miUML Metamodel Class Descriptions
Type Subsystem

Leon Starr

Monday, August 1, 2011

mint.miUMLmeta.td.8

Version 2.2.3

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.

Copyright © 2002, 2009-2011 by

MODEL INTEGRATION, LLC

Table of Contents

Introduction

This metamodel captures the semantics of Executable UML as described in the book Executable UML: A Foundation for Model Driven Architectures. This document explains how the Class Model Metamodel supports these rules. Justification and explanation of Executable UML rules themselves are beyond the scope of this metamodel.

These descriptions are part of the Executable UML Class Model Metamodel Class Model. Some of the Classes in this model are well known entities defined in [Steve REF]. The beginning of each such description starts with the phrase "Defined in Executable UML". These descriptions do not attempt to duplicate those definitions. But for convenience, concise statements are included here. For further details, please read these books. All of the other classes were invented for the purposes of this metamodel. Full descriptions are included.

Key to Styles and Conventions

COLOR is used to convey information in this document. If you have the ability to print or display this document in color, it is highly recommended. Here are a few notes on the font, color and other styles used in class model descriptions.

Naming classes, types and subsystems

Modeled elements such as classes, types and subsystems are written with initial caps. The text Air Traffic Controller, for example, would probably be a class. If you see the text Application Domain, you know it is a modeled element, most likely a domain.

Naming attributes

Attributes are named with an initial cap followed by lower case, Time_logged_in, for example.

Instance values

Instance data, such as attribute values, are either in quote marks as in "Gwen" and "Owen" or represented in the following color/font: Gwen and Owen.

Whitespace

When programming, the squashedNamingStyle is optimal for easy typing and is adequately readable for short names. That's fine for programming, but analysis is all about easy readability and descriptive, sometimes verbose names. Programming style is retained for method names, but everywhere else whitespace or underscores are employed. For example: Air Traffic Controller, Time logged in or On_Duty_Controller.Time_logged_in.

hardcoreJavaProgrammersMayDisputeThisPointAndArgueThatWhiteSpacesAreEntirelyUnnecessaryForImprovingReadabilityButIdisagree.

Attribute heading color

Attribute headings are colored according to the role the attribute plays in its class. Referential attributes are brown, **Logged in controller**, for example. Naming attributes are blue, **Number**, for example. Finally, descriptive attributes are red, **Time logged in**, for example. Derived attributes are purple **\Volume**, for example.(One of the nice things about color is that it helps eliminate the need for underscores).

Referential renaming

A referential attribute will often have a name that reflects its role rather than the name of the base reference. Station.Logged_in_controller may refer to On_Duty_Controller.ID, for example. Such renaming will be defined in the referential attribute's description.

MDA (Model Driven Architecture) Layers

The MDA defined layers are referenced from time to time. It is sometimes useful, especially in a metamodel, to distinguish the various meta layer names: M0, M1, M2 and M3. These are M0: data values ('N3295Q', 27L, 490.7 Liters, ...) M1: domain specific model elements (class 'Aircraft', attribute 'Altitude', relationships 'R2 - is piloted by'), M2: metamodel elements to define miUML (class: Class, class: Attribute, class: Relationship, attribute: 'Rnum'...) and M3: meta-metamodel elements to define (ambitiously) any modeling language (class 'Model Node', class 'Structural Connection'). For our purposes, we will seldom refer to the M3 layer and M2 will be relevant only if the subject matter at hand is a metamodel.

Keep in mind that these layers may be interpreted relative to the subject matter at hand. If you are modeling an air traffic control system, 490.7 Liters is likely data in the M0 layer with the class 'Aircraft' at the M1 layer. 'Class' would be at M2 - the metamodel layer.

But if the subject matter is Executable UML, as is the case in the miUML metamodels, both the class 'Aircraft' and 490.7 Liters could be compressed into the M0 layer. ('Aircraft' is data populating the Class and Attribute subsystem and 490.7 Liters populated into the Population subsystem). At the M1 layer we would possibly have 'Class' and 'Value'. And, since the metamodel will populate itself, as we bootstrap into code generation, the M2 layer is also 'Class' and 'Value'.

In other words, we can create a data base schema from the miUML metamodel and then insert the metamodel itself into that schema. So if we define a table 'Class' based on our metamodel in the data base, we could then insert metamodel classes 'Class', 'Attribute', etc. into that table. Thus we see 'Class' both at the M0 (inserted data) and M1 (modeled) and M2 (metamodel) layers!

Local Data Types

The following data types are used by attributes in this subsystem.

Name

This is a string of text that can be long enough to be readable and descriptive. A handful of words is okay, a full tweet is probably too much. Since the exact value may change to suit a particular platform, it is represented symbolically as "LONG_TEXT".

