Validated Multi-Layer Meta-modeling via Intrinsically Modeled Operations

a model-defined modular constraint framework for multi-level meta-modeling

Dániel Urbán, Gergely Mezei, ***Zoltán Theisz***

Budapest University of Technology and Economics
Department of Automation and Applied Informatics

Outline

- DMLA background
- DMLAScript explained
- Bootstrap for Validation
  - Core entities
  - Core mechanisms (alpha & beta formulae)
- Operations in DMLAScript
- Validation mechanism
- Person example – validation in practice
- Conclusion

DMLA in a Nutshell

- Model = Labelled Directed Graph (LDG)
  - Nodes, edges have labels
  - Attribute = virtual nodes ➔ have labels
  - {Nodes, edges, attributes} – entities of DMLA
  - Dual-field notation (Node$_{ID}$, Node$_{meta}$…)
- Based on Abstract State Machines (ASM)

Tuple Syntax Notation

- Superuniverse
  $\{U_{Bool}, U_{Number}, U_{String}, U_{ID}$ and $U_{basic}\}$ + undef
  - Label values are from universes
- Labels → tuples
  - DMLA 1.0 : 6- tuple
    (ID, Name, Meta, Value, Cardinality, Children)
  - DMLA 2.x : 4 – tuple

    $X = \{X_{ID}, X_{Meta}, X_{values,} X_{Attributes}\}$
    $X_{ID} : U_{ID}, X_{Meta} : U_{ID}, X_{Values} : U_{Basic}[], X_{Attributes} : U_{ID}[]$

ASM Semantics

- Statemachine
  - State = model snapshot
  - Transition = changes in the model
- Functions
  - Shared functions – snapshot evolution
    - Modification internal (by algebra) or external (by user)
    - Label manipulation, entity creation/deletion (Meta(U_{ID}), …)
  - Derived functions – obtaining information
    - Shortcuts for utility functions
    - Contains(U_{ID}, U_{ID}), DeriveFrom(U_{ID}, U_{ID})

DMLAScript Explained

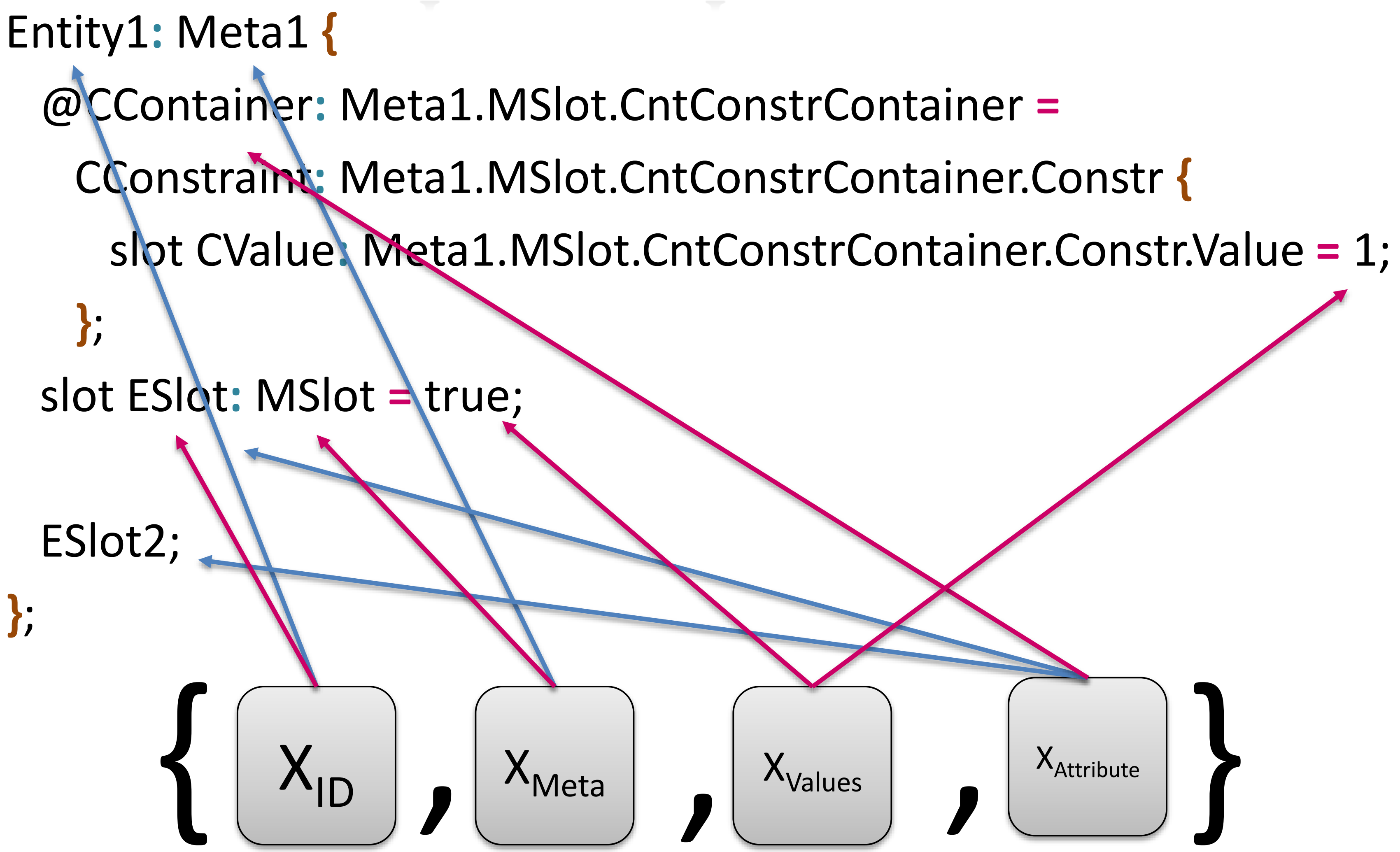
- Tuple = abstract syntax
  - Only for storage/execution
  - Concrete syntax – XML / DSL → DMLAScript
- DMLAScript – DMLA DSL in Xtext
  - Quick & comprehensible
  - Creation of context illusion for related tuples
  - Integrated scripting syntax for operations

