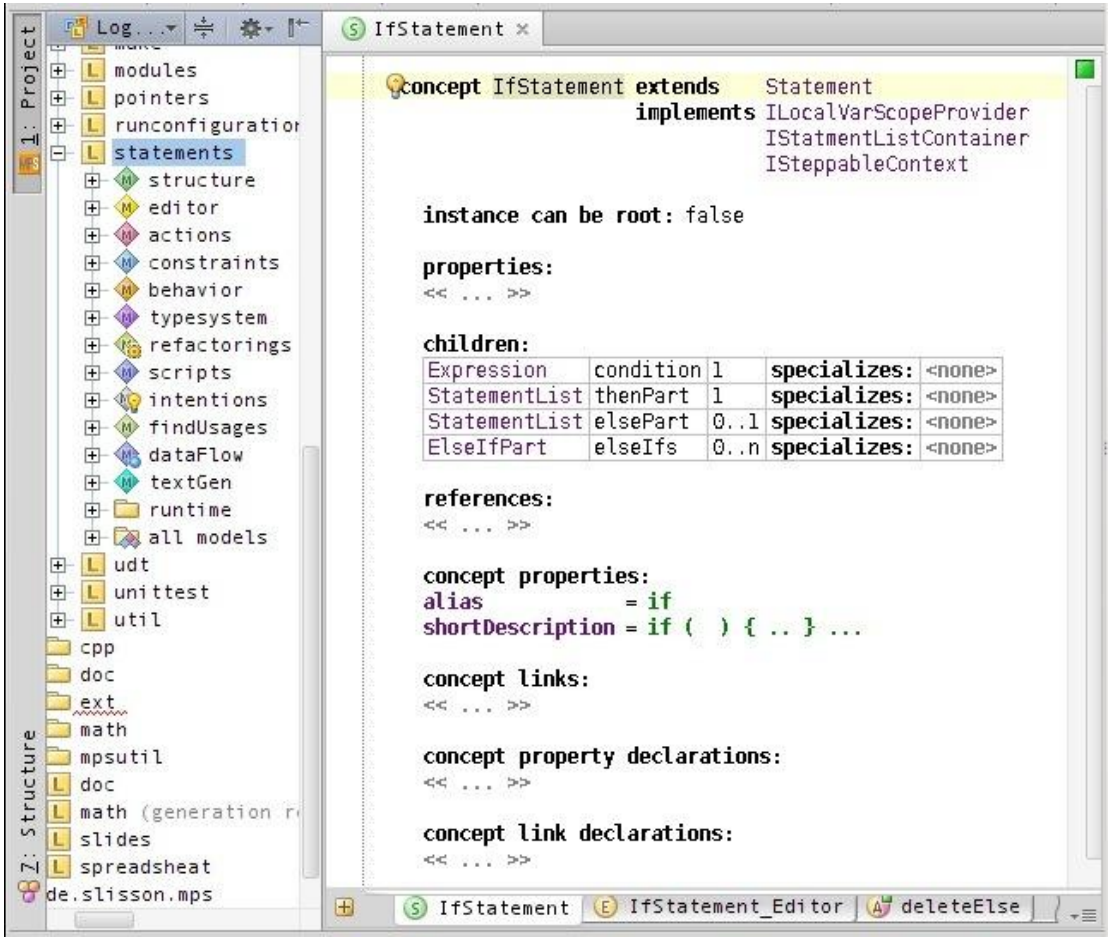
Language Engineering for C++ over the mbeddr Project C implementation

Presents: **Zaur Molotnikov**
Advisor: Dr. rer. nat. Daniel Ratiu
Supervisor: PD Dr. rer. nat. habil. Bernhard Schätz