Domain: String [LONG_TEXT]

Nominal

This is a whole number used purely for naming with no ordinal or computational properties.

Identifiers used only for referential constraints may be stripped out during implementation (assuming the constraints are still enforced). During simulation and debugging, however, these can be nice to have around for easy interpretation of runtime data sets without the need for elaborate debug environments. "File-6:Drawer-2:Cabinet-"Accounting" is easier for a human to read than a bunch of pointer addresses.

Domain: The set of positive integers {1, 2, 3, …}.

Description

This is an arbitrarily long amount of text. The only length limit is determined by the platform.

References

[MB] Executable UML: A Foundation for Model-Driven Architecture, Stephen J. Mellor, Marc J. Balcer, Addison-Wesley, 2002, ISBN 0-201-74804-5

[DATE1] An Introduction to Database Systems, 8th Edition, C.J. Date, Addison-Wesley, 2004, ISBN 0-321-19784-4

[LSART] How to Build Articulate UML Class Models, Leon Starr, Model Integration, LLC, Google Knol, http://knol.google.com/k/how-to-build-articulate-uml-class-models

[SM1] Object-Oriented Systems Analysis, Modeling the World in Data, Sally Shlaer, Stephen J. Mellor, Yourdon Press, 1988, ISBN 0-13-629023-X

[HTBCM] Executable UML: How to Build Class Models, Leon Starr, Prentice-Hall, 2002, ISBN 0-13-067479-6

[3RDMAN] Third Manifesto ...

Atomic Type ATOMTYPE

An Atomic Type is created by choosing a supplied Unconstrained Type such as _I, _R, _string, etc. and imposing further constraints as required by the application being modeled. For example, a modeler may define minutes by choosing the the Unconstrained Type _I, and establishing bounds 0-59. Thus an *Atomic Type* represents a type constraint established by the application (or some subject matter's) requirements.

The Operators defined on an Unconstrained Type serve as a reference. They may be copied wholly or selectively with any desired non-existent Operators added for the Atomic Type. An Atomic Type counter, for example, may be based on the Unconstrained Type _I, bounded 0-∞, with Update Operators ++, -- retained (increment/decrement), the Update Operator reset added (set to zero), and all other _I Operators excluded.

Some example Atomic Types are speed, pressure, integer, real, primary color, etc. It may be apparent that these examples are a mix of Domain and Common Types.

Each presupplied Unconstrained Type determines the specialization of any derived Atomic Types. The _string Unconstrained Type can be constrained by Symbolic Types whereas _I will be constrained by Integer Types. The modeler will create and maintain Atomic Types whereas Unconstrained Types are for reference only.

More complex Types may be constructed by creating Structured Types with Fields, each constrained by some Atomic Type.

Attributes

Name

Type.Name

Unconstrained type

Unconstrained type.Name – This Atomic Type further constrains this Unconstrained Type.

Identifiers

I> Name

Same as the superclass.

Relationships

R717

Atomic Type IS BASED ON *exactly one* Unconstrained Type
Unconstrained Type IS THE BASIS OF *zero, one or many* Atomic Type

An Atomic Type may impose further constraints upon an existing Unconstrained Type such as boundaries, operator exclusions, value exclusions, eligibility criteria, units, precision and so forth. In the simplest case, no further constraints are applied. For example, the _I Unconstrained Type may be minimally 'constrained' by the integer Common Type with no bounds or units defined and all Operators retained (copied across).

The relevant constraints that may be applied depends on the specific Unconstrained Type (_I vs. _string, for example) which determines the Atomic Type specialization (Integer vs. Symbolic). The Operators defined for an Unconstrained Type are available for reference by a derived Atomic Type. These Operators may be used, excluded or modified in the Atomic Type definition.

The Unconstrained Types will be supplied without necessarily being used by any particular Atomic Type.

R706

Atomic Type IS A Boolean, Integer, Real, Rational, Enumerated OR String Type

The intention is to keep Atomic Types platform independent, mathematical and disjoint so that the modeler has a precise and flexible set of building blocks for defining more complex Constrained Types, both System and Domain specific.

Each supplied Unconstrained Type defines an Atomic Type specialization. The modeler can then create constrained variations of each specialization as instances of Atomic Type.

Boolean Type BOOL

A *Boolean Type* enables the choice between two mutually exclusive conditions. When applied to an appropriately named Attribute, Open as a valve status for example, an evaluation such as if myValve.Open may be executed.

Based on the Unconstrained Type: _boolean

Example Operators: toggle

Unlike other Atomic Types, the Boolean Type has no constraint modifiers, so one size fits all. A single Boolean Type named boolean should be adequate. The only reason to create more than one instance of Boolean Type would be to provide alternate names such as bool, binary, etc.

All Boolean Types default to false as an initial value.