DMLAScript – Example

```
Entity1: Meta1 {
  @CContainer: Meta1.MSlot.CntConstrContainer =
    CConstraint: Meta1.MSlot.CntConstrContainer.Constr {
      slot CValue: Meta1.MSlot.CntConstrContainer.Constr.Value = 1;
    };
  slot ESlot: MSlot = true;

  ESlot2;
};
```

DMLAScript – Tuple Positions



```
Entity1: Meta1 {
  @CContainer: Meta1.MSlot.CntConstrContainer =
    CConstraint: Meta1.MSlot.CntConstrContainer.Constr {
      slot CValue: Meta1.MSlot.CntConstrContainer.Constr.Value = 1;
    };
  slot ESlot: MSlot = true;

  ESlot2;
};
```

Tuple Production - DMLAScript

```
Entity1: Meta1 {
  @CContainer: Meta1.MSlot.CntConstrContainer =
    CConstraint: Meta1.MSlot.CntConstrContainer.Constr {
      slot CValue: Meta1.MSlot.CntConstrContainer.Constr.Value = 1;
    };
  slot ESlot: MSlot = true;

  ESlot2;
};
```

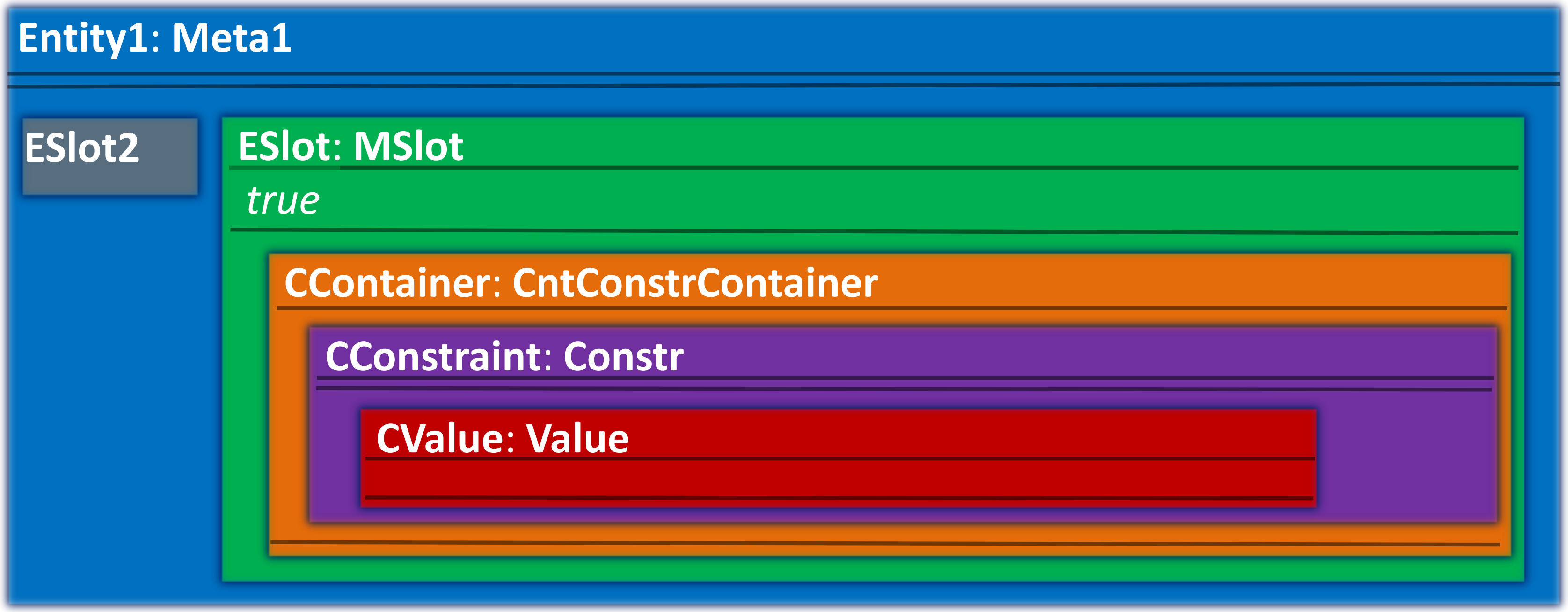
Tuple Production – 4-Tuples

- {Entity1, Meta1, undef, [Entity1.ESlot, ESlot2]}
- {ESlot2, …}