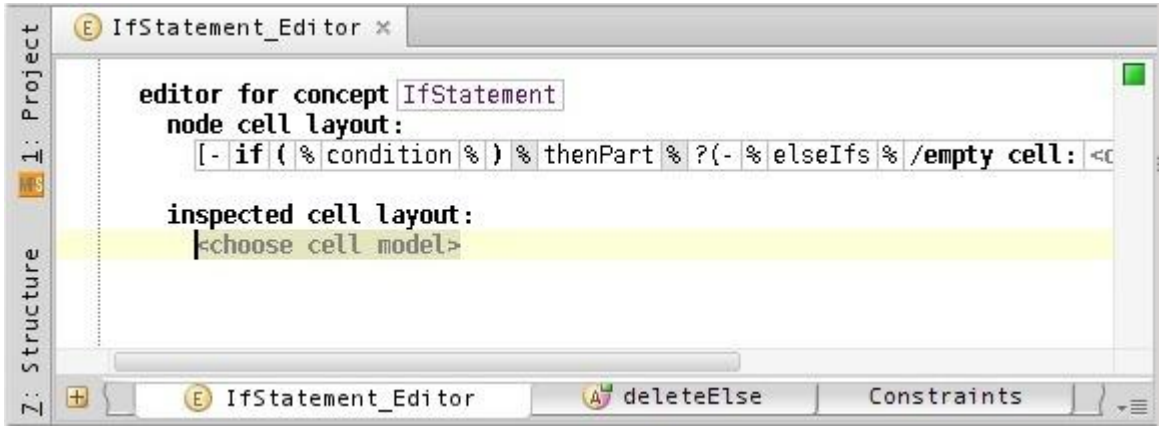
Presentation Structure

- Introduction
  - JetBrains MPS
- Projectional C++
- Future Work

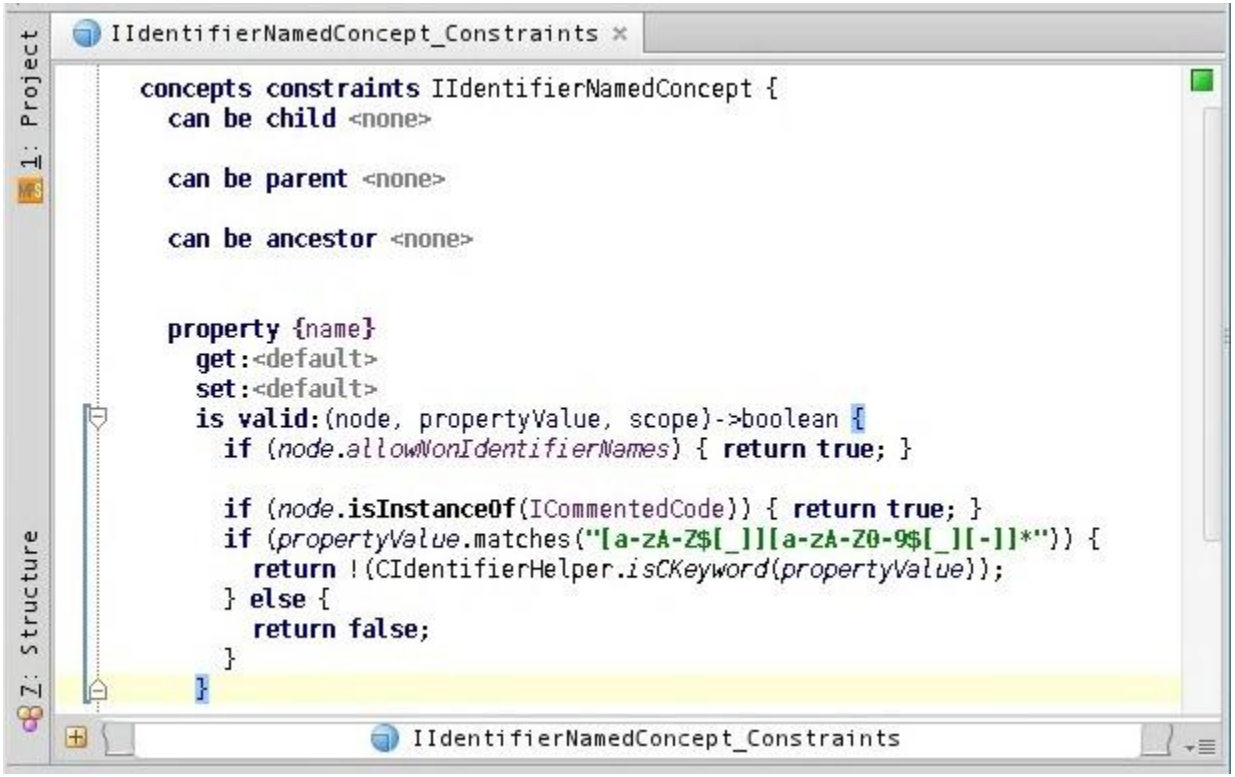
Structure View - MPS



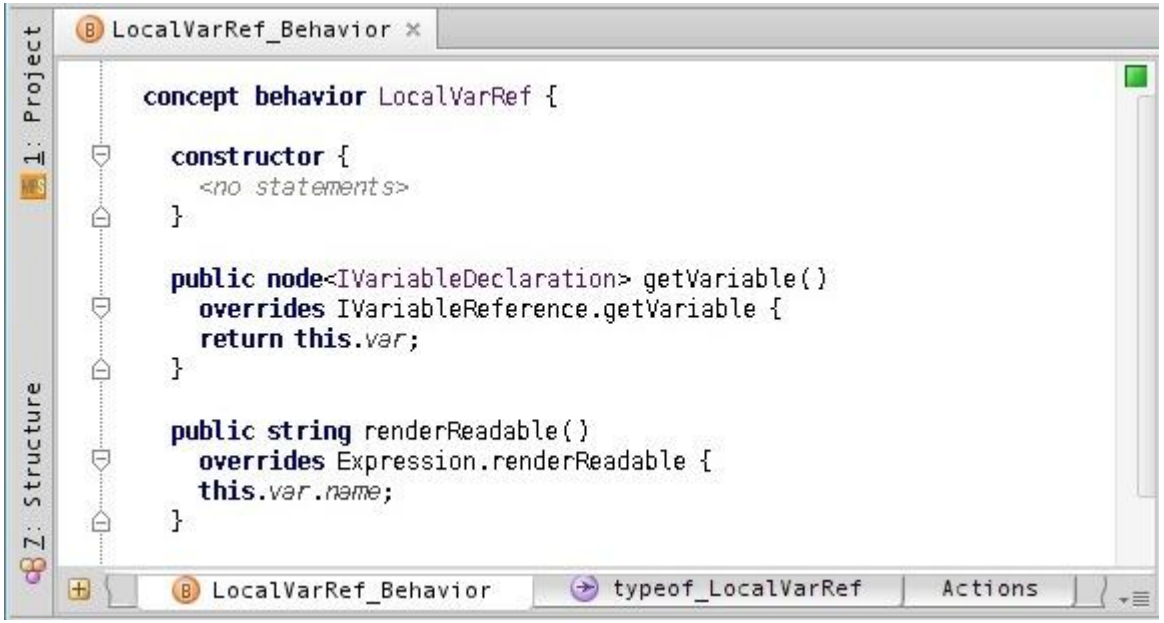
Editor View - MPS



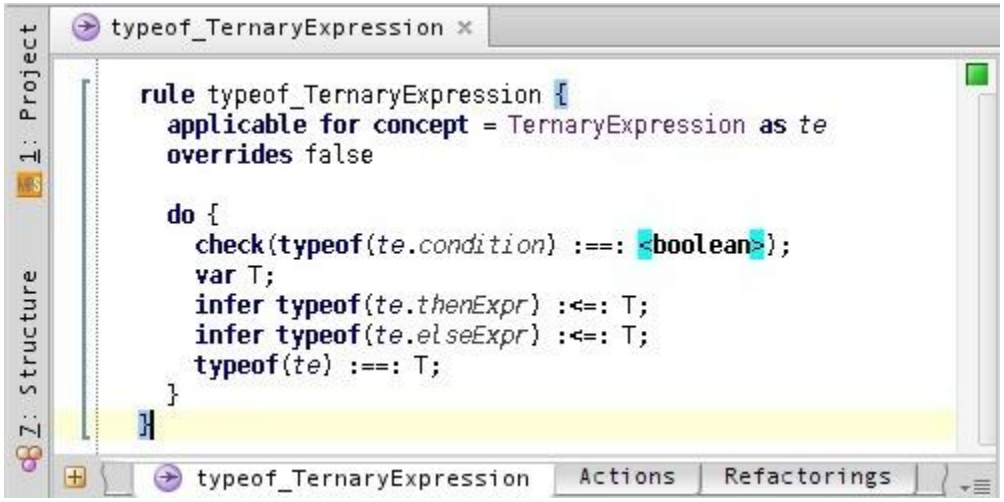
Constraints View - MPS



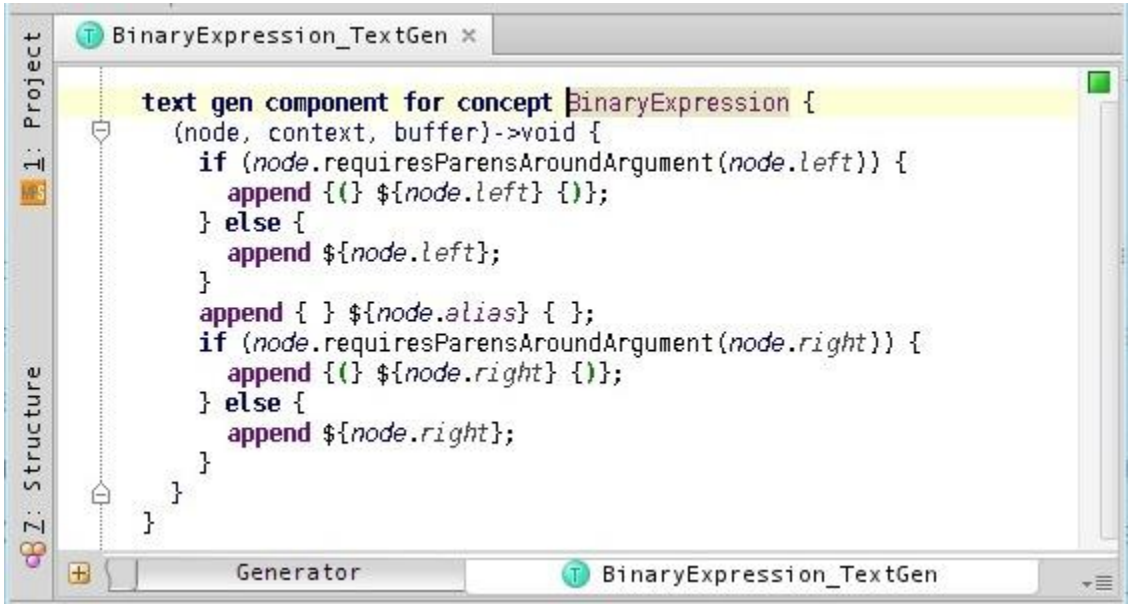
Behavior View - MPS



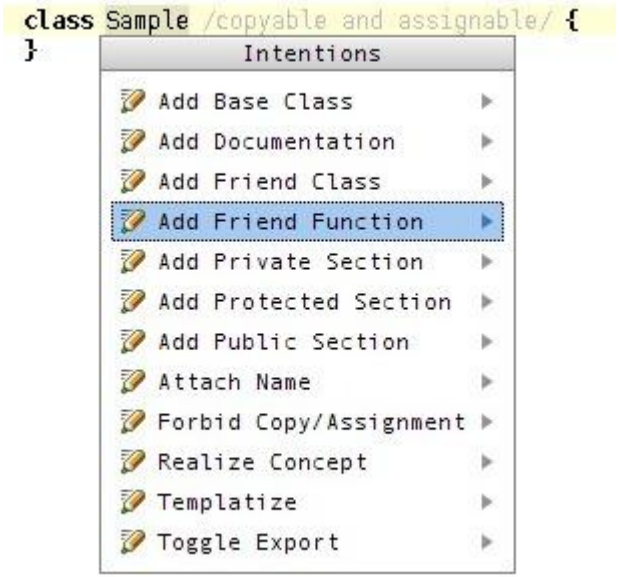
Type System View - MPS



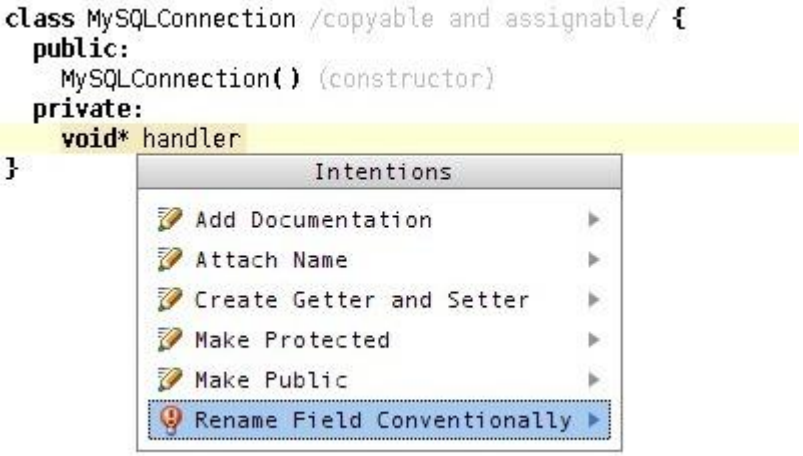
TextGen View - MPS



Intentions Example - MPS



Non-Type-System Checks Example - MPS



Presentation Structure

- Introduction
  - JetBrains MPS
- Projectional C++
- Future Work

Presentation Structure

- Introduction
- Projectional C++
  - Points of Interest
  - One-Side-Awareness
  - C and C++
  - Object-Oriented Programming
  - Operator Overloading
  - Templates
  - Advanced Functionality
- Future Work

One-Side-Awareness