Is the Boolean Type really required? Why not just configure an Enumerated Type with two Enum Values? Or, for that matter, an Integer Type bounded 0-1. Two reasons. First, the forthcoming action language will recognize the special nature of boolean variables so that true/false tests may be performed concisely. Second, certain Operators can be defined on a Boolean Type that don't apply to an Integer or Enumerated Type such as toggle.

ATTRIBUTES

Name

Atomic Type.Name

IDENTIFIERS

I> Name

RELATIONSHIPS

None.

Constrained Type CONTYPE

A Type defined by a modeler, either domain specific or system wide, is a *Constrained Type*. It establishes a constraint on the values that may be assigned to an Instance's Attribute, variable or parameter. The specific constraint is established by the requirements in a particular Domain or by common system wide needs. Each Constrained Type is derived from one or more Unconstrained Types (via Atomic Types).

The 'user data type' described in [MB] corresponds to Constrained Types defined for a particular Domain (Domain Types).

Attributes

Name

Type.Name

Identifiers

I> Name

Unique by policy for the system as a whole (spanning all Domains).

Relationships

R700

Constrained Type IS AN Atomic OR A Structured Type

A Type is either fundamental (in the sense that there are not really any extractable components) or it is a hierarchical structure built up from more fundamental Types.

R708

Constrained Type IS A Domain OR A Common Type

A Type is either domain specific type or it is available system wide. Domain Types are defined to suit the subject matter relevant to a particular Domain. A Common Type, on the other hand, is generally applicable in many contexts and so is made available across all Domains.

Both Domain and Common Types may be defined by the modeler.

Domain Type DOMTYPE

A Type specific to the subject matter in a Domain is a *Domain Type*. In a car navigation application, for example, a Domain Type might be GPS Position. In an elevator control system, floor name or service direction might be Domain Types. In a cardiac pacing application, pace mode could be a possibility. The name of a Domain Type is often the same as the Attribute it constrains.

By definition, a Domain Type is defined locally within a Domain and is not available to constrain the Attributes in any other Domain. This makes sense since Types required for Shutter.Speed and Vehicle.Speed probably belong in different Domains. (If not, the Type names would have to be different).

Attributes

Name

Constrained Type.Name — Ultimately refers to Type.Name which fully qualifies the name to avoid any naming conflicts between Domain and Common Types.

Local name

A Name that is descriptive within a Domain and often matches the name of an Attribute such as speed, height, pressure, heading, etc.

Type: name

Domain

Domain.Name – The Type is defined locally within this Domain.

Element

Spanning Element.Number – This Domain Type is a Spanning Element since it is available across Subsystems within its Domain.

Identifiers

1> Name

Unique since this is ultimately derived from identifier 3, via the Type class identifier.

2> Domain + Element

Element numbers are unique by Domain.

3> Local name + Domain

Domain Types are named uniquely within each Domain.

RELATIONSHIPS

None.

Enum Value EVAL

An *Enum Value* is a value that may be assigned to an Enumerated Type.

Attributes

Name

An Enum Value's name is a unit of data without units or dimension. In an Enumerated Type such as compass direction, the names might be north, south, east and west.

Type: short name

Enumeration

Enumerated Type.Name

Identifiers

I> Name + Enumeration

Types are disjoint. That is to say that a value is always characterized by exactly one Type. A value like red might be represented by two different Enumeration Types such as primary color and traffic signal. But the value red-primary color is considered a completely different value from red-traffic signal.

Relationships

R707
Enum Value IS ASSIGNABLE IN *exactly one* Enumerated Type
Enumerated Type HAS ASSIGNABLE *one or many* Enum Value

An Enum Value is defined as part of an Enumerated Type. It has no meaning outside of its Type.

An Enumerated Type is a set of Enum Values. The set must include at least one Enum Value so that a value can be assigned to any Attribute or variable the Type might describe. It is possible to define an Enumerated Type with only one Enum Value, but the utility of a single value enumeration is not apparent.

Enumerated Type ENUM

Much like its implementation counterpart, an *Enumerated Type* specifies a set of assignable values. Enumerated Types are unordered and there is no ordinal associated with each Enum Value. In other words, an Enum Value is known only by its Name.

Based on the Unconstrained Type: _enum

Attributes

Name

Atomic Type.Name and Enum Value.Enumeration – The default value is constrained so that it is one of the Enum Values defined by this Enumerated Type (not just any Enum Value).

Default value

Enum Value.Name

Identifiers

I> Name

Relationships

R715

Enumerated Type DEFAULTS TO *exactly one* Enum Value
Enum Value IS THE DEFAULT VALUE OF *zero or one* Enumerated Type

Each Enumerated Type must designate one of its Enum Values as a default or initial value.

Any given Enum Value may or may not be that default value.