- {Entity1.ESlot, Meta1.MSlot, [true], [Entity1.ESlot.CContainer]}
- {Entity1.ESlot.CContainer, Meta1.MSlot.CntConstrContainer, [Entity1.ESlot.CContainer.CConstraint], undef}
- {Entity1.ESlot.CContainer.CConstraint, Meta1.MSlot.CntConstrContainer.Constr, undef, [Entity1.ESlot.CContainer.Cconstraint.CValue]}
- {Entity1.ESlot.CContainer.Cconstraint.CValue, Meta1.MSlot.CntConstrContainer.Constr.Value, [1], undef}

```
Entity1: Meta1 {
  @CContainer: Meta1.MSlot.CntConstrContainer =
    CConstraint: Meta1.MSlot.CntConstrContainer.Constr {
      slot CValue: Meta1.MSlot.CntConstrContainer.Constr.Value = 1;
    };
  slot ESlot: MSlot = true;
  ESlot2; };
```

DMLAScript – Context Illusion

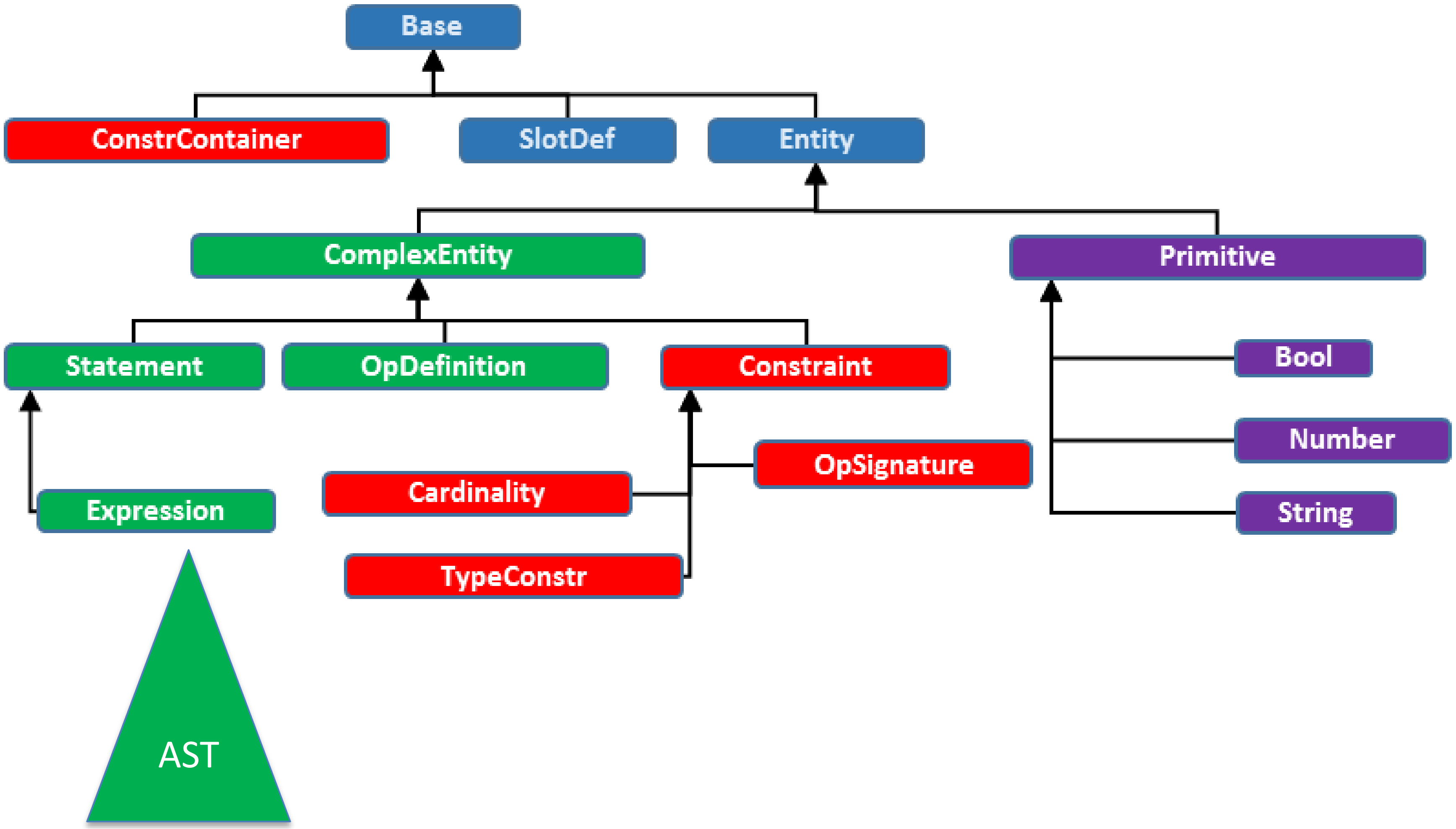
- {Entity1, Meta1, undef, [ESlot, ESlot2]}
- {ESlot2, …}
- {ESlot, MSlot, [true], [CContainer]}
- {CContainer, CntConstrContainer, [CConstraint], undef}
- {CConstraint, Constr, undef, [CValue]}
- {CValue, Value, [1], undef}