- Extend mbeddr so, that only Projectional C++ is aware of mbeddr, and not vice versa
  - to not to make mbeddr code base bigger
  - to make support of mbeddr easier
  - to give an example of “pure” extension

- It is not as “simple” as in usual development in programming languages because
  - it is not just an interface and a usage of it since
  - all views on a language have to be extended and
  - all the new code has to reside in Projectional C++

C and C++

Differences:

- Reference type and boolean type in C++
  - result from extending expressions language
- Modules
  - no modules in C
  - modules in mbeddr
  - namespaces and classes in C++
  - Projectional C++ solution: modules and namespaces
- Memory allocation
  - operators **new** and **delete**
  - extensions for expressions and statement languages

C and C++

- Major differences between C and C++ were listed before.
- Otherwise similarities stay, C being a subset of C++:
  - Expressions
  - Types
  - Statements
  - Functions
  - ...

Object-Oriented Programming

- Object-Oriented Programming in C++ is supported through classes:
  - Classes declaration and copying
  - Encapsulation and access control
  - Polymorphism:
    - Polymorphic casting
    - Abstract classes and virtual functions

Classes Declaration and Copying

```
class A /copyable and assignable/ {
  public:
    explicit A() (constructor)
    A& operator = (const A& originial ) (makes class assignable)
    A(const A& originial) (copy constructor)
    int16 getX()
  private:
    int16 x
}
```

- Safe declaration of constructors as explicit
- Copyability and assignability controlled
- Getters and Setters (safe) generated

Encapsulation and Access Control

```
class A /copyable and assignable/ {
  public:
    int8 valAPublic
  private:
    int8 valAPrivate
  protected:
    int8 valAProtected
  friends:
    friend compare (boolean compare(const A& a1, const A& a2))
}

class B : public A /copyable and assignable/ {
  public:
    B(const B& originial) (copy constructor)
}
```

```
B::B(const B& originial)  from B {
  this->valAPublic = originial.valAPublic;
  this->valAProtected = originial.valAProtected;
  this->valAPrivate;
}

boolean compare(const A& a1, const A& a2) {
  return a1.valAPrivate >= a2.valAPrivate;
} compare (function)

void printOut(B b) {
  cout <<  b.valAPublic;
  cout <<  b.valAPrivate;
  cout <<  b.valAProtected;
} printOut (function)
```

- Encapsulation supported fully in the editor
  - Class sections, inheritance, friends
- Friends are declared in special section
- Friends are more clear to see

Polymorphism

Here - polymorphism through virtual functions and inheritance.

In C++ polymorphism can be also achieved through template programming and operator overloading.