Field

A *Field* represents the inclusion of a Constrained Type at some level within a Structured Type. It also represents a terminal node within a Structured Type, though any substructure could may be further defined by another Constrained Type. Consider the Structured Type point with two Fields named x and y.

```
point ( rational x, rational y )
```

Now consider an instance of a class Cursor with an Attribute Position declared to be of type point. The value thisCursor.Position is *a single value with no user visible components*. That said, selection Operators may be defined on a Structured Type that return values of interest.

It is kind of a placeholder now, so the miUML Type subsystem does not fully support the features that follow, but here is the direction we are going...

For example, two selector Operators will be defined to access the values so that it will be possible to refer to thisCursor.Position.x and thisCursor.Position.y. This syntax may change in the forthcoming action language. Here the . Operator corresponds to the THE_ selector defined in [3RDMAN]. Now that may appear as if the components of the type point are visible, but consider the following...

Operators may further be defined on the Type point to return polar coordinates so that it will be possible to refer to thisCursor.Position.r and thisCursor.Position.Θ. These aren't necessarily structural components since they may have been computed. The point (no pun intended) is that from the perspective of the action language, it will be impossible to tell the difference. All this is described under 'possible representation' or 'possrep' in [3RDMAN].

Attributes

Name

Member.Name

Structured type

Member.Type — The name of the Structured Type of which this Field is a part.

Type - constrained

Constrained type.Name — Careful: 'Constrained' (type) and 'constrained' mean different things here! Welcome to metamodeling ;) This is the Type of the Field itself. It is constrained so that there can be no cycles in a Structured Type and so that any ultimate leaf node in an amalgamation of Structured Types resolves to an Atomic Type. This means that the value of this referential attribute may not match the self.Structured type value and no descent through the referenced Constrained Type may reach the Structured type value either.

IDENTIFIERS

I> Name + Structured type

Same as the superclass.

RELATIONSHIPS

R705
Field IS PLACEMENT OF VALUE CONSTRAINED BY *exactly one* Constrained Type
Constrained Type CONSTRAINS VALUE STORED IN *zero, one or many* Field

A Field specifies a location within a Structured Type where a value of a Constrained Type is stored. While a Field is a terminal node within a Structured Type, if its assigned Type is non-atomic, further substructure will be defined in the assigned Type (Structured Type) definition. Ultimately, all Fields in an amalgamation of Structured Types must resolve to Atomic Types.

In the example of a point Structured Type, the same Constrained Type, coordinate, might constrain two Fields named x and y. The Structured Type lineseg, on the other hand, breaks down into two Fields, each of Structured Type point.

A Constrained Type may be used on its own, without being incorporated in any Structured Type.

Integer Type INT

The *Integer Type* represents the mathematical set I of integers. The set can be left infinite in the model and made finite on a given implementation or it can be restricted to specific values as directed by the application requirements. If the modeler specifies infinite on a boundary, he or she is indicating that, as far as the application is concerned, there is no specific boundary. Limits will always be imposed based on the chosen platform for an implementation.

Example Operators: +, -, mod, *

Based on the Unconstrained Type: _I

Attributes

Name

Atomic Type.Name

Units

Can be none or any unit of measurement relevant to the application such as m/sec, degrees C, psi, etc.

Type: name

Lower limit

The minimum value that may be assigned.

Type: I inf

Upper limit - constrained

The maximum value that may be assigned. Must be greater than or equal to the Lower limit.

Type: I inf

Default value - constrained

This value is assigned initially and by default (when no other value is specified). Must be greater than or equal to the Lower limit and less than or equal to the Upper limit.

Type: I

Identifiers

I> Name

Same as superclass.

Relationships

None.

Member MEM

A component of a Structure is called a *Member*. This can be either a substructure or a lowest level Field defined by a Constrained Type.

```
lineseg ( point p1, point p2 )

point ( rational x, rational y )
```

This lineseg (line segment) Structured Type in the example above consists of two Members, each a Field of Structured Type point. This Type further breaks down into two Members (x and y), each of which has the Atomic Type rational.

Attributes

Name

A descriptive name such as point or color temperature.

Type: name

Parent structure

Structure.Name — The next highest level Structure above this Member or none (if it is the topmost Member.

Type

Structured Type.Name and Structure.Type — This member is part of this Type. IT is the same Type as that of the Parent structure, unless the Parent structure is none.

Identifiers

I> Name + Type

Members are named uniquely within a Type.

Relationships

R701
Member IS LEVEL OF ORGANIZATION WITHIN *exactly one* Structured Type
Structured Type IS HIERARCHICALLY ORGANIZED BY *one or many* Member

In the simplest case, a Structured Type will consist of a single Field within a single Structure. So there will always be at least two Members composing any Structured Type. A Structure may be built up from many Structures with Fields at various levels.

A Member can be defined only within a Structured Type.