The Bootstrap

- Enabler of multi-level meta-modelling
  - Instantiation
    - Clone
    - Instance (some concretization)
  - Multi-purpose meta hierarchy
    - Core entities – Basic structure of modelling
    - Primitives – Concrete, atomic values
    - Constraint handling entities – Validation framework
    - AST entities – Operation structure
    - Domain-specific entities
- Precise + adaptable meta-modelling
  - Fixed tuple representation & ASM functions
  - Exchangeable meta hierarchy

Meta Hierarchy



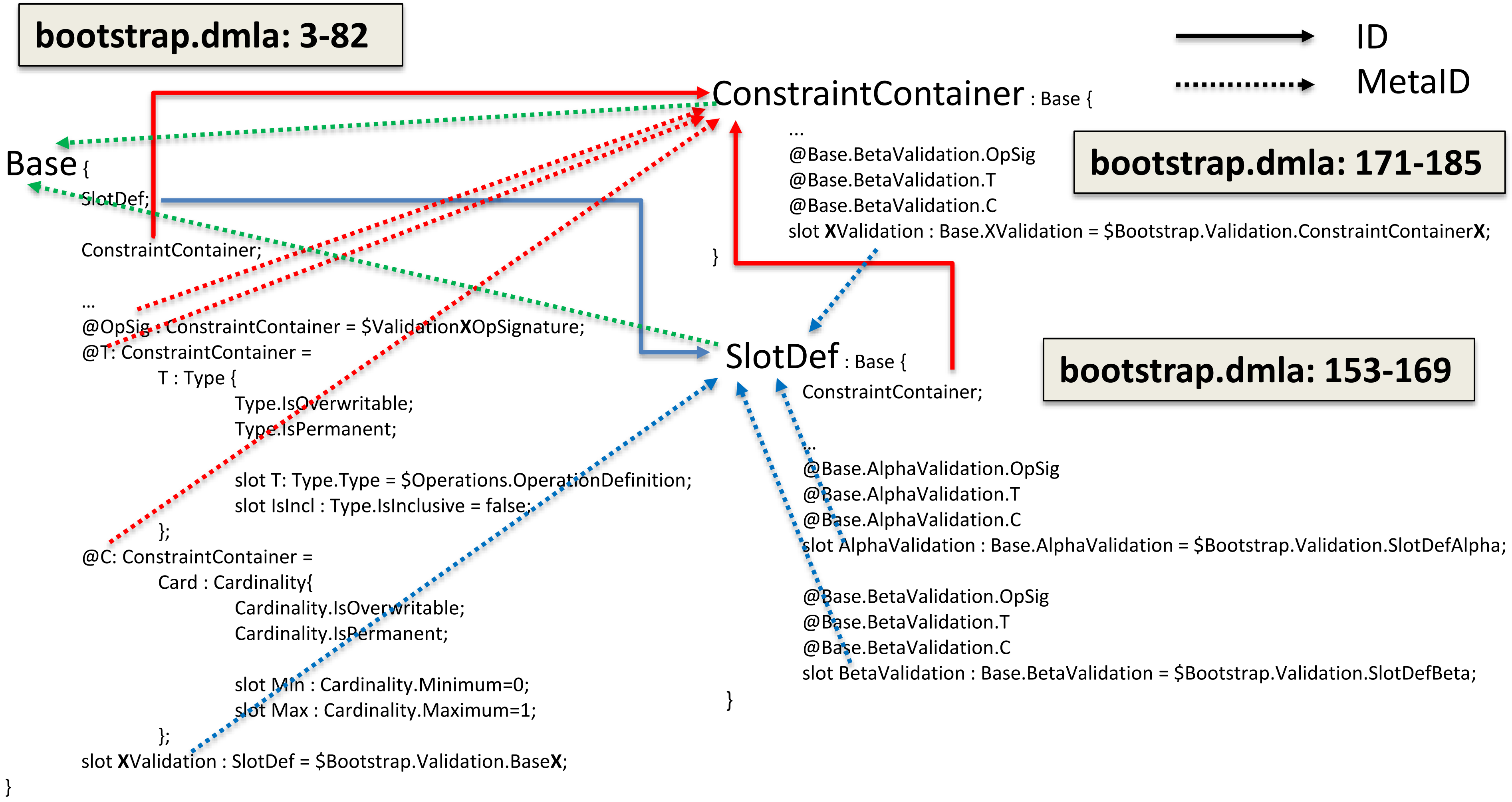
bootstrap.dmla: 3-82

Base – Root of meta-models

- Root of meta hierarchy → Every entity is an instance of Base
- Composite types: contains SlotDef (#4)
- Facilitator of validation framework
  - Constraints: contains ConstraintContainer (#6)
  - Slots for alpha and beta formula (#26-62)

    (Implemented in validation.dmla#212-227)

Enablers of Deep Meta-modelling



bootstrap.dmla: 3-82

```
Base {
    SlotDef;

    ConstraintContainer;

    ...
    @OpSig : ConstraintContainer = $ValidationXOpSignature;
    @T: ConstraintContainer =
            T : Type {
                    Type.IsOverwritable;
                    Type.IsPermanent;

