The Case for Simple Object-Orientation in VDM++

Erik Ernst
Aarhus University, Denmark

“An outsider..”

- My research area is OO language design, implementation, formalization, esp. type systems (virtual classes, fam.pol.)
- Worked in Coq, lectured on contracts
- No background specifically on VDM*
- Will offer some loud opinions ;-)

Overview

- Premises
- Language design rationale
- Features to include
- Features to omit
- Reductions (syntactic sugar)
- Summary

Overview

- Premises
- Language design rationale
- Features to include
- Features to omit
- Reductions (syntactic sugar)
- Summary

Premises

- VDM++ should remain compatible with the relevant VDM dialects (VDM-SL)
- VDM++ should enable precise reasoning about software, including tool support
- VDM++ should embrace multiple target languages
- The VDM 'mindset' is practical

From premises..

- The VDM++ language should have a well-defined semantics (for precision)
- The semantics should be simple (for reasoning and tool support)
- The semantics should build on core OO concepts, only (for coverage)

Semantic conflicts

- It is not possible to "take the union" of the semantics of all languages, e.g.:
- Java: two m-interfaces have shared impl.
- C#: may implement m per interface
- Java cannot support distinct behavior, C# cannot promise distinct declarations
- Conflict!

Overview

- Premises
- Language design rationale
- Features to include
- Features to omit
- Reductions (syntactic sugar)
- Summary

Language Design Rationale

- Go for a precise semantics based on a few, powerful core OO abstractions
- Familiar features may unfold to more verbose forms; name clashes etc. may be avoided because specification is an early activity

Overview

- Premises
- Language design rationale
- Features to include
- Features to omit
- Reductions (syntactic sugar)
- Summary

Features to include

- Static typing
- Object; method; class
- Mutable state; dispatch
- Inheritance; subtyping
- Assertions (invariants, pre-/post-cond...)

A word on inheritance

- A plethora of variants exist
- A simple core: Single inheritance
- Multiple inh. may be needed for coverage
- Really difficult — but could try to do
  - specification of superclass relations
  - explicit resolution of clashes, repeating...

Overview

- Premises
- Language design rationale
- Features to include
- Features to omit
- Reductions (syntactic sugar)
- Summary

Features to omit

- Static features
- Constructors
- Declarative initialization
- Static overloading
- Access control
- Nesting

Omitted: Static features

- Static state is just global variables with twisted names; so let's have globals explicitly, or leave it out
- (Isn't it "class object state"? No: try 2*getClass(), then newInstance())
- Static methods are just global procedures, similar treatment

Omitted: Constructors

- An anomaly — not inherited, implicit super chain calls, does not return the object
- It's just wrong to execute code in a context that does not exist! ;-)
- Primitive allocation establishes invariants from scratch by complete state arguments
- Factory procedures provide abstraction

Omitted: Declarative initialization

- Declarative = Good?
- Requires too much magic for initialization, in multiple un-ordered modules!
- Object strategy (start with complete state) not realistic
- Use explicit, multi-step initialization
- E.g., per module when dependencies acyclic

Omitted: Static overloading

- Static overloading rules are horribly complex. (Ask Gilad Bracha...)
- Incompatibilities in the details unavoidable
- Trivial to remove by using more explicit naming

Omitted: Access control

- Many different incompatible models
- Seems simple, but even Java has quirks (nesting, packages, protected)
- Should be separable: It only rejects some programs, never changes the semantics
- Can be added orthogonally

Omitted: Nesting

- Standard semantics does not depend crucially on block structure (nesting)
- Example: The javac compiler flattens the class space, adds immutable references to enclosing objects
- With virtual classes etc, this would be a crucial element, but not for mainstream languages

Overview

- Premises
- Language design rationale
- Features to include
- Features to omit
- Reductions (syntactic sugar)
- Summary

Reductions

- Idea: The core language is the output from a transformation process
- Several surface languages may add various features (e.g., constructors) to the core
- Specification writing is pragmatic
- Proof obligations in the core language?

Reduction issue

- Code generation may need to be based on surface language (for performance)
- Need justification for a surface language semantics
- Must be shown by (1) core language semantics, (2) transformation semantics preservation

Overview

- Premises
- Language design rationale
- Features to include
- Features to omit
- Reductions (syntactic sugar)
- Summary

The Case for Simple Object-Orientation in VDM++

- Premises/rationale: simple core + sugar
- Included: object, method, class, inheritance
- Omitted: static features, ctrs, declarative initialization, overloading, access ctl, nesting
- Priority: reasoning about programs!