R703
Member IS A Field OR A Structure

A Member of a Structured Type is either a Field, which represents the lowest possible level, or a Structure which is either an intermediate or top level Member.

R704
Member IS PART OF *zero or one* Structure
Structure IS COMPOSED OF *one or many* Member

A given Member may be the highest level Structure within a Structured Type in which case it is not part of any Member. Otherwise, it must be situated within some Structure. Since a Structured Type is organized hierarchically, a Member can have only one parent Structure.

A Structure cannot be empty. It must be broken down into at least one deeper Member.

Operator OP

An Operator is a function that may be defined for one or more Types. An Operator like + is defined for the _I Type, for example. It takes two Operands of type _I and returns an _I. The same Operator is also defined separately for the _R, _Q, and _string Types.

An Operator may take Operands of different Types. For example, the Operator 'PAD' might be defined to pad a _string with a certain number of trailing space characters. So there would be one _string Operand and a posint Operand returning a _string. This is clearly an Operation on a _string even though there is an Operand of Type posint participating in a modifier role.

An Operator may take an Operand not of the Operator Type which generates a return value of the Operator's Type. For example, RAND(5) might produce a _R as a return value using a pos-int seed.

Attributes

Name

A descriptive name applicable to multiple Types such as add, multiply, increment, etc.

Type: name

Symbol

A short, often single character, name to represent an Operator in a written expression. Common examples are +, -, *, ++, next, etc.

Type: short name

Identifiers

1> Name

Operator names are unique system wide.

2> Symbol

The Symbol + can mean something different for real vs integer addition, since in each case there is separate Supported Operator. In this sense, an Operator is polymorphic. But there is only one +.

Relationships

None.

Operand OPND

An input parameter defined for a Typed Operator is an *Operand*. The + Operator defined on the _I Type has the Operands i1 and i2 as shown:

```
_I ( _I i1 + _I i2 )
```

The Operand Types are not always identical. For example, an Operator like scale might take an Operand of Type vector and another of Type integer to yield a vector result.

Attributes

Name

A short, descriptive or numeric name such as i1, i2, root, scale, etc.

Type: short name

Operator

Typed Operator.Operator – The Operator that processes this Operand.

Operator type

Typed Operator.Type – The Type upon which the Operator is defined.

Operand type

Type.Name – The Type of this Operand which may or may not match the Operator type.

Identifiers

I> Name + Operator + Operator type

Each Operand is named uniquely for its Typed Operator

Relationships

R713
Typed Operator OPERATES ON *zero, one or many* Operand
Operand IS OPERATED ON BY *exactly one* Typed Operator

A Typed Operator may not have any Operands at all. Consider the nominal type which provides unique, non-mathematical, numeric names for Identifier Attributes. An Operator such as next might be defined which takes no Operands, but returns a nominal value as the next unused number in sequence.

An Operand is really part of the definition of a Typed Operator, so it has no meaning outside that context.

R718
Type CONSTRAINS VALUE OF *zero, one or many* Operand
Operand HAS VALUE CONSTRAINED BY *exactly one* Type

Each Operand represents an input value and a value, by definition [3RDMAN], must be typed. A Type may or may not constrain any Operands.

Usually, an Operand will be the same Type as its Typed Operator, but not necessarily. In such a case, the differently typed Operand is playing the role of a modifier on some base value such as scaling a vector or extending the length of a string. Or perhaps the Operand is simply seeding an expression that yields a value of the Typed Operator's Type such as `_R : RAND(_I seed)`.

Rational Type RAT

The *Rational Type* represents the mathematical set Q of rational numbers. The set can be left infinite in the model and made finite on a given implementation or it can be restricted to specific values as directed by the application requirements.

The modeler should take care to specify the rational Type when this is indeed the platform independent application requirement and let the programmer/compiler choose the appropriate implementation type or types to suit the platform and performance requirements.

Example Operators: +, -, /, *

Based on the Unconstrained Type: _Q

Attributes

Name

Atomic Type.Name

Units

Can be none or any unit of measurement relevant to the application such as m/sec, degrees C, psi, etc.

Type: name

Precision

The number of required decimal places to the right. Zero effectively specifies an integer.

Type: posint

Lower limit

The minimum value that may be assigned.

Type: Q inf

Upper limit - constrained

The maximum value that may be assigned. Must be greater than or equal to the Lower limit.

Type: Q inf

Default value - constrained

This value is assigned initially and by default (when no other value is specified). Must be greater than or equal to the Lower limit and less than or equal to the Upper limit.

Type: Q

Identifiers

I> Name

Same as the superclass.

Relationships

None.

Read Only Operator ROOP

A *Read Only Operator* processes its Operands and produces a result of some Type without modifying the Operands in any way. Example:

```
_string ( _string s1 + _string s2 )
```

In the example above, the + Operation is defined on _string with a return result of Type _string. The specification of a return value establishes a Typed Operator as read only.