                    slot T: Type.Type = $Operations.OperationDefinition;
                    slot IsIncl : Type.IsInclusive = false;
            };
    @C: ConstraintContainer =
            Card : Cardinality{
                    Cardinality.IsOverwritable;
                    Cardinality.IsPermanent;

                    slot Min : Cardinality.Minimum=0;
                    slot Max : Cardinality.Maximum=1;
            };
    slot XValidation : SlotDef = $Bootstrap.Validation.BaseX;
}
```

bootstrap.dmla: 171-185

```
ConstraintContainer : Base {
    ...
    @Base.BetaValidation.OpSig
    @Base.BetaValidation.T
    @Base.BetaValidation.C
    slot XValidation : Base.XValidation = $Bootstrap.Validation.ConstraintContainerX;
}
```

bootstrap.dmla: 153-169

```
SlotDef : Base {
    ConstraintContainer;

    ...
    @Base.AlphaValidation.OpSig
    @Base.AlphaValidation.T
    @Base.AlphaValidation.C
    slot AlphaValidation : Base.AlphaValidation = $Bootstrap.Validation.SlotDefAlpha;

    @Base.BetaValidation.OpSig
    @Base.BetaValidation.T
    @Base.BetaValidation.C
    slot BetaValidation : Base.BetaValidation = $Bootstrap.Validation.SlotDefBeta;
}
```

