Object Oriented Issues in VDM++

Nick Battle, Fujitsu UK

(nick.battle@uk.fujitsu.com)

Background

VDMJ implemented VDM-SL first (started late 2007)

- Formally defined. Very few semantic problems

VDM++ support added to VDMJ subsequently (started mid 2008)

- Informally defined. Several semantic ambiguities
- Problems and proposed solutions detailed in [1]
- This paper summarizes the original issues:
  - *Mainly common sense*
  - *Based on normal OO programming*
  - *Try to make formal analysis simpler where possible*

VDM-SL and VDM++ Compared

VDM-SL has a simple “flat” structure

- State is local to a module
- State initializers are *atomic* record expressions
- Import/export of everything *except* state

VDM++ has a hierarchical class structure

- Public static state can be accessed directly from anywhere
- Object (instance) state as well as class (static) state
- State initializers are *separate* expressions
- Inheritance, overloading, overriding and polymorphism
- Threads (non-deterministic?)

Static Initialization

- Public static instance variables are roughly like state in VDM-SL
- A specification must initialize into a known state
- Similar to defining a single *spec_init()* operation
- Must be deterministic

***Proposal**: static variables are initialized by computing the dependency graph of static variables and constants in the specification, including references made via operations and constructors. The order of initialization of independent sub-graphs is not defined. A circular dependency is an error. Initialization cannot create threads, or use loose or non-deterministic statements.*

Static Initialization

- But initializers can update state, so ordering is important even within the graph of dependencies
- There is no convenient ordering of variable definitions in arbitrary classes (files) in a specification
- VDMJ and VDMTools initialize the following differently:

```
class A
instance variables
   static varA:nat := Z`nextValue();
   ...

class B
instance variables
   static varB:nat := Z`nextValue();
   ...
```

Instance Initialization

- Similar issue when initializing instance variables in a new object
- But declaration order can be used
- Relax the deterministic requirement?

***Proposal**: instance variables are evaluated using a dependency graph of the variables they depend on, including references made via operations and constructors. Independent sub-graphs can be initialized in any order, and circular dependencies are an error. Variables which have common dependencies are initialized in declaration order.*

Multiple Superclass Construction

- Superclass constructors are called explicitly, in any order
- Difficult to call multiple explicit constructors without “hiding” the return value:

```
class Z is subclass of A, B
operations
   public Z: int * int ==> Z
   Z(a, b) ==
      let - = B(b), - = A(a) in skip;

end Z
```

Multiple Superclass Construction

***Proposal**: VDM++ should have an explicit superclass construction syntax, similar to C++ and C#. Superclasses are constructed before subclasses. It would then be an error to try to call a constructor operation explicitly.*

```
class Z is subclass of A, B
operations
   public Z: int * int  -- NB. return type implicit
   Z(a, b): A(a), B(b) ==
      skip;

end Z
```

Multiple Superclass Construction

- In what order are multiple superclasses constructed?
- When are instance variables initialized in a hierarchy?

***Proposal**: VDM++ should make a depth first construction of superclasses, in the order of the classes in the "is subclass of" clause. Instance variable initializers are evaluated before the constructor at each object level.*

- When are default constructors used in a hierarchy?

***Proposal**: if no explicit constructor is called in the proposed new syntax, then a default constructor is called at the appropriate position in the initialization sequence. If no default constructor is provided by the class, a blank one is provided which just initializes the instance variables.*

Construction and Inheritance

- If a constructor calls an operation which is overridden, does it call the local version (C++) or the overridden one (Java)?
- Similarly with variable initializers which call overridden operations

***Proposal**: both constructors and instance variable initializers should call the local version of operations and functions. This means a constructor behaves the same way stand alone and in a hierarchy.*

Construction and Inheritance

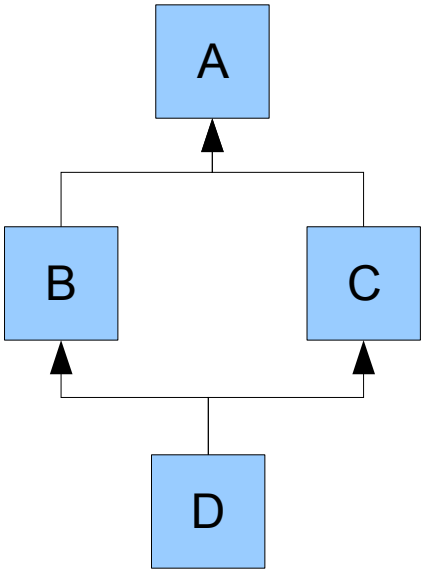
- Are constructors inheritable?

***Proposal**: VDM++ should not allow constructor inheritance. Most OO languages do not permit this, and inheritance would not fit with constructors calling local versions of overridden operations*

- When are class invariants enforced during construction?

***Proposal**: invariants are only enforced once the construction of an object is complete. This applies recursively to invariants on superclasses in a hierarchy. On completion of construction, the invariant is evaluated once, and on state changes thereafter.*

Diamond Inheritance



- Who constructs A?
- Is there one copy of A in D or two?

Proposal: *In diamond inheritance, subclasses have their own private copies of common superclasses, which are constructed separately. Hence, any formal analysis for subclasses in isolation is still applicable when the classes are composed in a diamond hierarchy.*

Behavioural Subtyping

- When do members with co/contravariant parameter and return types override superclass members?

***Proposal**: VDM++ should implement strong behavioural subtyping in class hierarchy overriding. This will simplify the formal analysis of specifications.*

- What does strong behavioural subtyping mean for pre- and postconditions?

***Proposal**: An overriding member's preconditions can be no stronger than the overridden member's preconditions. An overriding member's postconditions must be stronger than the overridden member's postconditions. An overriding member must have a covariant return type and contravariant parameter types (else it is an overload).*

Polymorphism and Currying

- How do polymorphic functions behave regarding overriding and overloading?

Proposal: *A polymorphic function can be overridden by a function of the same name, with the same shape of parameter types as the superclass (for example, "seq of @T" and "set of @T" are not the same shape). Polymorphic functions with the same name can overload each other if they have parameters that are a different shape.*

- How do curried functions behave regarding overriding and overloading?

Proposal: *A curried function can be overridden by another function with the same type signature (ie. including all the function type returns). Polymorphic functions, curried functions and simple functions can all overload each other as long as they are distinguishable by parameter types.*

Miscellaneous

- Can static functions/operations be called via an object reference, and do they behave polymorphically or bind statically?
- Note that all functions in VDM-10 are static

***Proposal**: static functions and operations can be called via an object reference, and do act polymorphically.*

- Note for code generation: neither C++ nor Java do this!

Operation pre- and postconditions

- What are the prototypes and semantics of operations' pre- and postcondition functions in VDM++?
- VDM-SL is simple, using local *Sigma* structures passed to pre_op and post_op
- VDM++ operations can potentially read/write the entire state of the specification

***Proposal**: VDM++ pre- and postcondition functions are similar to those in VDM-SL, except with "self" parameters rather than Sigma parameters. The self objects represent a deep copy of the state. Operations which reference static variables do not have pre- and postcondition functions.*

Further Issues

- Is VDM++ thread scheduling deterministic? If so, what is the scheduling policy?
- How can we do code generation if VDM++ semantics does not match the target language's “natural” OO semantics?
- What does *#act(op)* etc. mean if op is overloaded?
- What are the binding rules in the light of overloading, overriding, and a type checker with *possible* semantics?
- Many other subtleties...