Attributes

Operator

Typed Operator.Operator

Type

Typed Operator.Type

Identifiers

I> Operator + Type

Same as superclass.

Relationships

R719
Read Only Operator RETURNS VALUE OF *exactly one* Type
Type DEFINES VALUE RETURNED BY *zero, one or many* Read Only Operator

By definition, a Read Only Operator must return a value. And, also by definition [3RDMAN] a value has a distinct Type. So a Read Only Operator does return a value of some Type.

A given Type may or may not be returned by any Read Only Operators. It is possible to imagine, for example, a Type like post office address that is not returned by any particular Operator. And a common Type like posint could be returned by many Read Only Operators.

Real Type REAL

The *Real Type* represents the mathematical set R of real numbers. The set can be left infinite in the model and made finite on a given implementation or it can be restricted to specific values as directed by the application requirements.

Example Operators: +, -, /, *

Based on the Unconstrained Type: _R

Attributes

Name

Atomic Type.Name

Units

Can be none or any unit of measurement relevant to the application such as m/sec, degrees C, psi, etc.

Type: name

Precision

The number of required decimal places to the right. Zero effectively specifies an integer.

Type: posint

Lower limit

The minimum value that may be assigned.

Type: real inf

Upper limit - constrained

The maximum value that may be assigned. Must be greater than or equal to the Lower limit.

Type: real inf

Default value - constrained

This value is assigned initially and by default (when no other value is specified). Must be greater than or equal to the Lower limit and less than or equal to the Upper limit.

Type: real

Identifiers

I> Name

Same as superclass.

Relationships

None.

Structure STRUCT

A branch within the hierarchy of a Structured Type is called a *Structure*. It is strictly an organizational component.

```
waypoint :
    latitude :
        degrees : lat_degrees
        minutes : minutes
        seconds : seconds
    longitude :
        degrees : long_degrees
        minutes : minutes
        seconds : seconds
    elevation : altitude
```

In the example above, a Structured Type named waypoint is defined with top level Structures latitude and longitude and one top level Field height. Both the latitude and longitude structures are broken down into three Fields each.

The Structures may be replaced in waypoint by a location Field if a Constrained Type called surface position is defined to include the latitude and longitude Structures and then associated with the Field. (And even those Structures may be eliminated by defining Constrained Types latitude and longitude and referencing those from surface position.

Attributes

Name

Member.Name

Type

Member.Type

IDENTIFIERS

I> Name + Type

Same as the superclass.

RELATIONSHIPS

None.

Structured Type STRUTYPE

The relational model as described in [3RDMAN] accommodates complex Types including those that are hierarchically structured. A *Structured Type* is just such a hierarchical type similar, but not quite the same, as those found in most programming languages. From a relational perspective, a value defined by a Structured Type is a single, unparseable unit of data just like any other value. That said, a component of a Structured Type may be accessed, but only through one or more selector Operations defined on the Type. See the description of Field for more about how miUML structures are not quite the same thing as conventional programming language structures.

Attributes

Name

Constrained Type.Name

Identifiers

I> Name

Same as the superclass.

Relationships

None.

Symbolic Type SYMTYPE

The *Symbolic Type* is introduced in [OOADATA] and [MB] as a platform independent type constraint on the string implementation type.

Example Operators: +, length, >, <

Based on the Unconstrained Type: _string

It could be argued that the String Type is not necessarily an Atomic Type as it is made up of individual characters. But would a character type be useful in the context of platform independent requirements specification? (It is no doubt useful as an implementation type). And a single character could be supported by creating a Symbolic Type named alphachar, for example, with a length = 1, and a validation expression. It is the job of the compiler or programmer to map such a type constraint to the best char implementation type on the targeted platform.

Attributes

Name

Atomic Type.Name

Min length

The minimum number of characters that must be present. This includes any prefix and suffix contribution.

Type: posint

Max length - constrained

The maximum number of characters that may be present. This includes any prefix and suffix contribution.

This value is constrained to be greater than or equal to the Min length plus the combined length of any suffix and prefix contribution.

Type: posint

Prefix

A string value that will be inserted at the beginning of the value.

Type: string

Suffix

A string value that will be appended to the end of the value.

Type: string

Validation pattern

If this regular expression successfully identifies (finds) a proposed value (without the prefix or suffix) and the min/max length limits are not violated, then the value is valid.

Default value - constrained

A valid value that like any assignable value is accepted by the validation pattern and min/max length limits. This value will be automatically assigned to any variable or Attribute defined by this Symbolic Type when no other value is available.

Identifiers

I> Name

Same as the superclass.

Relationships

None.