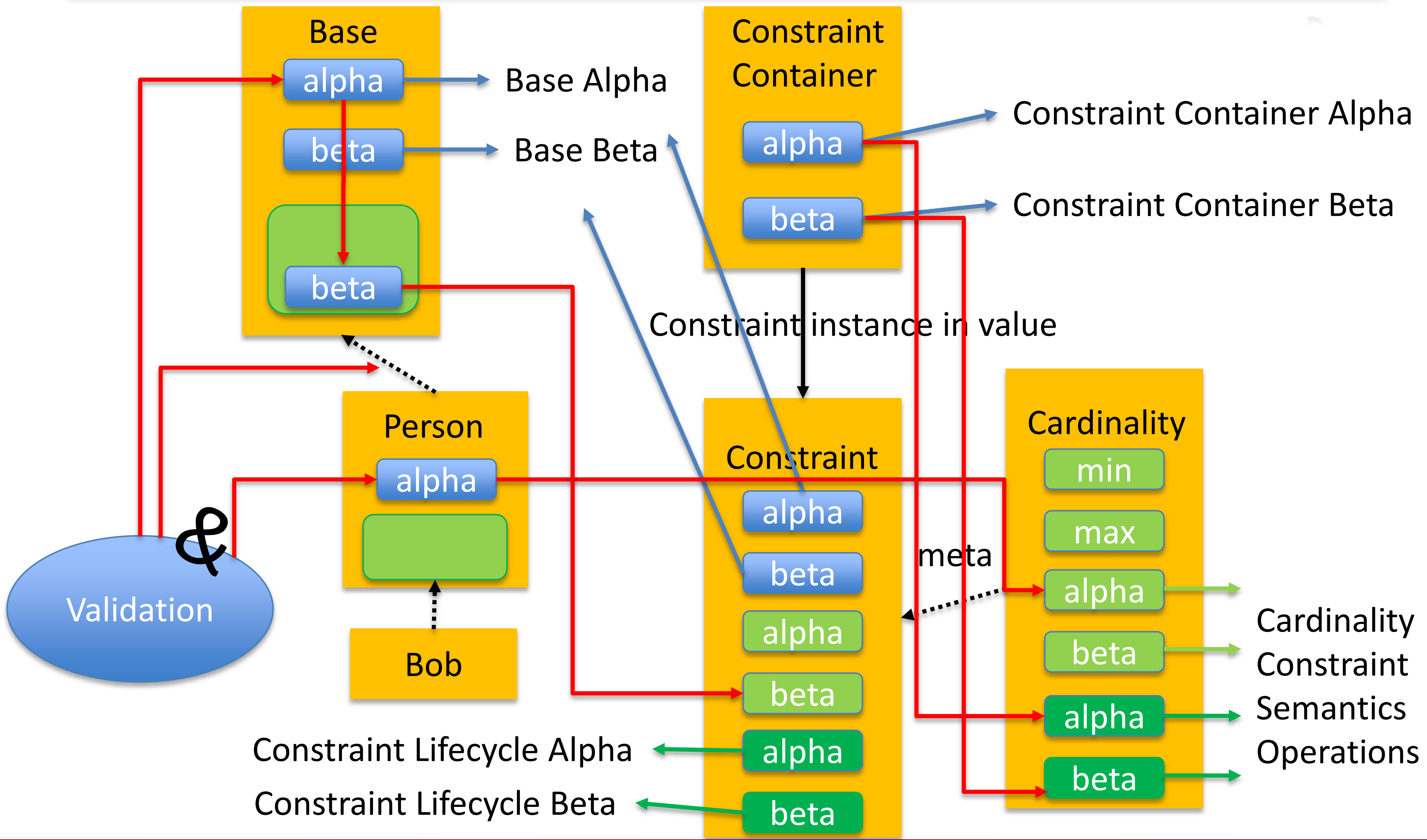
Validated Deep Meta-modelling

- Consistent multi-level meta-modelling
  - No self-reference in SlotDef
  - SlotDef inherits only SlotDef instances from Base as attributes
- Validation logic can be defined on every entity
  - Core: Alpha and beta formula slots in Base
  - Modular extenstion: ConstraintContainer in Base
- Modular validation logic – type, cardinality
  - ConstraintContainer instances with Constraint instance value (e.g. Cardinality entity instance)

Operations

- Fully modeled by Tuples and in Xtext syntax
  - Described by Abstract Syntax Tree (AST)
  - AST is built by DMLA entities in Bootstrap
  - Statements, Expressions (If, For, ...)
- OperationDefinition
  - OperationSignature constraint
  - Return type, context, params, body (statement)
- OperationCall

Validation Framework -Summary



Base
alpha
beta
beta
Base Alpha
Base Beta
Constraint Container
alpha
beta
Constraint Container Alpha
Constraint Container Beta
Constraint instance in value
Person
alpha
Bob
Constraint
alpha
beta
alpha
beta
alpha
beta
meta
Cardinality
min
max
alpha
beta
alpha
beta
Cardinality Constraint Semantics Operations
Validation
&
Constraint Lifecycle Alpha
Constraint Lifecycle Beta