Polymorphic casting

```
class NonPoly /copyable and assignable/ {
  public:
    void hello()
    int32 getFive()
    NonPoly() (constructor)
}

class NPChild : public NonPoly /copyable and assignable/ {
  public:
    NPChild() (constructor)
}

testcase NonPolymorphicCasting {

  NonPoly* parent = new NonPoly();

  ( parent as NPChild* )->hello();

  assert(0) ( parent as NonPoly* )->getFive() == 5;

} NonPolymorphicCasting(test case)
```

- New cast operation **as** checks if the cast is meaningful

Abstract Classes and Virtual Functions

```
abstract class Widget /copyable and assignable/ {
  public:
    explicit Widget(Widget* parent) (constructor)
    pure virtual Size getDimensions()  = 0
}

abstract class Button : public Widget /copyable and assignable/ {
  public:
    Button() (constructor)
    pure virtual boolean isPressed()  = 0
}

class PushButton : public Button /copyable and assignable/ {
  public:
    PushButton() (constructor)
    virtual Size getDimensions() overrides Widget::getDimensions()
    virtual boolean isPressed() overrides Button::isPressed()
}
```

- Explicit syntax added for
  - Pure virtual functions, abstract classes, overrides

Operator Overloading

```
class Coords /copyable and assignable/ {
  public:
    Coords() (constructor)
    Coords(int32 xx, int32 yy) (constructor)
    Coords operator + (Coords arg )
    Coords operator - (Coords arg )
    int32 operator [] (int32 index )
    int32 getX()
    int32 getY()
  private:
    int32 mX
    int32 mY
}
```

```
Coords v1 = Coords(1, 2);
Coords v2 = Coords(2, 3);
Coords v3 = v1 + v2;

assert(0) v3.getX() == 3;
assert(1) v3.getY() == 5;

Coords v4 = v2 - v1;

assert(2) v4.getX() == 1;
assert(3) v4.getY() == 1;
assert(4) v4[1] == 1;
```

- One-side-awareness and reuse
- Operator overloading as language engineering in C++

Templates

- Implemented through “C++ concepts”
- Have a number of advantages and disadvantages
  - explicit
  - checkable

  but
  - absent in C++
  - special importer needed
  - additional user work
  - code duplication

```
concept Comparable {
  public:
    int8 compare(Comparable c1)
}

realizes Comparable
class NumberWrapper /copyable and assignable/ {
  public:
    int8 compare(NumberWrapper other)
    NumberWrapper(int8 v) (constructor)
  private:
    int8 mValue
}

template <class T: Comparable>
class OrderedList /copyable and assignable/ {
  public:
    OrderedList() (constructor)
    int8 compare(T first, T other)
}
```

Some Other Language Features

- Exceptions
- Standard output stub

- STL will require all features of the language

Advanced Functionality

Some additional features are present in Projectional C++ editor:

- Primitive renamings - due to projection
- Getter and setter generation
- Naming conventions
- Method implemented check
- Abstract class construction check
- Array deallocation check

More checks can be added (class virtuality, size, exceptions, data flow…)

Complex Analyses : Qt

- Example code base - Qt Library - approximately 1000 classes, 1 parent each, 3 member fields, 3 methods, each with 1 parameter, 1 return type, 1 variable.

- Naming Conventions with Regular Expressions
  - 9 000 nodes to check, if they comply to a RegEx
- Method Implemented Check
  - 12 000 nodes to check
- Abstract Class Instantiation Check
  - 100 000 nodes to check

- Accessing nodes in MPS is slow
- Analyses can run in parallel

Presentation Structure

- Introduction
- Projectional C++
  - Points of Interest
  - One-Side-Awareness
  - C and C++
  - Object-Oriented Programming
  - Operator Overloading
  - Templates
  - Advanced Functionality
- Future Work

Presentation Structure

- Introduction
- Projectional C++
- Future Work
  - Potential Extensions

Potential Extensions

- Language constructions as emulated by preprocessor and templates (Alexandrescu)
- Signals and Slots (Qt, Objective-C)
- Object Oriented Design Patterns
- Higher level models with semantics, implemented by classes
- More? - Question by itself