Common Type COMTYPE

Domain Types are not the only Types definable by the modeler. A Constrained Type that is universally applicable across Domains, such as nominal, posint, name, short name, description, date, duration and count is a *Common Type*. The usual term 'system data type' as described in [MB] and elsewhere is not quite applicable for a few reasons: 1) The concept of 'system' can be a bit vague in a platform independent modeling language. 2) The modeler should have the ability to declare and modify any Constrained Type since, what may be common to one modeler or project might not be so common to another. 3) The notion of 'system defined' imposes the challenging and perhaps impossible requirement of declaring a complete set of generally useful Constrained Types for all developers.

A Common Type such as integer may be created by copying the Unconstrained Type _I to create a Constrained (in name only) Type named integer with units = none, Lower limit and Upper limits = infinite and default value of 0. In this manner, all Unconstrained Types may be copied into Common Type versions if desired.

In fact, it is probably a good idea to offer a 'starter set' of Common Types pre-populated as a convenience for the modeler. The Common Type nominal, for example, can supply values for Identifier Attributes. It can be configured as an instance of Integer Type with Units = none and Precision = 0 and an Operator such as next for generating a unique value.

Attributes

Name

Constrained Type.Name

Identifiers

I> Name

Same as the superclass.

Relationships

None.

Type

A *Type* is a named finite or infinite set of values. This differs a bit from the definition in [3RDMAN] where C.J. Date defines it as follows: 'A named, finite set of values.' He goes on to clarify that a mathematically infinite set, such as integers, will be characterized as the 'set of all integers that are capable of representation in the computer system under consideration'. In miUML this means that the upper and lower boundaries of a Type such as integer will be established possibly differently on each target platform. But as far as the platform independent models are concerned, the set of integers is truly infinite. So the miUML Type matches the [3RDMAN] definition only for an implementation.

Every value in miUML is characterized with a Type. Each Attribute is constrained by a Type. Therefore any value assigned to an Attribute of an Instance must belong to the declared Type. Furthermore, any parameter or variable defined in the action language will be similarly typed so that only corresponding values may be assigned. There is no such thing as a type-less value or variable in miUML.

Types are disjoint in that a value cannot simultaneously be described by more than one Type. That said, the physical representation of two values, each defined by a distinct Type may be identical. A postal code such as 94117, for example, might be representable by an integer or an enumeration or a string. But 94117 of the Type postal code is not the same value as the integer 94117 even though they may appear similar.

Here are some other relevant quotes from [3RDMAN]:

'... types are not limited to simple things like integers. Indeed, ... values and variables can be arbitrarily complex.'

'a type might consist of geometric points, or polygons, or X-rays, or XML documents, or fingerprints or arrays or stacks or lists ...'

At present, the typing system provided by miUML is a primitive placeholder so that work on the action language can proceed. The intention, though, is to ultimately provide support for arbitrarily complex types, such as multidimensional matrices, type inheritance and even relations eventually.

ATTRIBUTES

\Name

A descriptive name created as follows: If it is a Domain Type, the local type name and Domain name are concatenated with a '.' delimiter. If it is a Common Type it is just its name.

Name = [Domain Type.Domain + '.' + Domain Type.Local name | Common Type.Name]

Type: name

(The name type excludes common delimiters such as '.')

IDENTIFIERS

I> Name

Type names are unique system wide (across all Domains).

RELATIONSHIPS

R714
Type IS AN Unconstrained OR Constrained Type

In truth, every Type is a constraint in that it defines a set of assignable values. (The set of integers does not include any strings). So the term 'unconstrained' actually means that there are no modeler specified constraints. Therefore, a Type is either modeler defined (Constrained) or not because it is mathematically constrained (Unconstrained).

The unbounded set of all integers, _I, for example, would be an Unconstrained Type. A modeler will often constrain this set to suit the subject matter. A modeler might create the Constrained Type time minutes, for example, where _I is constrained with bounds of 0-59, units of minutes.

The 'core data type' described in [MB] corresponds most closely to an Unconstrained Type where the 'user data type' in [MB] is really a Constrained, Domain Type. (The Constrained Common Type is an extra feature provided by miUML).

Typed Operator TYPO

An Operator is polymorphic in that it may be applied to multiple Types. The + Operator, for example, means something different in the _I, _R and _string contexts. Each such context is represented by a *Typed Operator* with its own definition, set of typed Operands and a possible return value.

Attributes

Operator

Operator.Name

Type

Type.Name

Definition

This is a precise specification of the processing required to transform the Operands to produce either a return value or an update result, depending on the Typed Operator specialization.

At present, miUML does not provide an Operator definition language which, most likely, will be expressed using action language. For now, use pseudo-code such as the following:

```
c.value = min( c.ceiling, c.value + 1 ) // Increment counter unless ceiling has been
reached
```

The above definition assumes that the Operator is ++ with one operand named c of Type counter, a Structured Type with integer components value and ceiling. Since a result is assigned to the Operand, this must be an Update Operator.