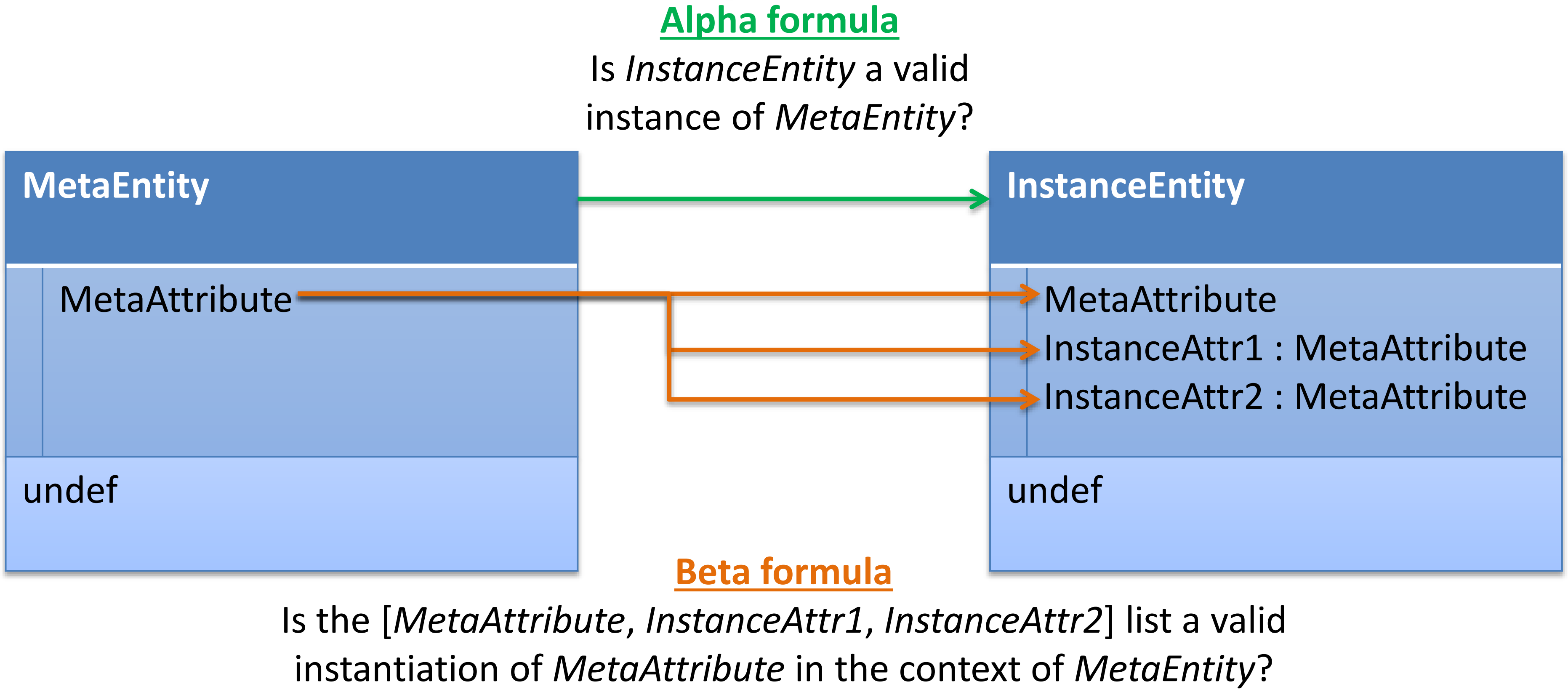
Alpha formula

- Validates an instance against meta (1:1)
- Valid instantiation
  - Cloning
  - Concretization
- bool ID::(ID inst)
  - „ID::” - *meta* entity
  - inst – entity to be validated
  - Returns true = valid DMLA instantiation (by structure)
- Default logic: BaseAlpha
  - calls: *beta formulae* of the *attributes* of *meta*
  - calls: *constraint alpha formula* of the *constraints* contained by *meta*

Beta formula

- Validates a meta entity against a list of entities (1:n)
- „In-context" check: the meta entity is contained by an entity E1 as an attribute, and the entities in the list are contained by an entity E2, where E2 is an instance of E1
- For cardinality-like constraints
- bool ID::(ID container, ID[] children)
  - „ID::" – *meta* entity
  - container – the entity containing *meta* as an attribute
  - children – list of entities to validate
  - Returns true = valid instantiation
- Default logic: BaseBeta
  - Calls: *constraint beta formulae* of the *constraints* in *meta*

Semantics of Alpha & Beta



Alpha formula

Is *InstanceEntity* a valid instance of *MetaEntity*?

Beta formula

Is the [*MetaAttribute*, *InstanceAttr1*, *InstanceAttr2*] list a valid instantiation of *MetaAttribute* in the context of *MetaEntity*?