Type: pseudocode

Identifiers

I> Operator + Type

An Operator like + only means one thing in the context of _I, as determined by the Definition, for example.

Relationships

R712

Type ESTABLISHES CONTEXT FOR DEFINITION OF *one or many* Operator
Operator IS USED IN A CONTEXT ESTABLISHED BY *one or many* Type

An Operator has no utility unless it is relevant to at least one Type. The same Operator may be used with many Types making it polymorphic. The + Operator, for example, could be defined differently for _I, _R and _string.

A Type has no utility without at least one Operator. Since assignment and equality must be defined for all Types, there should at the very least be two Operators defined for any given Type.

Not yet sure how to handle automatic, required Operators such as assign, get, set. These should be handled just like any Operators except that they will be generated and probably non-modifiable. Equality is also an interesting case which should be modifiable and possibly automatic. This will likely be sorted out with the rest of the action language.

R716

Typed Operator IS AN Update OR A Read Only Operator

A Typed Operator either modifies its Operands or it simply references them to produce a result. This policy is inspired by [3RDMAN] and is designed to avoid side effects.

Unconstrained Type UNTYPE

Mathematically based *Unconstrained Types* are the fundamental building blocks for specifying more constrained and complex Types. The set of all integers, I, for example, is unbounded and without units in its platform independent and mathematical form. It can be represented with an Unconstrained Type named _I. A movie rating system on a scale from 1 to 5 may be created by creating a Constrained Type based on _I with bounds from 1-5 and specifying optional units, such as stars or thumbs up.

The other Unconstrained Types are: _I, _R, _Q, _symbolic, to represent the constrainable integer, real, rational and symbolic sets, respectively, as well as _boolean (sure, constrainable also, but why?) and _enum (user definable). Names such as _I are used instead of a programmer-friendly _int to underscore (no pun intended) mathematical rather than implementation semantics.

The correspondence between these names and the Atomic Type specializations is not coincidental. Each Atomic Type is created by copying initial qualities from a base Unconstrained Type. Specializations are required since the fundamental operators and constraining properties are different for each Unconstrained Type. Precision, for example, is relevant only to real and rational numbers. A toggle operator is relevant only to boolean values. This means that the complete set of Unconstrained Type instances is bound at model-time, so they are not editable.

A set of Operators must be defined for every Type. Certain Operators are defined for multiple Unconstrained Types such as assignment and addition. It would be a bit of a burden if the modeler were required to define addition for each new Integer Type. To make life easier for the modeler, common Operators will be predefined and non-editable for each Unconstrained Type. A new Atomic Type is created by copying the relevant Operators from its base, Unconstrained Type. Since Unconstrained Types are set in stone (non-editable), the modeler can always depend on a a full set of reference-able base Types and Operators.

For now, copying serves as a "poor man's inheritance". In the future it would be nice to have a full type inheritance model.

Attributes

Name

Type.Name

Identifiers

I> Name

Same as the superclass.

Relationships

None.

Update Operator UPDOP

An *Update Operator* modifies its Operands and does not return a result. No result is returned to avoid the well documented nastiness of side effects.

Example: with i as Type _I, ++i increments the value of i without returning a result.

Attributes

Operator

Typed Operator.Operator

Type

Typed Operator.Type

Identifiers

I> Operator + Type

Same as superclass.

Relationships

None.

Updating Operand UPOP

This is an Operand of an Update Operator whose value is modified during execution of the associated Operation.

Modeling note: This association class is required since a conditional reference from Operand, Operator + Type, would always have a value via R713, updating or not. On the other hand, a Read Only Operand will simply not exist as an instance of Updating Operand. The only other option is to specialize Operand into Read Only and Updating.

Attributes

Operand

Operand.Name

Operator

Update Operator.Operator and Operand.Operator — The Operand is part of this Update Operator.

Operator type

Update Operator.Type and Operand.Operator Type — Both the Operand and the Update Operator are bound to the same Type.

Identifiers

I> Operand + Operator + Operator type

Standard association class composition for a 1x:Mx association using the reference from the many-side as the identifier.

Relationships

R720
Operand IS UPDATED BY *zero or one* Update Operator
Update Operator UPDATES *one or many* Operand

By definition, an Update Operator must modify at least one Operand. Ordinarily, there is only one, as is the case with the ++ (increment) Operator. But it is certainly possible to define an Operator that modifies multiple values. Furthermore, a non-updating Operand may be present in modification role, such as scale.

A given Operand may or may not be updated by an Update Operator. Certainly it won't be if it is part of a Read Only Operator. And if it is part of an Update Operator, it might just be a modifier.