Constraint Alpha & Beta

- Recurring logic in validation (e.g. type constraint on a slot)
- Modular „hooks” for formulae → Constraint
  - Containment-based
  - Empowered by multi-level modelling
- BaseAlpha/Beta
  1. find contained ConstraintContainers
  2. call ConstraintAlpha/Beta of constraints

Lifecycle Alpha & Beta

- Can the constraint be dropped/concretized during container instantiation?
- ConstraintContainer – manages constraint lifecycle - constraint-specific logic
- Hook in ConstraintContainer’s alpha/beta
  - Constraints specify their own life cycle logic
- ConstraintContainerAlpha/Beta → call contained Constraint’s LifeCycleAlpha/Beta

Validation

- Validation is a stand-alone operation
  - The only operation executed by ASM in JAVA
- Calls alpha formulas on all entities
  - Loops over all entities - instance
  - Gets meta by Meta(instance)
  - Fetches all alphas – hierarchy chain up to Base
  - Combines alphas (meta, instance) by AND
  - Combines entity validations by AND

Person Example

- Person with first names and last name
  - maximum two first names allowed
  - only one last name allowed
  - first names cannot be the same
  - Anonymous is not permitted ☺ - name is a must
- Examples:
  - OK: Bob Smith, Bob Rob Smith
  - Not OK: Bob Bob Smith

Person in DMLAScript - simplified

```
Person : ComplexEntity {

    @T: ComplexEntity.Children.T =
        Type : ComplexEntity.Children.T.T {
            slot Type : ComplexEntity.Children.T.T.T = $ComplexName;

        };
    @C: ComplexEntity.Children.C =
        Card : ComplexEntity.Children.C.Card {
            slot Min : ComplexEntity.Children.C.Card.Min = 1;
            slot Max : ComplexEntity.Children.C.Card.Max = 1;
        };

    slot FullName : ComplexEntity.Children;
    ...

    slot AlphaValidation : Base.AlphaValidation = $PersonAlpha;
}
```

ComplexName in DMLAScript - simplified

```
ComplexName : ComplexEntity{
     @T: ComplexEntity.Children.T =
          Type : ComplexEntity.Children.T.T {
               slot Type : ComplexEntity.Children.T.T.T = $String;
          };
     @C: ComplexEntity.Children.C =
          Card : ComplexEntity.Children.C.Card {
               slot Min : ComplexEntity.Children.C.Card.Min = 1;
               slot Max : ComplexEntity.Children.C.Card.Max = 2;
          };
     slot FirstName : ComplexEntity.Children;

     @T: ComplexEntity.Children.T =
          Type : ComplexEntity.Children.T.T {
               slot Type : ComplexEntity.Children.T.T.T = $String;
          };
     @C: ComplexEntity.Children.C =
          Card : ComplexEntity.Children.C.Card {
               slot Min : ComplexEntity.Children.C.Card.Min = 1;
               slot Max : ComplexEntity.Children.C.Card.Max = 1;
          };
     slot LastName : ComplexEntity.Children;
}
```

Alpha formula in Person

```
operation Bool ID::PersonAlpha(ID instance){

    ID fullName = call $GetRelevantAttributeValue(instance,
        $Person.FullName);

    if(fullName==null)
        return true;

    Object[] firstNames = call $GetRelevantAttributeValues(fullName,
    $ComplexName.FirstName);

    if(firstNames==null || size(firstNames)<2)
        return true;

    return index<Object>( firstNames, 0) != index<Object>(firstNames, 1);
}
```

Person Instances

```
BobSmith : Person{
        slot FullName : Person.FullName =
                BobSmithFullName : ComplexName {
                        slot FirstName : ComplexName.FirstName = "Bob";
                        slot LastName : ComplexName.LastName = "Smith";
                };
}
```

```
BobRobSmith : Person{
        slot FullName : Person.FullName =
                BobSmithFullName : ComplexName {
                        slot FirstName : ComplexName.FirstName = ["Bob", "Rob"]
                        slot LastName : ComplexName.LastName = "Smith";
                };
}
```

```
BobBobSmith : Person{
        slot FullName : Person.FullName =
                BobSmithFullName : ComplexName {
                        slot FirstName : ComplexName.FirstName = ["Bob", “Bob"]
                slot LastName : ComplexName.LastName = "Smith";
                };
}
```

Conclusions

- Dynamic Multi-Layer Algebra
  - Formal
  - Self-describing*
  - Self-validating
  - Flexible
  - Modular
  - New interpretation of layers
- Future directions
  - Case studies, pratical applications
  - More bootstraps (e.g. for MOF, potency notion, ...)
  - Java engine – DMLA ASM – DMLA VM
  - FSM-based lifecycle

The bicycle case study

- Bicycle example, 2 shortcomings:
  - No model level validation (e.g. max number of instances of an entity in the whole model) – gamma formula (full model validation per entity)
  - Cannot differentiate entity instantiation degree – overridable operation in Base, entities can calculate their own degree, where degree has predefined categories
    - meta (abstract, unfulfilled constraints)
    - intermediate (can be further instantiated, but meets all constraint criteria)
    - fully instantiated (cannot be instantiated any further)

Thank You & Any Questions?

Dynamic Multi-Layer Algebra

http://www.aut.bme.hu/Pages/Research/VMTS/DMLA